Theory Recap via Questions

Q: What is the difference between Supervised and Unsupervised Learning?

- **Supervised Learning (SL):** In supervised learning, the algorithm is trained on a **labeled** dataset $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$ where $x^{(i)}$ is the feature vector where each component is called **feature**, **attribute,** or **covariate**, and $y^{(i)}$ is the corresponding **target** (or **label**) that the model aims to predict.
  The goal is to learn a mapping function from inputs to outputs, allowing the model to make accurate predictions on new data. The best-known tasks belonging to this class of problems are:
  - **Regression**: Predicts a continuous value.
  - **Classification**: Predicts a discrete category or class
- **Unsupervised Learning (USL)**: In **unsupervised learning**, the algorithm learns from unlabeled data:

  $$D = \{x^{(i)}\}_{i=1}^{m}$$

  Unlike supervised learning, there are no predefined target outputs. Instead, the model identifies patterns or structures in the data. The most common unsupervised learning tasks include:
  - **Clustering**: Groups similar data points together.
  - **Dimensionality Reduction**: Reduces the dimensionality of the data (number of features) while preserving essential information.

Q: What is the difference between discriminative and generative models?

- **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.

Q: What is a Linear Regression model, which is its purpose and how the error for such a model is measured?

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 ML context the independent variables are the features and the dependent variable is the target variable.

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

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

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

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

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 strength of the linear relationship between the input x and the output y.

If we deal with linear data, a linear regression once trained (i.e., when the optimal parameter values are determined) allows us to predict a value near to the target variable value based on input features.

A classic example of application of linear regression is the Housing price prediction where the goal is to estimate the price of a house (target) given features 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 and the features. To find these best parameters we need a way to assess how close our predictions are to the actual target values and for that we can use a **cost function**. The cost function quantifies the error between the predicted values and the target values. The cost function generally adopted for linear regression 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:

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

where m is the number of training examples in our training set.

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

$$\theta_{min} = argmin_\theta(J(\theta))$$

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

There are different ways to reach this minimum: one way is using an optimization algorithm called **gradient descent**.

Q: Talk about Gradient Descent, also explaining what is the purpose of the learning rate.

Gradient Descent (GD) is an optimization technique that helps us **iteratively** adjust the parameters of a model to find the minimum of 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.

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

The steps to perform gradient descent are as follows:

1. **Initialize the Parameters**: We randomly initialize the θ values.
2. **Update the Parameters**: Each parameter is updated based on the partial derivative of the cost function w.r.t. to that parameter:

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

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

- 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.

The process is repeated iteratively until the cost function converges.

It's important to underline that first we compute all the partial derivates and then once computed all we can proceed with the parameters' updates.

Note: The partial 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 parameters update formula.

Q: Show how is obtained the explicit form expression of the partial derivative of the MSE cost function w.r.t. a parameter $\boldsymbol{\theta_j}$.

Let's consider the multiple linear regression (the same holds for simple 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 $x_0^{(i)} = 1$. The MSE cost function is given:

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

Since we need the partial derivatives of the cost function with respect to any $\boldsymbol{\theta_j}$, we differentiate as follows:

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

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

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

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

where:

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

→ 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:

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

and you can then use it within the GD update.

Q: Explain the advantages of Mini-batch GD over Full-batch GD and SGD and also the differences between these (write pseudocode for each of them).

Batch Gradient Descent (BGD) use **all** training samples in the training set to calculate the gradient and perform a single update of the parameters:

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

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:

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

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.

To achieve a balance between the stability of BGD and the efficiency of SGD, another technique commonly used is the **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 $\boldsymbol{b}$ randomly chosen training samples ($\boldsymbol{1 < b \ll m}$)** to compute the gradient and update the parameters:

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

The cost per iteration for MBGD is $O(b)$, which depends on the **mini-batch size $\boldsymbol{b}$**.

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.

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;
    }
}
```

Q: List the Gradient Descent stopping conditions.

Gradient descent algorithm converges to the minimum after an **infinite number of iterations**. In practice, we define a **stopping criterion** to decide when to terminate the algorithm. Common stopping criteria include:

- **A fixed number of iterations**: the algorithm stops after a fixed number of steps.
- **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.

Q: What is Polynomial Regression?

Polynomial regression is an extension of **linear regression** that allows modeling **non-linear relationships** between input features and the target variable by introducing polynomial terms. More specifically, in polynomial regression, we perform some transformations on the input feature(s) so to obtain higher-degree polynomial terms that we also add to the input. In this way, via the added polynomial terms we allow the model to capture non-linear relationships in the data that a simple linear regression can't handle

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 $\boldsymbol{t}$.

Basically, instead of only using x, we also include $x^2, \ldots, x^t$ as additional features.

However, 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.

Q: What is Min-Max and Z-Score Normalization, and why are they important?

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 (typically [0,1] or [-1,1]).

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.
- Moreover, if new data falls outside the original $[min, max]$ range, it may lead to inconsistencies (e.g., during the prediction phase).

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

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

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**.

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.

+ Both techniques have their place—min-max normalization is ideal for bounded data, while Z-score normalization is preferred when dealing with data with varying scales.

Feature normalization basically performs **shifting and scaling** of the features which is crucial for machine learning because:

- 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.
- Prevents large-scale features from dominating smaller ones, in fact all of them are transformed in a small similar range.

Q: Explain what Normal Equations method is and show also its derivation.

The **Normal Equations** is a method to directly compute the optimal parameters θ (the ones that minimize the cost function $J(\theta)$) by solving a system of equations derived in closed form setting the gradient of the cost function $J(\theta)$ to zero and solving for θ. This is an alternative to the Gradient Descent which instead finds these best parameters iteratively.

Normal Equations are specifically suited for solving the Ordinary Least Squares (OLS) problem in linear regression, which, instead of minimizing the MSE, minimizes the SSR:

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

The only difference between the two is the constant term $1/m$, in fact MSE is nothing more than a scaled version of SSR. **Minimizing the SSR in OLS also minimizes the MSE** and both approaches yield the same optimal parameters.

Said that, let's consider:

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

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.

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:

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

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:

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

We recall that the explicit form expression for the partial derivatives of the MSE cost function $J(\theta)$ w.r.t. to any θ_j is given by:

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

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

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

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:

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

which is basically the dot product between the transpose of the input matrix X and the residual vector. Thus, the gradient becomes:

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

Further expanding this:

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

Now wanting 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 $J(\theta)$.

Normal Equations method provides an **exact solution**, however to work, $X^T X$ must be **invertible**. Moreover, computing the inverse of the matrix $X^T X$ has a time complexity of $O(n^3)$, therefore this becomes very computationally expensive as the number of features increases. Normal equations method is efficient for smaller datasets.

Q: Derive the cost function of a linear regression using a probabilistic approach.

A: In linear regression, the model predicts the target values y based on inputs x using a hypothesis $h(x) = \theta^T x$. 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.

Now we show that under a specific set of **probabilistic assumptions**, 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.

Now, since our goal is always to maximize the probability that the predicted outputs are as close as possible to the actual outputs (target values), 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.

By the independence assumption on the $e^{(i)}$'s (and hence also the $y^{(i)}$'s given the $x^{(i)}$'s), this 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 $1/\sigma^2$ (since it is just a scalar):

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

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** of the data (i.e. doing maximum likelihood estimation (MLE)).

Q: What is meant by classification task?

A **classification task** refers to predicting a **discrete category or class** as the output, rather than a continuous value (as in regression). The goal is to assign each input to one of several **predefined classes** based on its features. 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.

Classification involves setting a **decision boundary** such as a threshold on the basis of which, depending on whether the predicted values are below/above it, it is possible to divide them into different classes.

For instance, in a **binary classification** task, a model might set a threshold such that:

- If $h(x) \geq 0.5$, , the instance is classified as **class 1**
- If $h(x) \leq 0.5$, it is classified as **class 0**

Instead of making a rigid yes/no decision, we could choose a model that reflects the inherent **uncertainty** in predictions which is linked to a probabilistic formulation of the type:

- 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

A such model is the **logistic regression**.

Q: Explain the concept of logistic regression and a detailed derivation of its cost function.

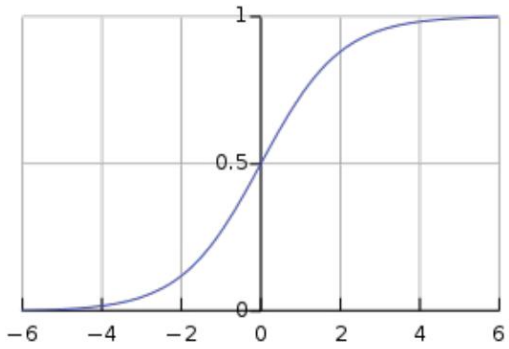
Logistic regression is a supervised learning algorithm used for classification tasks, particularly **binary classification**. Unlike linear regression, which predicts continuous values, logistic regression estimates the **probability** that a given input belongs to the positive class. This probability is then used to make a classification based on a threshold (usually 0.5).

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, $y\_pred \in (0,1)$.

Therefore, for an input feature vector x and parameters θ, the logistic regression model estimates the **probability** that $y = 1$ (appertains to the positive class):

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

Since there are only two possible classes (positive and negative), 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:

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

To obtain the best parameters we can perform a **Maximum Likelihood Estimate** (**MLE**), i.e. find the best parameters θ that produce the most probable output given the input (maximize the likelihood).

Assuming that training samples are **independent and identically distributed (i.i.d.)**, we express the likelihood function as:

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

Since y can have only two possible outputs, 0 and 1:

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

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

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

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

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, 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**.

Q: Show how is obtained the explicit form formulation of the partial derivative of the BCE cost function w.r.t. a parameter θ_j.

To minimize the BCE 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:

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

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

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:

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

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

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

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

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

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

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

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

Therefore:

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

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

Q: Explain the One-vs-All strategy for multiclass classification

Multiclass classification refers to the task of categorizing input data into one of **multiple** predefined classes. However, many classification algorithms, such as **logistic regression**, are inherently designed for **binary classification** (i.e., distinguishing between two classes). To extend these methods to multiclass settings, one common approach is **One-vs-All (OvA)**.

The **One-vs-All** strategy transforms a **multiclass classification** problem into **multiple binary classification** problems. Instead of directly predicting one class out of many, we train **one binary classifier per class**, treating that class as the **positive class** and all other classes as a single **negative class**.

Given a dataset with **K classes**, OvA trains **K** separated **binary classifiers**, each focusing on one specific class versus all the others. More specifically, for each class **k**, a separate classifier is trained using the following labels:

- **Positive class (1):** Instances belonging to class **k**
- **Negative class (0):** Instances belonging to any other class

During inference (prediction), each classifier outputs a probability score (or confidence level) for its class. The final prediction is determined by **choosing the class with the highest confidence score**.

Q: Explain the concept of model fitting in machine learning and the role of bias and variance.

"**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 "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.

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:

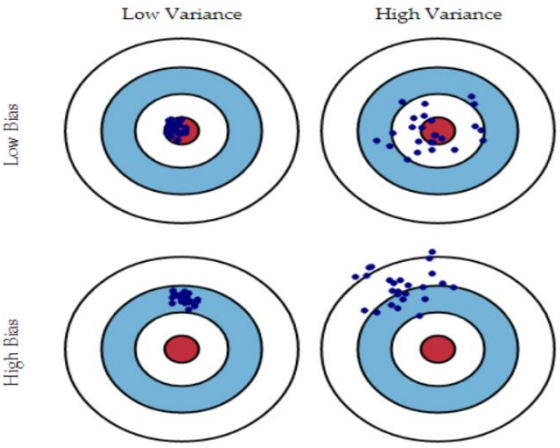
$$Bias\big(h(X), f(X)\big) = E[h(X)] - f(X)$$

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

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

Considering their combination 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 not varying so much (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.

Q: Show the derivation of the Generalization Error (GER) and explain what it leads to. How is this concept related to underfitting and overfitting?

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:

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

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**:

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

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)**:

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

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

Note that the last step can be performed because of the linearity of the expected value operator.

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

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:

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

Since expectation is linear, we can distribute it:

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

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

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 $Var(e^{(i)}) = \mathbb{E}\left[\left(e^{(i)}\right)^2\right] - 0 \Rightarrow \mathbb{E}\left[\left(e^{(i)}\right)^2\right] = Var(e^{(i)}) = \sigma_e^2$. 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:

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

$$= 2\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right) E_D[h^{(D)}(x^{(i)}) - \bar{h}(x^{(i)})] =$$

$$= 2\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)\left(E_D[h^{(D)}(x^{(i)})] - E_D[\bar{h}(x^{(i)})]\right) =$$

$$= 2\left(\bar{h}(x^{(i)}) - f(x^{(i)})\right)\left(\bar{h}(x^{(i)}) - \bar{h}(x^{(i)})\right) = 0$$

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).

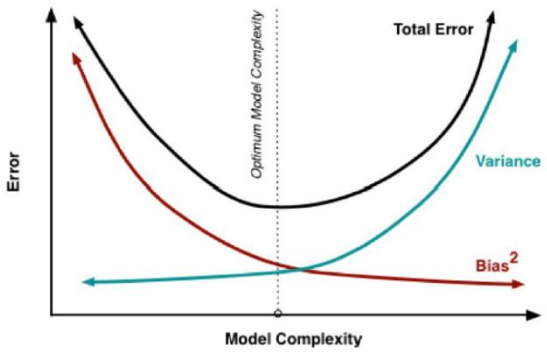
- Variance represents how much a single hypothesis can differ from the best estimate. **High variance** means that the same model, trained with different datasets, generates very different hypotheses. The hypotheses generated are very complex, are prone to overfit and are unable to generalize.

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.

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.

Q: What is regularization and how it works?

Regularization is a technique used to **prevent overfitting** by adding a **penalty term** to the cost function, which discourages the model from assigning excessively large values to its parameters. This helps in **simplifying the model** and improving generalization to unseen data.

Intuition: The intuition behind regularization is that overfitting occurs when the model becomes too complex, often due to large parameter values for certain features. These large parameters can cause the model to fit the training data too closely, aligning too much with specific patterns in the data rather than capturing the underlying relationships.

To address this, we can add a large penalty to the parameters, which forces the optimization process to shrink them to smaller values. This results in a smoother model that is less sensitive to the specific details of the training data.

→ Therefore, in Regularization, instead of removing features, we retain all of them but we reduce their magnitude which leads to a **simpler hypothesis**, less prone to overfitting and that more likely will generalize well to new, unseen data.

The **general formulation** of a regularized cost function is:

$$J(\theta) + \lambda\Omega(\theta)$$

where:

- $J(\theta)$ is the original loss function (e.g., MSE for linear regression, BCE for logistic regression).
- λ is the **regularization hyperparameter**, which controls the penalty's strength.
- $\Omega(\theta)$ is the **regularization term**, which defines how the penalty is applied to the model parameters. Two commonly used regularization terms are the **l2-norm** and the **l1-norm**.

Note: The regularization is applied to all parameters except the bias parameter (i.e. θ_0).

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.

Q: Show how is obtained the explicit form formulation of the partial derivative of the MSE cost function w.r.t. a parameter θ_j when l2-norm regularization is applied.

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**, in this case an l2-norm:

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

As we can see in the penalty term the 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.

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

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

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:

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

Note: This formulation applies also to logistic regression (since they have the same explicit form for partial derivative), but of course the hypothesis is different.

Q: Difference between l2-norm and l1-norm in regularization.

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 penalizes large parameter values encouraging their values to decay **toward zero** (but not exactly zero). This leads to a smoother and more stable model.

The $\boldsymbol{l^1}$ **norm** penalizes the sum of the absolute values of the model's parameters:

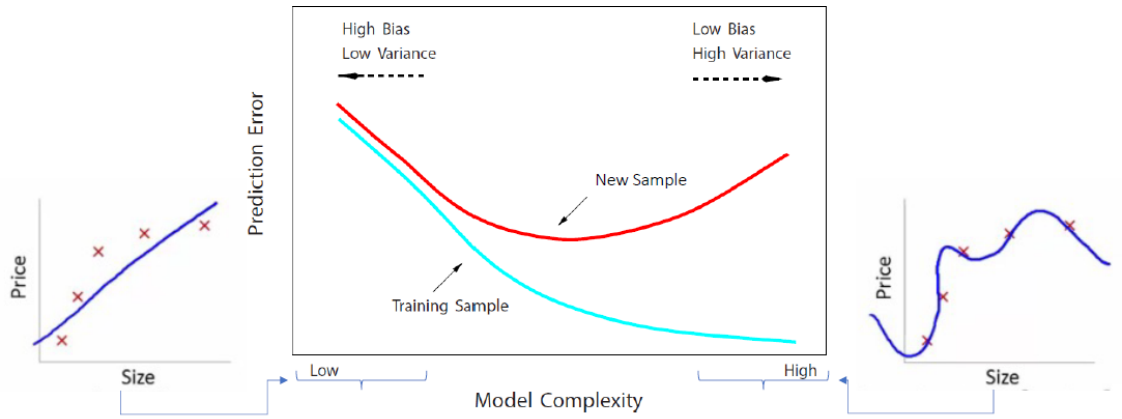
$$l^1 = \|\theta\|_1 = \sum_{j=1}^{n}|\theta_j|$$

This regularization encourages some parameter values to become **exactly zero**, effectively performing a **feature selection**.

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.

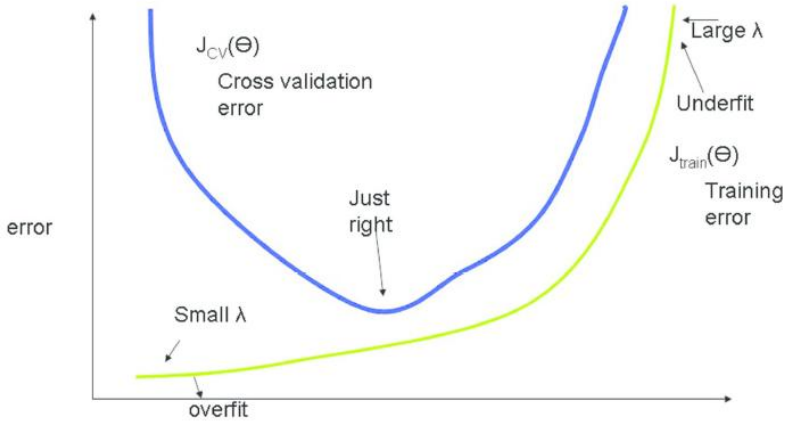
Q: How does the regularization hyperparameter λ influence the Bias-Variance tradeoff?

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.



Regularization directly impacts the **Bias-Variance tradeoff** by controlling the complexity of the model through the **regularization hyperparameter** λ: 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**



Q: Why is data preprocessing important and what are its main steps?

Real-world data are often **incomplete or inaccurate**, leading to poor model performance. This is captured by the **Garbage In, Garbage Out (GIGO) principle**—if the input data is poor, the model's output will also be poor, no matter how good the model is.

Therefore, it is essential to perform an accurate preprocessing of the data. This phase generally takes a long time and consists of the following steps:

- **Data cleaning** (removing outliers, noise, duplicates).

- **Data transformation** (normalization, discretization, aggregation).
- **Feature selection** (choosing the most relevant features via domain expertise, filters, wrappers, or dimensionality reduction like PCA).
- If required, create new features.

Q: What are quartiles, and how are they used to detect outliers in a dataset?

Quartiles divide a dataset into four equal parts (after the data has been sorted in ascending order):

- $\boldsymbol{Q_2}$ **(Second Quartile)** is equal to the **median** of the data set.
- $\boldsymbol{Q_1}$ **(First Quartile)** is the median of the values that are below Q_2.
- $\boldsymbol{Q_3}$ **(Third Quartile)** is the median of the values that are above Q_2.

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

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

The IQR measures the spread of the middle 50% of the data.

Outliers can be detected using 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**."

Q: Why is it incorrect to choose a model based on its performance on the test set?

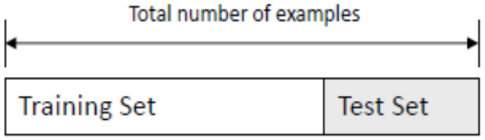
Choosing a model based on its performance on the **test set** is incorrect because the test set is meant to provide an **unbiased estimate** of the model's generalization ability. If we use the test set to select the best model, we risk "**overfitting" to the test data**, meaning the model may perform well on this specific test set but fail to generalize to truly unseen data.

Instead, the correct approach is to adopt a **validation set**. Specifically, we should split the dataset into three parts:

- **Training set:** Used to train the model.
- **Validation set:** Used to compare different models, tune hyperparameters, and select the best-performing model.
- **Test set:** Used **only** for final evaluation, after the model is selected.

Q: Explain in detail what are Hold-out, K-Fold Cross Validation and Random Subsampling.

The **Hold-out** method is commonly used to split a dataset into **training** and **test sets**. Typically, the data are 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 issue, **Stratified Sampling (Stratification)** is used, which performs the split ensuring that the class proportions in both the training and test sets match those in the original dataset.

A **validation set** can also be introduced by splitting the training data further (e.g., 70% training, 30% validation), but this reduces the number of samples available for training.

Since 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) other techniques such as K-Fold Cross Validation are preferred.

K-Fold Cross Validation is a more robust evaluation technique that reduces the dependency on a single train-test split. The dataset is divided into **K equally sized subsets** called **folds**, and the model is trained as follows:

1. Each time, one fold is used as the **validation set**, while the remaining K-1 folds form the **training set**.
2. The process is repeated K times, with each fold serving as the validation set once.
3. The final performance metric is obtained by averaging the results across all K iterations.

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

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

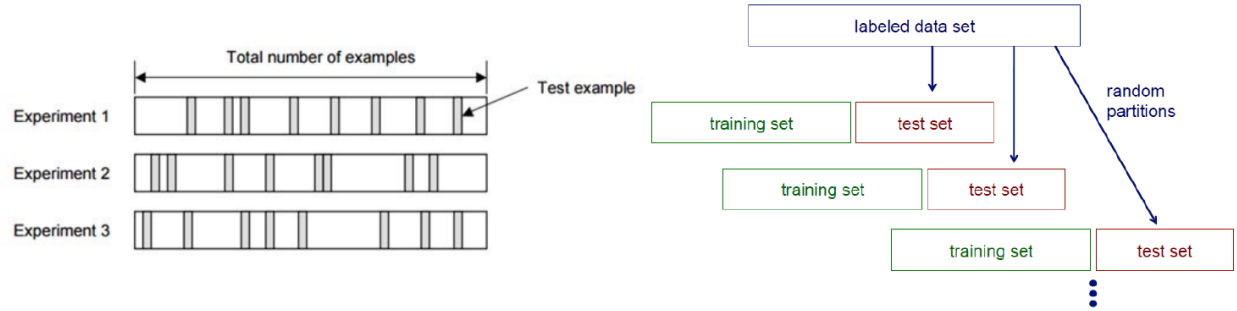
The advantage in performing K-Fold CV is that every data point is used for both **training and validation**, which reduces variance compared to a single train-test split.

10-folds or 5-folds cross validation is usually used.

Stratified K-Fold Cross Validation: To ensure that each fold maintains the original class distribution it is a good idea to apply stratification also to K-Fold Cross Validation.

Random Subsampling is another method that helps reduce the dependency on a single holdout split. Instead of performing one random split (as in the Holdout), 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 samples **without replacement** (therefore they can be picked more than once) as training samples and the remaining are used for validation.
- For each split, the classifier is retrained from scratch using the training samples, 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.

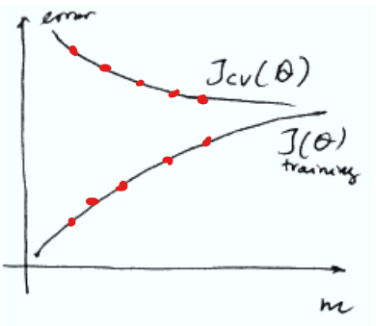


By repeating this process, random subsampling provides a more robust estimate of model performance, as it reduces the variance caused by any particular data split.

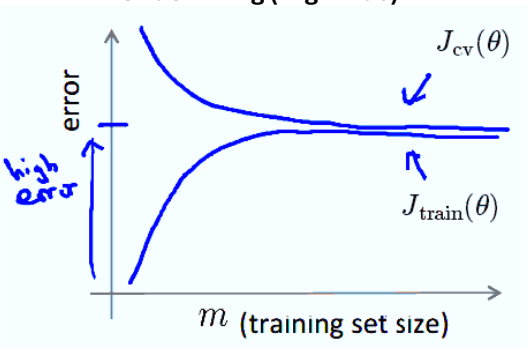
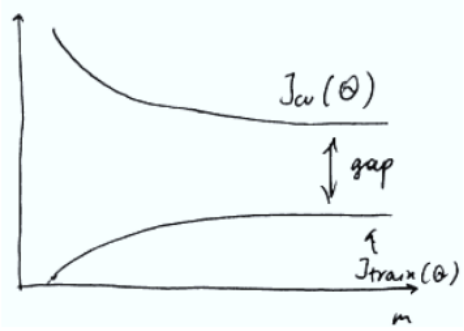
! ***K-fold cross validation vs Random subsampling***: In **K-fold cross-validation**, each data point is guaranteed to be in the validation set exactly once, ensuring complete coverage. In **Random subsampling**, we basically perform K experiments (K hold-outs) where splits are done randomly each time,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

Q: How can learning curves help diagnose and improve the performance of a machine learning model?

Learning curves are a valuable diagnostic tool for understanding whether a model is suffering from underfitting (high bias) or overfitting (high variance). They plot the training error $J_{train}(\theta)$ and cross-validation error $J_{cv}(\theta)$ as a function of training set size m:



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

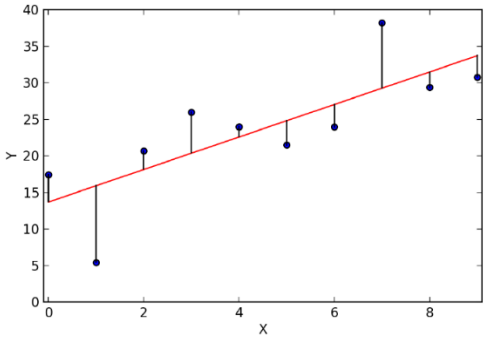
By analyzing learning curves, we can make informed decisions on how to adjust our model to achieve better generalization and improve overall performance.

→ **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.

Q: What are the key evaluation metrics used to assess machine learning models for regression and classification tasks?

The choice of evaluation metrics depends on the type of machine learning task considered.

- **Evaluation metrics for regression**: For regression the **residuals** are considered, which represent the difference between predicted values and actual values.



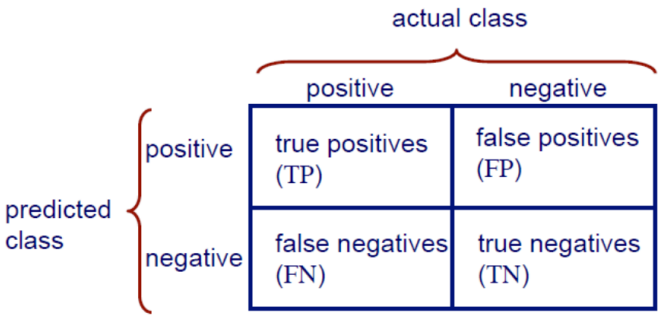
From these, you can adopt the following regression metrics:

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

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

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

- **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.

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

Q: What is the ROC curve, and how does AUC help evaluate the performance of a classification model?

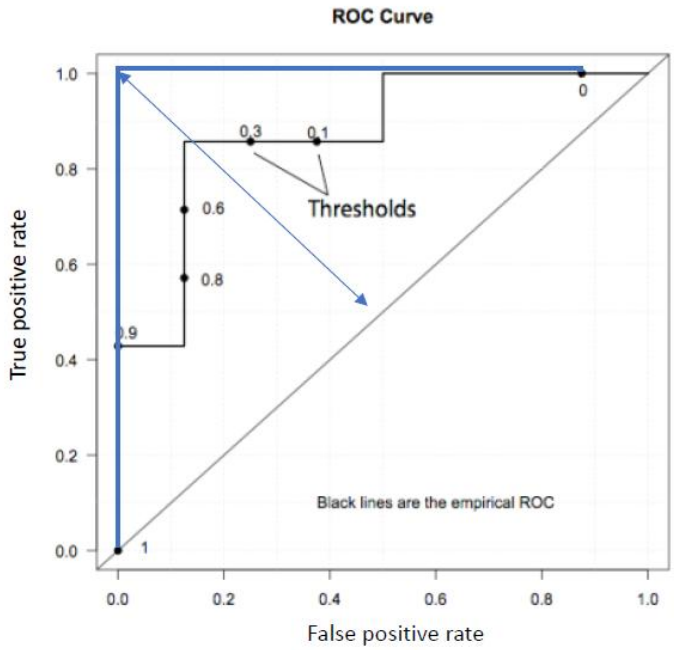
A **Receiver Operating Characteristic curve**, or **ROC curve**, is a graphical representation of a binary classifier's performance varying threshold values. The ROC curve plots the True Positive Rate (TPR) on y-axis against the False Positive Rate (FPR) on x-axis at each threshold.

- The best possible prediction model would yield a point in the upper left corner or coordinate **(0,1)** of the ROC space. This point is known as "**perfect classification**."

- The plot also includes a **diagonal line** from (0,0) to (1,1), 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).

To construct the ROC curve we proceed as follows: We start by considering the data consisting of actual class labels (y_true) and predicted probabilities (y_pred) of the binary classifier. We consider different classification thresholds (from 0 to 1) where each threshold produces a different confusion matrix and as consequence different unique TPR and FPR values. By connecting each unique couple of TPR and FPR so obtained, we create the ROC curve that shows the model performance across all thresholds.

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 representing the area under the ROC curve that summarizes the classifier's performance across all possible thresholds. AUC **ranges from 0 to 1**:

- **AUC = 1:** The model perfectly distinguishes between classes.
- **AUC = 0.5:** The model performs no better than random guessing.
- **AUC < 0.5:** The model is worse than random guessing

→ **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.

Q: What is the paired t-test and for what is used for?

The **Paired t-test** is a statistical method used to compare the performance (on a given evaluation metric) of two machine learning models on the same dataset and determine whether the observed difference is statistically significant or due to random variation.

To determine it, the method defines two competing hypotheses:

- **Null Hypothesis**: the performance differences between the two models are due to chance → all observable differences are explained by random variations.

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

To show how it works, let's consider two binary classifiers a and b. The pairs of observations consist of accuracy values computed by the two models:

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

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

First, 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.

Then we:

1. calculate the sample **mean**:

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

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

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

3. determine the corresponding $\boldsymbol{p}$***−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:

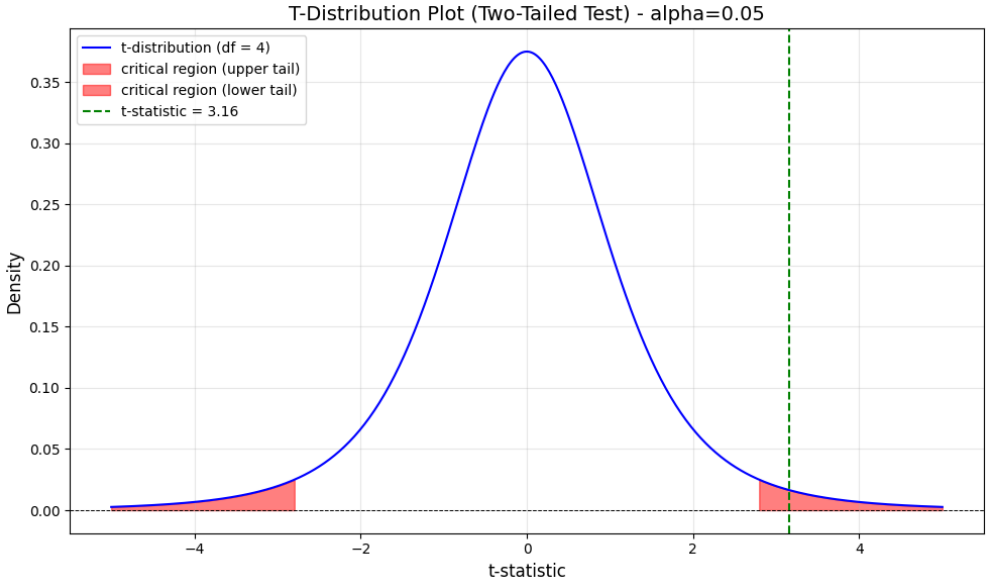
1. **Two-Tailed Test:** $\boldsymbol{p = 2 \cdot Pr(T > |t|)}$ → It answer to the question: "Is the accuracy of the two models different?". This test doesn't specify the direction (it doesn't tell us which of the two models is better).

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?"

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

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.



Q: What is the R^2 metric and for what is used for?

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.

Q: For what Grid Search is used for and what are its advantages and limitations?

Grid Search is a widely used method for hyperparameter tuning.

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

Grid Search involves defining a grid of hyperparameters and exhaustively searching through all possible combinations within that grid the best combination.

How Grid Search Works:

1. Define a grid of values for each hyperparameter.
2. Generate all possible combinations of these values.
3. Train and evaluate the model for each combination using a chosen evaluation metric (e.g., accuracy, F1-score, ...).
4. The combination yielding the best performance is selected as the optimal set of hyperparameters.

Advantages of Grid Search:

- **Systematic & Exhaustive** – Ensures that all combinations are tested.
- **Cross-Validation** – Often combined with **k-fold cross-validation** to reduce overfitting and improve robustness.

Limitations of Grid Search:

- **Computationally Expensive** – As the number of hyperparameters increases, the number of combinations grows exponentially, making it impractical for large models.
- **Limited to Predefined Values** – It can only test values that are explicitly specified, potentially missing better-performing values outside the grid. For large search spaces, **Bayesian Optimization** may be a more efficient alternative.

Q. What is a Neural Network? Write its matrix formulation and explain how forward propagation works.

A **Neural Network (NN)** is a machine learning model inspired by the structure of the human brain. It consists of interconnected processing units called **neurons**, organized into **layers**. The goal of a neural network is to learn complex patterns in data by adjusting its parameters (weights and biases) through training.

We can distinguish between two types of layers in a neural network:

- **Hidden Layers**: Intermediate layers where computations take place.
- **Output Layer**: Produces the final result of the network's computation, either a prediction or classification, as the result of the hypothesis function.

Each **neuron** (also known as **unit**) performs the following computation:

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

So, it performs a weighted sum $\theta^T x$ and applies to it a **nonlinear activation function** g. For instance, by adopting a **sigmoid function** we will have:

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

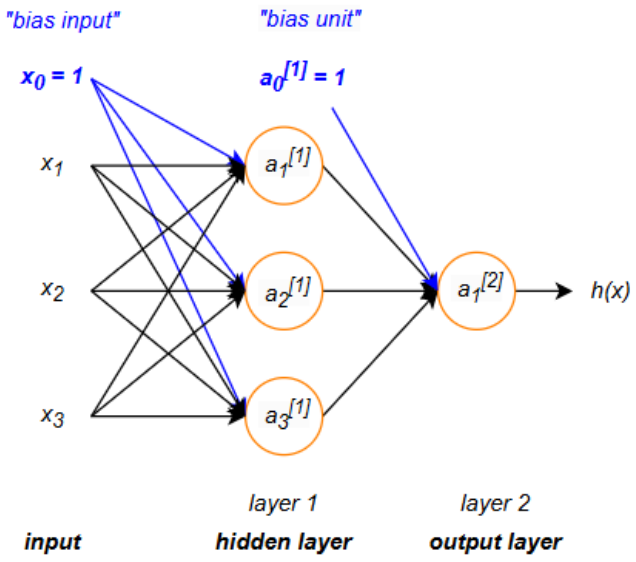
A neural network is essentially a collection of these neurons working together. Each neuron receives inputs from previous layer, computes a **weighted sum**, applies a **non-linear activation function**, and passes the result to the next layer. This entire process is called **forward propagation**.

Let's denote with:

- $x_j =$ the j-th input feature
- $z_j^{[l]}$= the weighted sum computed in a given unit j in layer l
- $a_j^{[l]}$= the "activation" of unit j in layer l
- $W^{[l]}$ = the **weight matrix** controlling function mapping from layer $l-1$ to layer l.
- $b^{[l]}$ = the **bias vector**, i.e. the vector containing all the bias parameters for layer l

Note: Here we separated the weights $W^{[l]}$ and the biases $b^{[l]}$ into distinct components so that regularization can be applied directly only to the weights $W^{[l]}$ (and not to the biases $b^{[l]}$) without needing to iterate through the parameter matrix to exclude bias terms (for instance via a slicing).

Suppose considering the following neural network:



In this case we will have:

$$W^{[1]} = \begin{bmatrix} W_{11}^{[1]} & W_{12}^{[1]} & W_{13}^{[1]} \\ W_{21}^{[1]} & W_{22}^{[1]} & W_{23}^{[1]} \\ W_{31}^{[1]} & W_{32}^{[1]} & W_{33}^{[1]} \end{bmatrix}$$

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

$$W^{[2]} = \begin{bmatrix} W_{11}^{[2]} & W_{12}^{[2]} & W_{13}^{[2]} \end{bmatrix}$$

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

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

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

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

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

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

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

The dimensions of each weight matrix (which depends on the number of units in adjacent layers) and each bias vector are determined as follows:

➔ **If we denote with s_{l-1} the number of units in layer $l-1$ and with s_l the number of units in layer l then $W^{[l]}$ will be of dimension $s_l \times s_{l-1}$, while $b^{[l]}$ will be of dimension $s_l \times 1$.**

In vectorized form we can rewrite the forward propagation steps seen above as:

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$

$$a^{[1]} = g(z^{[1]})$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$h(x) = a^{[2]} = g(z^{[2]})$$

where we indicated x with $a^{[0]}$.

Therefore, the **vectorized forward propagation** equations are:

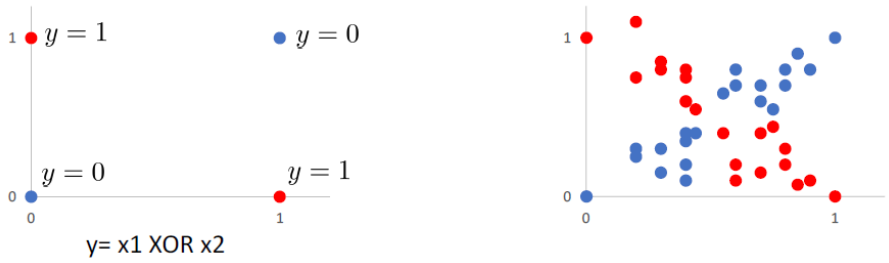
$$\boldsymbol{z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}}$$

$$\boldsymbol{a^{[l]} = g(z^{[l]})}$$

Q: How can a neural network be designed to handle a XOR classification problem?

First, I want to underline that a standard linear classifier such as a logistic regression cannot solve the XOR classification problem because the XOR function is not linearly separable. The decision boundary required to separate positive and negative examples in XOR is nonlinear, meaning a single straight line cannot partition the data correctly.

To solve XOR using a neural network, we can break it down into simpler logical operations: AND, OR, and NOT. Each of these functions can be implemented using a single neuron with appropriately chosen weights and biases. 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:

$$h(x) = g(-30 + 20x_1 + 20x_2)$$

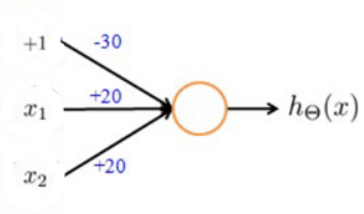
where g is the sigmoid activation function.

Let's see how this network behaves for all possible combinations of x1 and x2:

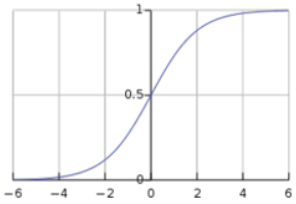
- If $x_1 = 0$ and $x_2 = 0, h(x) = g(-30)$, which is very close to 0.
- If $x_1 = 0$ and $x_2 = 1, h(x) = g(-10)$, also close to 0.
- If $x_1 = 1$ and $x_2 = 0, h(x) = g(-10)$, still close to 0.
- If $x_1 = 1$ and $x_2 = 1, h(x) = g(10)$, which is close to 1.

This output matches the AND function, where 1 appears only when both inputs are 1. So, basically like that we approximated an **AND**.

$x_1, x_2 \in \{0, 1\}$
$y = x_1 \text{AND} x_2$



$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$



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:

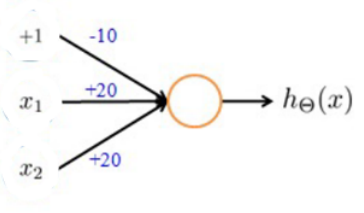
$$h(x) = g(-10 + 20x_1 + 20x_2)$$

This setup will result in:

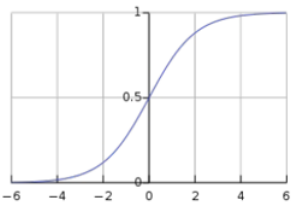
- If $x_1 = 0$ and $x_2 = 0, h(x) = g(-10)$, which is close to 0.
- If $x_1 = 0$ and $x_2 = 1, h(x) = g(10)$, also close to 1.
- If $x_1 = 1$ and $x_2 = 0, h(x) = g(10)$, still close to 1.
- If $x_1 = 1$ and $x_2 = 1, h(x) = g(30)$, which is very close to 1.

This matches the OR function, where 1 is output when at least one input is 1. So, basically like that we approximate an **OR**.

$x_1, x_2 \in \{0, 1\}$
$y = x_1 \text{OR} x_2$



$h_\Theta(x) = g(-10 + 20x_1 + 20x_2)$



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:

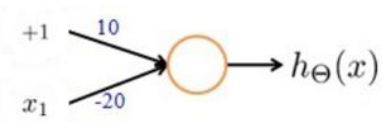
$$h(x) = g(+10 - 20x_1)$$

This setup yields:

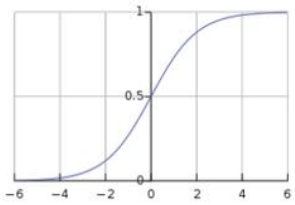
- If $x_1 = 0$, $h(x) = g(10)$, which is close to 1.
- If $x_1 = 1$, $h(x) = g(-10)$, which is close to 0.

Thus, this effectively performs the **NOT** operation.

$x_1 \in \{0, 1\}$
$y = \text{NOT } x_1$



$h_\Theta(x) = g(10 - 20x_1)$



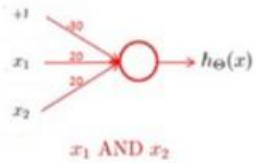
x_1	$h_\Theta(x)$
0	$g(+10) \approx 1$
1	$g(-10) \approx 0$

XOR function

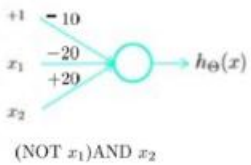
With these building blocks, we can now construct the XOR function by combining the AND, OR, and NOT networks:

$$x_1 \; XOR \; x_2 = \left(x_1 \; AND \; (NOT \; x_2)\right) \; OR \; \left(x_2 \; AND \; (NOT \; x_1)\right)$$

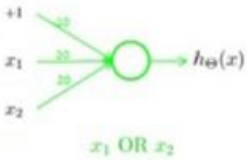
$x_1 \in \{0, 1\}$
$y = x_1 \text{XOR } x_2$



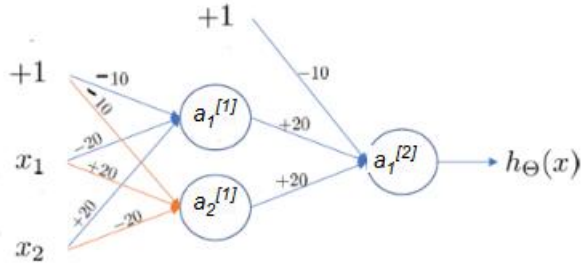
x_1 AND x_2



(NOT x_1)AND x_2



x_1 OR 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: $a_1^{[1]} \rightarrow$ "$x_2\ AND\ (NOT\ x_1)$" and $a_2^{[1]} \rightarrow$ "$x_1\ AND\ (NOT\ x_2)$".
- **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.

Q: How does a neural network handle multi-class classification?

Multiclass classification is a task where we classify data into one of several categories. To handle these additional classes, the neural network needs to include **multiple output units—one for each class**. So, instead of producing a single output, the network generates a vector where each value represents the probability that the input belongs to a specific category.

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 $\left(x^{(i)}, y^{(i)}\right)$, 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**.

For instance, supposing to have a multi-class classification problem consisting in four classes, i.e. pedestrian, car, motorcycle and truck, the label vector will follow a representation such that:

- The label vector of an input appertaining to the class pedestrian would be $y^{(i)} = [1, 0, 0, 0]^T$
- The label vector of an input appertaining to the class car would be $y^{(i)} = [0, 1, 0, 0]^T$
- The label vector of an input appertaining to the class motorcycle would be $y^{(i)} = [0, 0, 1, 0]^T$
- The label vector of an input appertaining to the class truck would be $y^{(i)} = [0, 0, 0, 1]^T$

Q: Write the Cost Function formulations for Regression, Binary Classification and Multi-class Classification in a Neural Network.

Cost Function for Regression: It is an extension of the Mean Squared Error (MSE) seen for linear regression adapted for neural network.

$$J(W, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(h\left(x^{(i)}\right) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{v=1}^{s_{l-1}}\sum_{j=1}^{s_l}\left(W_{jv}^{[l]}\right)^2$$

where $h\left(x^{(i)}\right) = Wx^{(i)} + b$ and the last term is the regularization term which sums all parameters in all layers of the network.

- L = total number of layers in the network
- s_{l-1} = number of units in layer $l-1$, s_l = number of units in layer l (not counting bias units)

Cost Function for Binary Classification: It is an extension of the Binary Cross-Entropy (BCE) seen for logistic regression adapted for neural network.

$$J(W,b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h\left(x^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-h\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{v=1}^{s_{l-1}}\sum_{j=1}^{s_l}\left(W_{jv}^{[l]}\right)^2$$

where $h\left(x^{(i)}\right) = g\left(Wx^{(i)} + b\right)$ and g is the sigmoid function.

Cost Function for Multiclass Classification: For **multi-class** classification, the cost function is the Cross Entropy (CE). Instead of a single output unit, we now handle K output units. If we indicate with:

$$h(x) \in \mathbb{R}^K \quad with\ \left(h(x)\right)_k = k^{th} output$$

a hypothesis that results in the k^{th} output, then our loss will be:

$$J(W,b) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log\left(\left(h\left(x^{(i)}\right)\right)_k\right) + \left(1-y_k^{(i)}\right)\log\left(1-\left(h\left(x^{(i)}\right)\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{v=1}^{s_{l-1}}\sum_{j=1}^{s_l}\left(W_{jv}^{[l]}\right)^2$$

Q: What is backpropagation in neural networks, and why is it necessary for training?

To find the optimal network parameters values we need to minimize the cost function and to achieve this, we can apply an optimization algorithm such as Gradient Descent.

However, to apply Gradient Descent, we must first compute the gradient of the cost function with respect to the parameters at each layer (i.e. $\frac{\partial J}{\partial W^{[l]}}$ and $\frac{\partial J}{\partial b^{[l]}}$). This is not straightforward because, in a multi-layer neural network, each parameter's contribution to the total error depends not only on its own value but also on the errors propagated from later layers.

Backpropagation addresses this challenge by passing error information **backward** through the network, layer by layer, starting from the last layer to the input allowing for the efficient computation of these gradients.

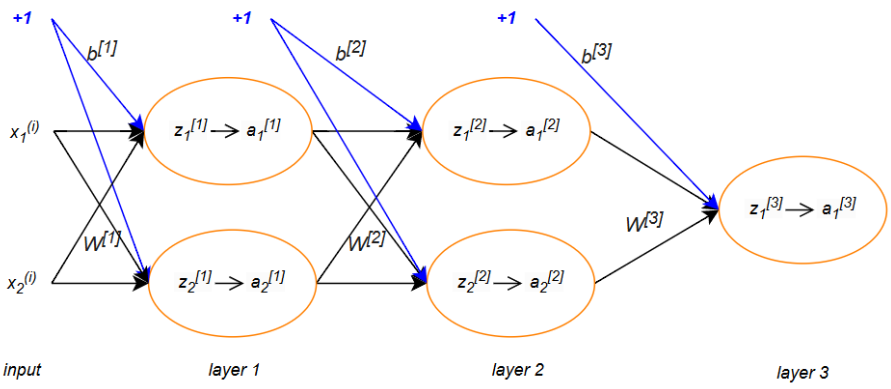
Once the gradients are determined, we can update the network's parameters (features parameters and bias parameters) of each layer using Gradient Descent update rule:

$$W^{[l]} := W^{[l]} - \alpha\frac{\partial J}{\partial W^{[l]}}$$

$$b^{[l]} := b^{[l]} - \alpha\frac{\partial J}{\partial b^{[l]}}$$

Below I show how backpropagation is performed on a simple neural network in case of a regression task.

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 (MSE) in vectorial form can be expressed as (supposing to consider a single training sample):

$$J(W,b) = \frac{1}{2}\left(a^{[3]} - y\right)^2$$

where $a^{[3]} = h(x)$ is the output of the network and y is the true value.

We first start by doing the **forward propagation**. So given the training example (x, y) (recall that we indicate $x = a^{[0]}$) we push it through the network as:

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$

$$a^{[1]} = g\left(z^{[1]}\right)$$

$$z^{[2]} = W^{(2)}a^{[1]} + b^{[2]}$$

$$a^{[2]} = g\left(z^{[2]}\right)$$

$$a^{[3]} = z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$$

! We recall that we don't apply sigmoid to the last layer for a regression task.

Next, we perform the **back propagation**, i.e. we walk through the network in reverse order (from the last layer to the input) computing the gradients of the cost function w.r.t. to the parameters at each layer. We can make use of the chain rule to help us in solve them. To build the intuition we derive them using scalar notation (even if it is not correct since we are operating with vectors and matrices).

We start by computing the gradient of the cost function with respect to $W^{[3]}$ and $b^{[3]}$.

For $\frac{\partial J}{\partial W^{[3]}}$ we have:

$$\frac{\partial J}{\partial W^{[3]}} = \frac{\partial J}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial W^{[3]}}$$

we can compute each of these partial derivatives separately:

$$\frac{\partial J}{\partial a^{[3]}} = \frac{\partial \frac{1}{2}\left(a^{[3]} - y\right)^2}{\partial a^{[3]}} = \frac{1}{2}2\left(a^{[3]} - y\right) = \left(a^{[3]} - y\right)$$

since $a^{[3]} = z^{[3]}$ we will have $\frac{\partial a^{[3]}}{\partial z^{[3]}} = \frac{\partial z^{[3]}}{\partial z^{[3]}} = 1$

$$\frac{\partial z^{[3]}}{\partial W^{[3]}} = \frac{\partial\left(W^{[3]}a^{[2]} + b^{[3]}\right)}{\partial W^{[3]}} = a^{[2]}$$

Thus, the gradient with respect to $W^{[3]}$ is:

$$\frac{\partial J}{\partial W^{[3]}} = \frac{\partial J}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial W^{[3]}} = \left(a^{[3]} - y\right)a^{[2]}$$

Let's indicate $\left(a^{[3]} - y\right)$ with $\boldsymbol{\delta^{[3]}}$.

For $\frac{\partial J}{\partial b^{[3]}}$ we have:

$$\frac{\partial J}{\partial b^{[3]}} = \boldsymbol{\delta^{[3]}} \frac{\partial z^{[3]}}{\partial b^{[3]}} = \delta^{[3]} \frac{\partial \left(W^{[3]} a^{[2]} + b^{[3]}\right)}{\partial b^{[3]}} = \delta^{[3]}$$

Now, let's compute the gradients of the cost function with respect to $W^{[2]}$ and $b^{[2]}$.

For $\frac{\partial J}{\partial W^{[2]}}$ we have:

$$\frac{\partial J}{\partial W^{[2]}} = \delta^{[3]} \frac{\partial z^{[3]}}{\partial W^{[2]}} = \delta^{[3]} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial W^{[2]}}$$

Let's compute the other terms not already computed:

$$\frac{\partial z^{[3]}}{\partial a^{[2]}} = \frac{\partial W^{[3]} a^{[2]} + b^{[3]}}{\partial a^{[3]}} = W^{[3]}$$
$$\frac{\partial a^{[2]}}{\partial z^{[2]}} = \frac{\partial g\left(z^{[2]}\right)}{\partial z^{[2]}} = g'\left(z^{[2]}\right)$$
$$\frac{\partial z^{[2]}}{\partial W^{[2]}} = \frac{\partial W^{[2]} a^{[1]} + b^{[2]}}{\partial W^{[2]}} = a^{[1]}$$

Substituting these into the equation gives:

$$\frac{\partial J}{\partial W^{[2]}} = \delta^{[3]} W^{[3]} g'\left(z^{[2]}\right) a^{[1]}$$

Indicating $\delta^{[3]} W^{[3]} g'\left(z^{[2]}\right)$ with $\delta^{[2]}$, the gradient with respect to $\Theta^{[2]}$ becomes:

$$\frac{\partial J}{\partial W^{[2]}} = \delta^{[2]} a^{[1]}$$

For $\frac{\partial J}{\partial b^{[2]}}$ we have:

$$\frac{\partial J}{\partial b^{[2]}} = \delta^{[2]} \frac{\partial z^{[2]}}{\partial b^{[2]}} = \delta^{[2]} \frac{\partial \left(W^{[2]} a^{[1]} + b^{[2]}\right)}{\partial b^{[2]}} = \delta^{[2]}$$

Finally, we compute the gradient with respect to $W^{[1]}$ and $b^{[1]}$.

For $\frac{\partial J}{\partial W^{[1]}}$ we have:

$$\frac{\partial J}{\partial W^{[1]}} = \delta^{[2]} \frac{\partial z^{[2]}}{\partial W^{[1]}} = \delta^{[2]} \frac{\partial z^{[2]}}{\partial a^{[1]}} \frac{\partial a^{[1]}}{\partial z^{[1]}} \frac{\partial z^{[1]}}{\partial W^{[1]}}$$

Let's compute the remaining terms:

$$\frac{\partial z^{[2]}}{\partial a^{[1]}} = \frac{\partial W^{[2]} a^{[1]} + b^{[2]}}{\partial a^{[1]}} = W^{[2]}$$
$$\frac{\partial a^{[1]}}{\partial z^{[1]}} = \frac{\partial g\left(z^{[1]}\right)}{\partial z^{[1]}} = g'\left(z^{[1]}\right)$$
$$\frac{\partial z^{[1]}}{\partial W^{[1]}} = \frac{\partial W^{[1]} a^{[0]} + b^{[1]}}{\partial W^{[1]}} = a^{[0]}$$

Substituting these gives:

$$\frac{\partial J}{\partial W^{[1]}} = \delta^{[2]} W^{[2]} g'\left(z^{[1]}\right) a^{[0]}$$

Indicating $\delta^{[2]} W^{[2]} g'\left(z^{[1]}\right)$ with $\delta^{[1]}$ the gradient with respect to $\theta^{[1]}$ becomes:

$$\frac{\partial J}{\partial W^{[1]}} = \delta^{[1]} a^{[0]}$$

For $\frac{\partial J}{\partial b^{[1]}}$ we have:

$$\frac{\partial J}{\partial b^{[1]}} = \delta^{[1]} \frac{\partial z^{[1]}}{\partial b^{[1]}} = \delta^{[1]} \frac{\partial \left(W^{[1]} a^{[0]} + b^{[1]}\right)}{\partial b^{[1]}} = \delta^{[1]}$$

Reporting the formulations we got using vectorial notation we will have:

$$\frac{\partial J}{\partial W^{[3]}} = \delta^{[3]} \left(a^{[2]}\right)^T \qquad \frac{\partial J}{\partial b^{[3]}} = \delta^{[3]}$$

$$\frac{\partial J}{\partial W^{[2]}} = \delta^{[2]} \left(a^{[1]}\right)^T \qquad \frac{\partial J}{\partial b^{[2]}} = \delta^{[2]}$$

$$\frac{\partial J}{\partial W^{[1]}} = \delta^{[1]} \left(a^{[0]}\right)^T \qquad \frac{\partial J}{\partial b^{[1]}} = \delta^{[1]}$$

where:

$$\delta^{[3]} = \left(a^{[3]} - y\right)$$

$$\delta^{[2]} = \left(\left(W^{[3]}\right)^T \delta^{[3]}\right) \odot g'\left(z^{[2]}\right)$$

$$\delta^{[1]} = \left(\left(W^{[2]}\right)^T \delta^{[2]}\right) \odot g'\left(z^{[1]}\right)$$

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.

$\boldsymbol{\delta^{(l)}}$ term represent the "**error**" of neurons in layer l.

From this we can see that to compute the gradient of the cost function with respect to the parameters of a given layer we can simply perform a dot product of its error δ with the transpose of activation values vector from the previous layer:

$$\frac{\partial J}{\partial W^{[l]}} = \delta^{[l]} \left(a^{[l-1]}\right)^T \; for \; l = 1, \ldots, L$$

where the error of the neurons in a hidden layer l can be computed starting from the error derived from the previous step (layer $l + 1$):

$$\delta^{[l]} = \left(\left(W^{[l+1]}\right)^T \delta^{[l+1]}\right) \odot g'\left(z^{[l]}\right)$$

In summary, the gradient computations follow these general rules:

$$\boldsymbol{\frac{\partial J}{\partial \theta^{[l]}} = \delta^{[l]} \left(a^{[l-1]}\right)^T \; for \; l = 1, \ldots, L}$$

$$\frac{\partial J}{\partial b^{[l]}} = \delta^{[l]} \; for \; l = 1, \dots, L$$

where we can compute the error δ as follows:

- For the **output layer** L, the error $\delta^{[L]}$ in case of regression is simply the difference between the predicted value $a^{[L]}$ and the true value y:

$$\delta^{[L]} = \left(a^{[L]} - y\right)$$

- For any **hidden layer** the error contribution δ is given by:

$$\delta^{[l]} = \left(\left(W^{[l+1]}\right)^T \delta^{[l+1]}\right) \odot g'\left(z^{[l]}\right)$$

  where the g' derivative term depends on the type of activation function considered. For instance, if the activation function is sigmoid $a^{[l]} = g\left(z^{[l]}\right) = \sigma\left(z^{[l]}\right)$ than its derivative is:

$$g'\left(z^{[l]}\right) = a^{[l]} \odot \left(1 - a^{[l]}\right)$$

Backpropagation Algorithm

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:

$$\delta^{[L]} = a^{[L]} - y$$

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:

$$\delta^{[l]} = \left(\left(W^{[l+1]}\right)^T \delta^{[l+1]}\right) \odot \left(a^{[l]} \odot \left(1 - a^{[l]}\right)\right)$$

4. **Gradient Calculation**:
   - Once the errors are computed for all layers, we calculate the gradient of the cost function with respect to the parameters $W^{[l]}$ and $b^{[l]}$ for each layer l as follows:

$$\frac{\partial J}{\partial W^{[l]}} = \delta^{[l]}\left(a^{[l-1]}\right)^T + \lambda W^{[l]}$$

$$\frac{\partial J}{\partial b^{[l]}} = \delta^{[l]}$$

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 $\frac{\partial}{\partial W^{[l]}} J(W, b)$ and $\frac{\partial}{\partial b^{[l]}} J(W, b)$.

After that we can use these gradients that we have found with backprop to update the parameters of each layer using **gradient descent**:

$$W^{[l]} := W^{[l]} - \alpha \frac{\partial J(\theta, b)}{\partial W^{[l]}}$$

$$b^{[l]} := b^{[l]} - \alpha \frac{\partial J(\theta, b)}{\partial b^{[l]}}$$

Backprop Pseudo-algorithm (for BGD)

Training set $\{(x^{(1)}, y^{(1)}), \dots, x^{(m)}, y^{(m)}\}$

Set $\Delta_W^{[l]} = 0$ (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 $l = 1, \dots, L$

Using $y^{(i)}$, compute $\delta^{[L]} = a^{[L]} - y^{(i)}$ (this in case of regression)

Compute $\delta^{[L-1]}, \delta^{[L-2]}, \dots, \delta^{[1]}$

$\Delta_W^{[l]} = \Delta_W^{[l]} + \delta^{[l]} \left(a^{[l-1]}\right)^T$

$\Delta_b^{[l]} = \Delta_b^{[l]} + \delta^{[l]}$

At the end:

$$\frac{\partial J}{\partial W^{[l]}} = \frac{1}{m}\left(\Delta_W^{[l]} + \lambda W^{[l]}\right)$$

$$\frac{\partial J}{\partial b^{[l]}} = \frac{\Delta_b^{[l]}}{m}$$

Q: Derive the explicit form error formulation of the output layer of a Neural Network designed to perform Binary Classification.

The cost function for binary classification (using sigmoid activation) is the **BCE** (supposing to consider a single training sample):

$$J = -\left[y \log a^{[3]} + (1 - y) \log\left(1 - a^{[3]}\right)\right]$$

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 $\frac{\partial J}{\partial a^{[3]}}$:

$$\frac{\partial J}{\partial a^{[3]}} = \frac{\partial\{-[y \log a^{[3]} + (1-y)\log(1-a^{[3]})]\}}{a^{[3]}} =$$

$$= -\left[\frac{1}{a^{[3]}} y + (-1)\frac{1}{1-a^{[3]}}(1-y)\right] =$$

$$= -\left[\frac{y}{a^{[3]}} - \frac{(1-y)}{1-a^{[3]}}\right]$$

Thus, the error term for the output layer becomes:

$$\boldsymbol{\delta^{[L]} = \frac{\partial J}{\partial z^{[L]}} = -\left[\frac{y}{a^{[L]}} - \frac{(1-y)}{1-a^{[L]}}\right] \odot g'(z^{[L]})}$$

Q: Why is zero initialization problematic in neural networks, and how does random initialization help overcome this issue?

Zero initialization is ineffective in training neural networks because it causes **symmetry** in weight updates: if all parameters are initialized to zero, neurons in the same layer will receive identical gradients during backpropagation, leading to identical weight updates. This prevents the network from learning distinct features and makes all neurons in a layer redundant.

To break this symmetry, **Random initialization** is used. Instead of setting weights to zero, they are initialized to small random values, typically drawn from a standard normal or uniform distribution → $W_{ij}^{[l]} \in [-\epsilon, \epsilon]$. Another very commonly used (and more effective approach) is **Xavier initialization**.

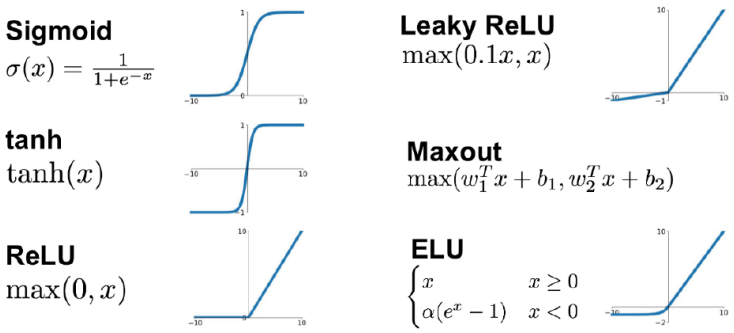
By breaking symmetry, random initialization allows different neurons to learn different features, enabling effective training.

Q: What is the role of activation functions in a neural network?

An **activation function** in a neural network introduces **non-linearity**, allowing the network to learn complex patterns and relationships in the data. Without activation functions, a neural network—regardless of how many layers it has—would reduce to a linear model, limiting its ability to solve complex problems.

Activation functions operate on the weighted sum of inputs in a neuron and determine whether the neuron should be "activated" or not.

There are different types of activation functions, each having different properties. Some of the most common are:

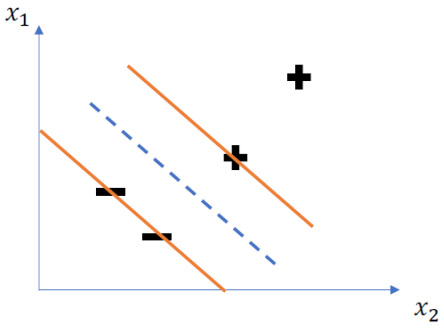


Choosing the right activation function depends on the problem being solved. For hidden layers, ReLU is commonly used due to its efficiency, while Softmax or Sigmoid is often used in output layers for classification tasks.

Q: Explain the concept of Support Vector Machines (SVM) in classification. Discuss the derivation of the optimal hyperplane, including the concept of margin maximization, and its transformation into the dual problem using Lagrange multipliers.

Support Vector Machines (SVMs) are powerful supervised learning algorithms primarily used for classification tasks. Unlike logistic regression or neural networks, which focus on probability-based decision boundaries, SVMs work by identifying an optimal linear hyperplane that maximally separates two classes in a dataset.

Let's consider a dataset consisting of m training examples $\left(x^{(i)}, y^{(i)}\right)$ where $y^{(i)} \in \{1, -1\}$, therefore we have a positive class (+1) and a negative class (-1).



The objective is to find a **hyperplane** that best separates these classes, defined as:

$$\omega^T \cdot x + b$$

Given it, the **decision rule** is based on which side of the hyperplane a point lies:

$$if\ \omega^T \cdot x^{(i)} + b \geq 0 \rightarrow y^{(i)} = +1$$

$$if\ \omega^T \cdot x^{(i)} + b < 0 \rightarrow y^{(i)} = -1$$

Since multiple hyperplanes can separate two classes, SVM selects the one that maximizes the **margin**, which is the distance between the hyperplane and the nearest data points also known as **support vectors**.

The support vectors lie on two parallel hyperplanes and are determined by the following equations:

$$\omega^T \cdot x_+ + b = 1 \qquad (for\ y^{(i)} = +1)$$

$$\omega^T \cdot x_- + b = -1 \qquad (for\ y^{(i)} = -1)$$

Let's suppose that the data points are perfectly **linear separable**. Given it, no points can exist within the margin and we can deduce that:

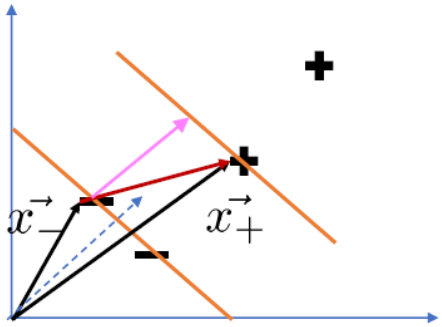
- for all values with label $y^{(i)} = +1, \quad \omega^T \cdot x_+ + b \geq 1$
- for all values with label $y^{(i)} = -1, \quad \omega^T \cdot x_- + b \leq -1$

We can rewrite these constraints in a unified form such as:

$$\boldsymbol{y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1 \geq 0 \quad \forall i = 1, \ldots, m}$$

This constraint ensures that all points are classified correctly and remain outside the margin. So, this will be the constraint for the SVM objective function.

Now, we need to find the expression of the **maximum margin**.



First, we calculate the margin width by projecting the difference vector, $x_+ - x_-$, onto the hyperplane's unit vector $\omega/\|\omega\|$ perpendicular to the margin:

$$M = (x_+ - x_-) \cdot \frac{\omega}{\|\omega\|}$$

Recalling that the two support vectors have the following equations:

$$\omega \cdot x_+ + b = 1 \qquad (for\ y^{(i)} = +1)$$

$$\omega \cdot x_- + b = -1 \quad (for\ y^{(i)} = -1)$$

the margin width expression simplifies as follows:

$$M = (x_+ - x_-) \cdot \frac{\omega}{\|\omega\|} = \left(\left(\frac{1-b}{\omega}\right) - \left(\frac{-1-b}{\omega}\right)\right) \cdot \frac{\omega}{\|\omega\|} = \frac{2}{\omega} \cdot \frac{\omega}{\|\omega\|} =\cdot \frac{2}{\|\omega\|}$$

Now, wanting to maximize the margin we will have:

$$\max(M) = \max \frac{2}{\|\omega\|}$$

Performing a couple of mathematically convenient steps we end up with the maximum margin objective function:

$$\max \frac{2}{\|\omega\|} \rightarrow \min \frac{1}{2}\|\omega\| \rightarrow \min \frac{1}{2}\|\omega\|^2$$

Therefore, we have obtained that we can maximize the margin M by minimizing the quantity:

$$\boldsymbol{\max(M) = min \frac{1}{2}\|\omega\|^2}$$

$$\boldsymbol{s.t \quad y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1 \geq 0 \quad \forall i = 1, \dots, m}$$

Since there are some constraints to be considered we can use the lagrangian multipliers to move to an unconstrained optimization problem:

$$L(\omega, b, \alpha) = \frac{1}{2}\|\omega\|^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}\left(\omega^T \cdot x^{(i)} + b\right) - 1\right]$$

where α_i are the Lagrange multipliers and L is the so called Lagrangian function.

We will now consider the **stationarity conditions** which impose that the partial derivatives with respect to the parameters of the Lagrangian must be zero, in such a way as to find the **critical points**:

- $\frac{\partial L}{\partial \omega} = \omega^* - \sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)} = 0 \qquad \Rightarrow \boldsymbol{\omega^* = \sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)}}$

  This shows us that the vector ω^* is just a linear combination of the samples $x^{(i)}$ weighted by the corresponding Lagrange multipliers α_i. Importantly **NOT ALL** the samples contribute to ω^*, but just the ones for which $\alpha_i^* \neq 0$.

- $\frac{\partial L}{\partial b} = -\sum_{i=1}^{m} \alpha_i^* y^{(i)} = 0 \qquad \Rightarrow \sum_{i=1}^{m} \boldsymbol{\alpha_i^* y^{(i)}} = \mathbf{0}$
- $\frac{\partial L}{\partial \alpha} = -\left[y^{(i)}\left(\omega^{*T} \cdot x^{(i)} + b^*\right) - 1\right] = 0 \qquad \Rightarrow \boldsymbol{y^{(i)}\left(\omega^{*T} \cdot x^{(i)} + b^*\right) - 1 = 0}$

  This shows that the only samples that contribute 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.

We can substitute the expression for ω^* that we derived above into the original Lagrangian function:

$$\frac{1}{2}\|\omega\|^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}\left(\omega \cdot x^{(i)} + b\right) - 1\right] =$$

$$= \frac{1}{2}\left(\sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)}\right) \cdot \left(\sum_{j=1}^{m} \alpha_j^* y^{(j)} x^{(j)}\right) - \left(\sum_{i=1}^{m} \alpha_i^* y^{(i)} x^{(i)}\right) \cdot \left(\sum_{j=1}^{m} \alpha_j^* y^{(j)} x^{(j)}\right) - b \sum_{i=1}^{m} \alpha_i^* y^{(i)} + \sum_{i=1}^{m} \alpha_i^*$$

Moreover, since $\sum_{i=1}^{m} \alpha_i^* y^{(i)} = 0$, 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 formulation reveals that the optimization depends only on the **inner products** between pairs of samples $\langle x^{(i)}, x^{(j)} \rangle$. This is extremely important since it will make computationally efficient and pivotal incorporate kernel methods as required from Kernel SVM.

The **dual optimization problem** is therefore defined as:

$$\max_{\alpha} \left(\sum_{i} \boldsymbol{\alpha_i^*} - \frac{1}{2} \sum_{i} \sum_{j} \boldsymbol{\alpha_i^* \alpha_j^* y^{(i)} y^{(j)}} \, \boldsymbol{\langle x^{(i)}, x^{(j)} \rangle} \right)$$

$$\boldsymbol{s.t. \quad \alpha_i^* \geq 0 \quad and \quad \sum \alpha_i^* y^{(i)} = 0 \quad \forall i, j = 1, \ldots, m}$$

The solution to this dual problem provides the optimal α_i^*.

Once the α_i^* values have been obtained we can recover the equation of the hyperplane $\omega \cdot x + b$ by:

$$\omega^* \cdot x + b^* = \sum_{i} \boldsymbol{\alpha_i^* y^{(i)} \langle x^{(i)}, x \rangle + b^*}$$

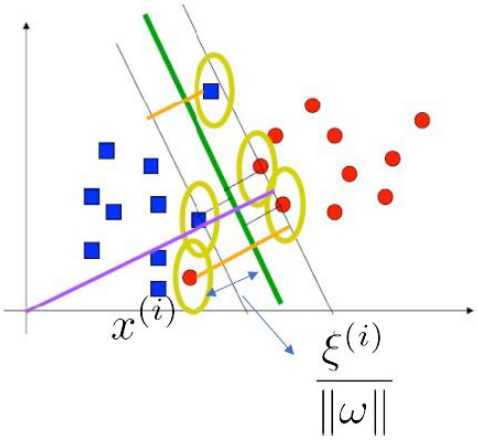
where, obviously, only the samples lying on the support vectors (samples for which $\alpha_i \neq 0$) contribute to the sum. This gives us the equation of optimal hyperplane, i.e. the separation **hyperplane with maximum margin**.

Q: Explain what the Soft-Margin SVM is and for what purpose it is used.

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**:
  $$y^{(i)}\left(\omega^T x^{(i)} + b\right) \geq 1 - \xi^{(i)}$$
  where $\xi^{(i)}$ represents how much the i-th data point deviates from the correct classification.

- An additional term C is introduced in the cost function to **penalize misclassification** errors:
  $$\frac{1}{2}\|\omega\|^2 + C\sum_{i=1}^{m}\xi^{(i)}$$
  $\boldsymbol{C}$ (**regularization** parameter) is used to control the trade-off between margin maximization and error tolerance. A higher C penalizes misclassification more heavily, leading to fewer margin violations, while a lower C allows more violations for better generalization.

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.

Q: What is meant by Kernel SVM and how does the kernel trick enable SVMs to classify non-linearly separable data efficiently?

The linear hyperplane of standard SVM can only separate linearly separable data. However, many real-world problems require non-linear decision boundaries. Instead of modifying the classifier, we can define a feature transformation which maps the data to a **higher-dimensional space** where the data becomes linearly separable making it possible to apply a simple linear hyperplane to bisect the data in the classes. This transformation is computationally expensive, but we can **avoid explicit transformation** using the **kernel trick**. **Kernel SVM** make use of a kernel function to be able handle complex fashion non-linearly separable data.

Let's see this in more detail.

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 low-dimensional space"*.

Therefore, the key idea is that, since in the dual formulation of SVM data points only appear as **inner products** $\langle \boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)} \rangle$**,** we could think to apply a transformation function on each of the original input data of the inner product which maps them into a higher-dimensional space where hopefully the data will become linearly separable:

$$\Phi\left(x^{(i)}\right) \cdot \Phi\left(x^{(j)}\right)$$

Since computing the transformation $\Phi(x)$ explicitly can be computationally expensive, we can make use of the **kernel trick**, using instead a kernel function. 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:

$$\boldsymbol{k}\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \boldsymbol{\Phi}\left(\boldsymbol{x}^{(i)}\right) \cdot \boldsymbol{\Phi}\left(\boldsymbol{x}^{(j)}\right)$$

Crucially, this is done without ever explicitly constructing or interacting with the transformed data itself. Therefore, the kernel trick enables SVMs to handle complex, non-linear relationships in the data while avoiding the computational pitfalls of high-dimensional transformations.

Different kernels enable SVM to handle various types of non-linear patterns. Some of the most popular ones are:

- Linear kernel
  $(\vec{u} \cdot \vec{v})$
- Polinomial kernel of degree d
  $(\vec{u} \cdot \vec{v} + r)^d$
- 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
  $e^{-\frac{\left(x^{(i)} - x^{(j)}\right)^2}{2\sigma^2}}$

Q: What are Generalized Linear Models (GLMs)?

Generalized Linear Models (GLMs) extend traditional linear models by allowing non-linear transformations of the input while maintaining a linear relationship.

For instance, considering a linear regression, the hypothesis function is given by:

$$h_\theta(x) = \sum_{i=1}^{m} \theta_i \cdot x^{(i)} = \theta^T x$$

Here we have a limitation of the expressiveness of the model because the hypothesis $h_\theta(x)$ is only able to approximate linear functions of the input.

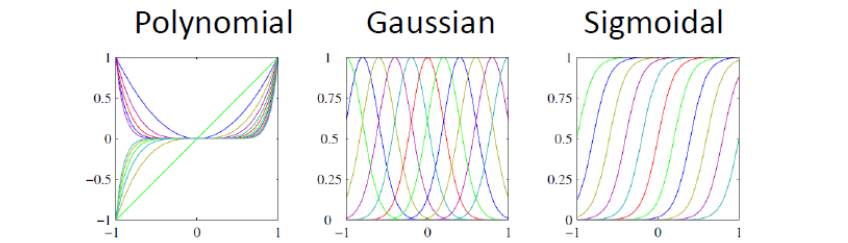
GLMs address this limitation by introducing **basic functions** $\Phi(x)$, which transform the input into a new space where the relationship can be better approximated:

$$h_\theta(x) = \sum_{i=1}^{m} \theta_i \cdot \Phi\left(x^{(i)}\right) = \theta^T \Phi(x)$$

So, we always consider linear combinations in the parameters, but they are made with respect to non-linear transformations of the input.

Different choices of basic functions $\Phi(x)$ allow GLMs to adapt to various types of non-linearity in the data:

$$\begin{aligned} &Linear && \phi_i(x) = x_i \\ &Polynomial && \phi_i(x) = x_i^p \\ &Gaussian && \phi_i(x) = e^{-\frac{x-\mu_i}{2\sigma^2}} \\ &Sigmoidal && \phi_i(x) = \frac{1}{1+e^{-\frac{x-\mu_i}{2\sigma^2}}} \end{aligned}$$



Q: What is a Decision Tree and which advantage does it give w.r.t to a neural network?

A **Decision Tree** is a predictive model structured as a tree composed of nodes. Each node represents a condition or a decision based on input attributes (features). The tree branches out based on the evaluation of these conditions, leading to a final decision or prediction at the leaf nodes.

In the decision tree structure, there are two types of nodes:

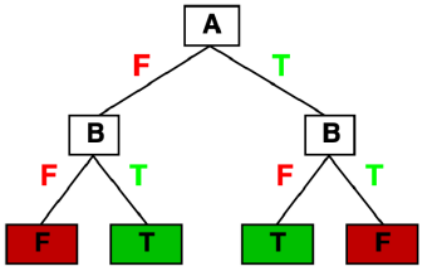
- **Decision Nodes**: These nodes represent the conditions or decision-making statements (e.g., "Is age > 30?").
- **Leaf Nodes**: These represent the final predictions or outcomes that result from evaluating the conditions along the path.

We can distinguish between two types of decision trees:

- **Classification Trees** are used when the goal is to classify data into distinct discrete categories (e.g. True or False).
- **Regression Trees** are used when the objective is to predict continuous numeric values (e.g. estimate drug effectiveness).

For instance, here is reported the structure of classification tree that implements the logic of a XOR:

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F



One important characteristic of decision trees is their **interpretability**. They offer a transparent model where you can trace the decisions step-by-step from root to leaf. This makes decision trees highly valuable when you need to explain the reasoning behind predictions in a human-understandable way, unlike black-box models like neural networks.

However, a decision tree can **overfit** to the training data, meaning it becomes too complex and specific to the training set, potentially losing its ability to generalize to new, unseen data. To avoid overfitting, the tree is often pruned or restricted to ensure it remains generalizable.

Q: What Is a Regression Tree and How It Is Built?

A **Regression Tree** is a type of decision tree used for predicting a continuous target variable. Instead of assigning a category to the data (as in classification trees), regression trees predict a continuous (numeric) value at each leaf node. The prediction is typically the average of the target variable values of the data points in that leaf.

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 Regression tree is built by recursively splitting the data into subsets (based on feature values) where each split is determined to minimize the **Residual Sum of Squares (RSS)**.

Steps to Build a Regression Tree:

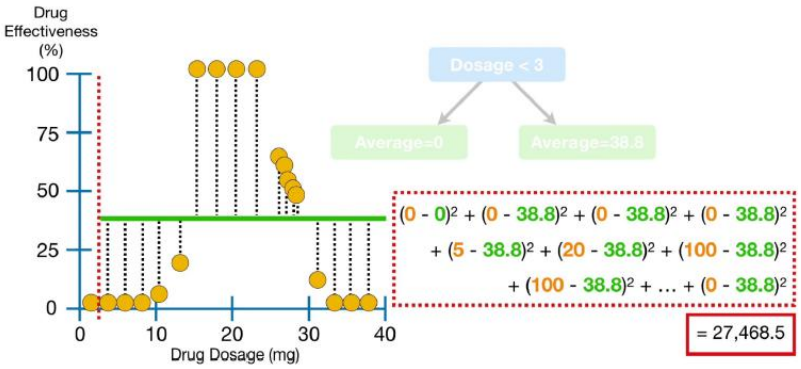
1. **Initialization**:
    - Begin with the entire dataset X, containing all observations and their corresponding target values.

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...

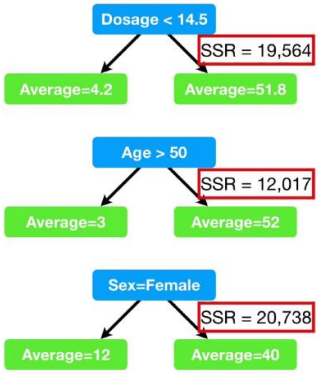
2. **Splitting Criterion**:
    - For each feature X_j, evaluate all possible thresholds t, where each threshold splits the dataset into two groups. For each split, compute the **prediction** $h(x)$ for both groups, which is the **average** target value in each group.
    - Using these predictions, calculate the **Residual Sum of Squares (RSS)**, also known as the Sum of Squared Residuals (SSR), which calculates the sum of the squared residuals:

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

Note: The **residual** is the difference between the observed and predicted values.

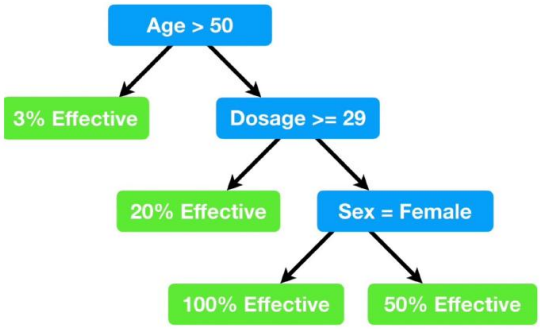


3. **Choosing the Best Split**:
    - Identify the threshold t^* for each feature that **minimizes the RSS**. This threshold becomes the **candidate split** for that feature.
    - Compare the smallest RSS values of each feature across all features to determine the **feature threshold** that yields the **overall smallest RSS**. This one defines the **root node**.



4. **Recursive Splitting**:
    - Divide the dataset based on the selected feature threshold.
    - Repeat the process recursively for each subset (resulting from the split), identifying new decision nodes thresholds, until the stopping criterion is met.
5. **Stopping Criterion**:
    - 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.

The final regression tree consists of **decision nodes** (internal nodes) and **leaf nodes** (terminal nodes). Each leaf node represents the mean target value of the observations within that node, which serves as the prediction for new observations falling into that group.



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…

Note: Regression trees can **overfit** when they perfectly fit the training data (high variance, low bias). One technique to mitigate this is **Pruning** the tree after it is built.

Advantages of Regression Trees:

1. Regression trees can model non-linear relationships, unlike linear regression models.
2. The resulting tree structure is easy to interpret and visualize.

Q: What Is a Classification Tree and How It Is Built?

A **Classification Tree** is a type of **decision tree** used for classification problems, where the goal is to assign an input to a discrete category or class. Each internal node of the tree represents a decision based on a feature, and each leaf node represents a class label (for instance in case of a binary classification task we will have the class "True" or "False").

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**.

To build a Classification Tree, we need to determine the order in which to consider the features. The ideal feature should split the dataset into subsets that are as **pure** as possible, meaning that most or all samples within a subset

belong to the same class. One approach to selecting features is to minimize misclassification errors, ensuring that each split contributes significantly to class distinction.

To quantify the informativeness of a feature, we use **Entropy**. For a random variable with n possible values, where each value i occurs with a probability p_i, the **entropy H** is defined as:

$$H(p_1, p_2, \dots, p_n) = \sum_{i=1}^{n} - p_i \log_2 p_i$$

Entropy, on the same hand, is a measure of the level of **uncertainty** or equivalently the **expected surprise** associated with a random variable. Entropy is lowest when one outcome is highly probable (low uncertainty) and highest when all outcomes are equally likely (high uncertainty).

Features that lead to lower entropy splits are preferred, as they provide clearer distinctions between classes (the greater purity).

The first step in building a classification tree is to compute the **initial entropy H_0** of the dataset before considering any split:

$$H_0 = H\left(\frac{t}{t+f}, \frac{f}{t+f}\right) = -\frac{t}{t+f}\log_2\frac{t}{t+f} - \frac{f}{t+f}\log_2\frac{f}{t+f}$$

Next, we evaluate the discriminatory power of each feature E by computing the entropy **after splitting** based on the considered feature. The **final entropy after splitting based on the feature E**, $H(E)$, is computed as the weighted sum of the entropies of its k subsets:

$$H(E) = \sum_{i=1}^{k} \frac{t_i + f_i}{t+f} H(E_i)$$

where each subset entropy $H(E_i)$ is computed as:

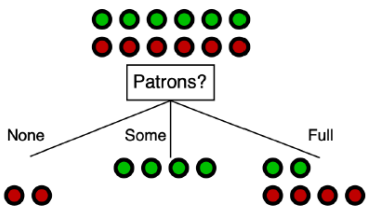
$$H(E_i) = H\left(\frac{t_i}{t_i+f_i}, \frac{f_i}{t_i+f_i}\right) = -\frac{t_i}{t_i+f_i}\log_2\frac{t_i}{t_i+f_i} - \frac{f_i}{t_i+f_i}\log_2\frac{f_i}{t_i+f_i}$$

From this, for each feature we calculate the **Information Gain (IG)**, which measures the reduction in entropy of a dataset after splitting it based on a specific feature (in this case E)

$$IG(E) = H_0 - H(E)$$

The feature with the **highest Information Gain** is chosen as the root node because it provides the greatest reduction in uncertainty.

At this point, the correctly classified pure examples are removed from the dataset **to focus only on the examples characterized by misclassification** (the ones that end up in **impure leaves**).

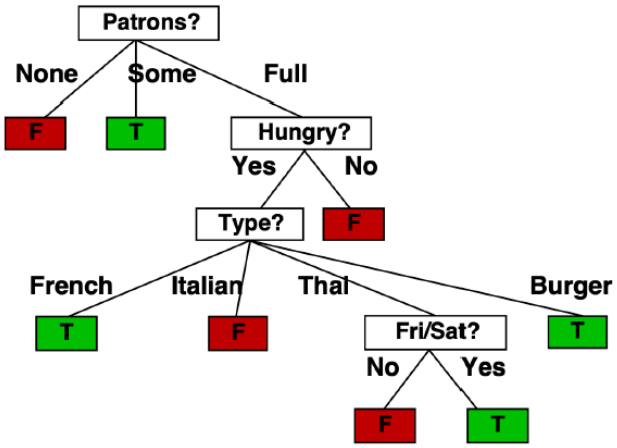


Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	~~T~~	~~F~~	~~F~~	~~T~~	~~Some~~	~~$$$~~	~~F~~	~~T~~	~~French~~	~~0–10~~	~~T~~
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	~~F~~	~~T~~	~~F~~	~~F~~	~~Some~~	~~$~~	~~F~~	~~F~~	~~Burger~~	~~0–10~~	~~T~~
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	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 as described above and the feature with the highest IG is chosen as feature to continue the build of the tree.

This process is repeated recursively until all examples are correctly classified (i.e., no more impure leaves remain).

At the end of this iterative process, we obtain a fully constructed **Classification Tree** that optimally classifies the dataset.



Q: How can numerical values be handled in Classification Trees?

Supposing having a numerical feature in a Classification Tree, we follow these steps:

1. Sort the feature values in ascending order.
2. Calculate the average between each pair of adjacent values to create possible threshold candidates.
3. Evaluate the IG for all these possible threshold candidates and select the threshold that yields the highest IG.

Once the best numerical threshold is determined, we can compare its IG with the IG of the other categorical features and select the feature with the highest IG as the best split.

Q: How can missing values be handled in a Decision Tree?

(This answer is valid not just for Decision Trees, but for datasets is general)

We talk about **missing values** when we are missing some feature values in our dataset. In such cases a first approach, if there are just a few, could be simply dropping the rows containing them. Otherwise, if we want to keep all of them there are different strategies to fill those missing values, depending on whether the missing feature is categorical or numerical.

For **categorical features**, two common approaches are:

1. Fill the missing value with the **most frequent** value of that feature. For example, if the "Blocked Arteries" feature has a missing value and "Yes" appears more frequently than "No", we replace the missing value with "Yes".

2. Alternatively, we could find a **highly correlated feature and use it as guideline** to infer the missing value. For instance, if "Chest Pain" and "Blocked Arteries" show similar patterns, we can predict the missing value in "Blocked Arteries" based on the corresponding "Chest Pain" value.

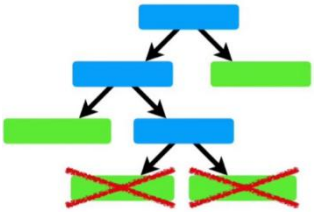
For **numerical features**, some common techniques include:

1. Replace the missing value with the **mean** or **median** of the feature values.
2. If another numerical feature has a high correlation with the missing-value feature, we can perform **linear regression** and use the least square line to predict the missing value. For example, if "Height" is strongly correlated with "Weight," we can use a regression model to predict the missing weight based on the given height.

Q: What is the purpose of pruning a Regression Tree and why is it applied?

The main purpose of **pruning** a regression tree is to **prevent overfitting** the training data, ensuring that the tree generalizes well to unseen test data. Regression trees are prone to overfitting because they can become overly complex by fitting the training data too closely, which can lead to poor performance on new unseen data.

Pruning helps address this issue by **removing some of the leaves** and replacing them with a single leaf representing the **average of a larger set of observations**. By doing so, we reduce the tree's complexity and make it more robust to new, unseen data.



Pruning can be applied at different points of the tree, leading to various pruned versions. The challenge is determining which pruned tree is the best. To assess the quality of the pruning, we can use **Weakest Link Pruning**.

Weakest Link Pruning works by calculating a **Tree Score** given by:

$$\boldsymbol{Tree\ Score} = \boldsymbol{SSR} + \boldsymbol{Tree\ Complexity\ Penality} =$$

$$= \boldsymbol{SSR} + \boldsymbol{\alpha T}$$

which combines two components:

- **SSR**: The sum of squared residuals, which measures the prediction error.
- **Tree Complexity Penalty ($\boldsymbol{\alpha T}$)**: This penalty is based on the number of leaves (T) in the tree and a regularization parameter α. The complexity penalty increases with more leaves, encouraging simpler trees. **Note**: By increasing the regularization parameter α, the tree becomes more penalized for complexity, which encourages pruning.

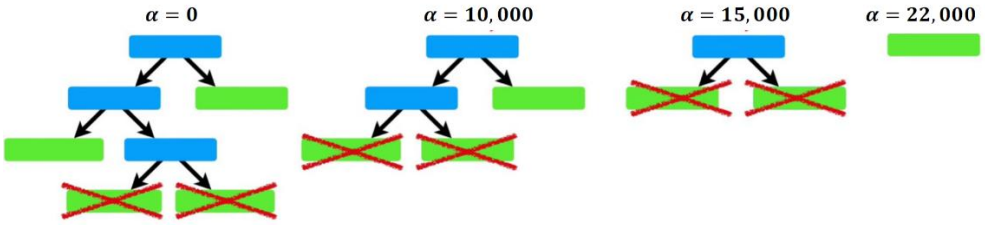
To determine the best pruned tree, we apply **Weakest Link Pruning** by calculating the Tree Score for the tree and each of its sub-trees, and then selecting the one with the **lowest Tree Score**.

However, the Tree Score is dependent on the value of α, which plays a critical role in determining the best sub-tree. Different values of α can lead to different pruned trees. Therefore, we need to find the **optimal α value**. To do it we proceed as follows.

First, we build a full-sized regression tree using **all** the data (both training data and test data). This tree will have the **lowest Tree Score when $\alpha = 0$** because, at this point, there is no penalty for complexity.

Then we gradually increase α 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, this process produces a sequence of pruned trees, each optimal for a specific α value (i.e. each tree minimizes the Tree Score for that particular α).



So now we need to identify the best overall α value to choose.

Therefore, the next step is to evaluate each potential value of α obtained from the pruning process in order to determine the optimal overall α value. We can do this using **10-Fold Cross-Validation**. 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.

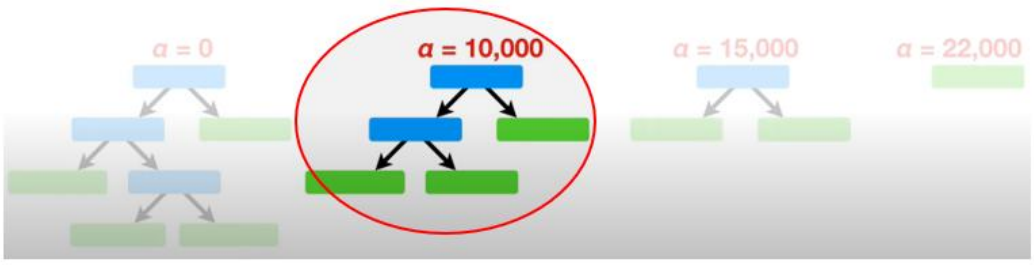
During the procedure all values of α found are kept **fixed**.

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 Tree Score** across the 10 folds is selected as the **final α**.

Lastly, once we determined the optimal α, we can return to the original tree and sub-trees obtained in the step before the evaluation and from these we pick the tree/sub-tree that corresponds to the final value for α that we found.



This **final pruned tree** is considered the best model because it balances model fit and complexity, minimizing overfitting.

Q: What is a Random Forest, and how does it address the limitations of Decision Trees?

Decision Trees are easy to interpret but suffer from **overfitting**, meaning they perform well on the data used to create them, but they struggle to generalize to new data, leading to **poor accuracy**.

Random Forests are **ensemble learning algorithms** that solve this problem by constructing multiple Decision Trees and combining their outputs to produce a more **robust and reliable** prediction.

Random Forest uses an ensemble technique known as **Bagging (Bootstrap Aggregating)** which means **b**ootstrapping the data plus **agg**regating the predictions obtained from the different trees to make a decision. More specifically:

- During the training phase we perform the **b**ootstrapping of the data:
  - Different **Bootstrapped datasets** are created, which are datasets with samples selected randomly from the original dataset **with replacement** (i.e. some samples can be picked more than once) having the same size as the original. Each of them is used as training data to build an individual tree.
  - At each step in constructing an individual tree, only a **random subset of features** is considered, which ensures diversity among the trees.
    Note: The **number of random features to consider** 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.
  - At the end of this process we end up with a forest of decision trees.

- During the prediction phase, given a new sample we run it down in each ensemble tree and we aggregate the results to obtain the final prediction:
  - in case of a **classification** we take the **majority vote** among all trees.
  - in case of a **regression** we compute the **average** prediction of all trees.

Since each tree in a **Random Forest** is trained on a **bootstrapped dataset** some samples are **left out** during the training process. These left-out samples are known as **Out-of-Bag (OOB) Data** and can be used to evaluate the prediction error of the random forest.

To calculate the OOB error, 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.

The **Out-Of-Bag error** is then 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.

Q: How does Random Forest handle missing values?

Random Forest considers two types of missing values:
- Missing values in the original dataset used to create a random forest.
- Missing values in a new sample that we want to categorize through a random forest.

Missing Values in the Dataset:

1. Initially, a guess is made for the missing values based on the most frequent category for categorical features and the median for continuous features.

2. Then, the algorithm builds a random forest and runs all the data through the trees. It identifies which samples are similar to the one with missing values by checking which samples end up in the same leaf of each tree. Specifically, a **Proximity Matrix** is used to track these similarities, where a value of "1" is added for each pair of samples that end up in the same leaf. This matrix is then normalized with respect to the total number of trees considered.
3. The proximity matrix is then used to refine the guesses. For categorical features, the missing value is imputed based on a **weighted frequency** and for continuous features, it is imputed with a **weighted average**, using proximity values as the weights.
4. This process is repeated iteratively until the missing values converge, meaning that the values obtained no longer change.

Missing Values in a New Sample:

1. First, we create two copies of the new sample—one assuming the prediction value of the positive class (e.g. "Yes") and the other one with that of the negative class (e.g. "No").
2. Then we use the same iterative method that we saw above to handle missing values in the dataset to make a good guess about the missing values.
3. These copies are then run through the forest, and the predictions are compared. The missing value is filled in with the option that receives the most votes from the trees.

Q: How does the K-Means algorithm work and what are its advantages and disadvantages?

K-means is a popular clustering used to partition a dataset into a predefined number of clusters, "K" (hyperparameter).

It starts by randomly initializing K **centroids** $\mu_1, \mu_2, \dots, \mu_K$, which represent the center of each cluster. The algorithm then proceeds with the following steps:

1. **Assignment Step**: Each data point $x^{(i)}$ is assigned 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 $\mu_1, \mu_2, \dots, \mu_K$.

2. **Update Step**: After assigning all data points to clusters, the centroids are updated by calculating the mean of all data points in each cluster. This new mean becomes the new centroid.

These two steps (assigning points to clusters and updating centroids) are repeated iteratively until convergence, meaning there are no more switches of data points between clusters or until a certain number of epochs is reached.

K-means clustering algorithm:

Randomly initialize K cluster centroids $\mu_1, \mu_2, \dots, \mu_K$.

Repeat:

for $i = 1$ *to* m

$$C_k^{(i)} := arg \min_{k \in \{1,\dots,K\}} d\left(x^{(i)}, \mu_k\right)$$

for $k = 1$ *to* K

$$\mu_k := \frac{1}{|C_k|} \sum_{x^{(i)} \in C_k} x^{(i)}$$

The K-means algorithm itself can be interpreted as an application of the Expectation Maximization (EM) algorithm.

Given the cost function of K-means defined as the ***distortion measure***:

$$\boldsymbol{J = \frac{1}{m} \sum_{k=1}^{K} \sum_{i=1}^{m} r_{ik} \left\| x^{(i)} - \mu_k \right\|^2}$$

K-Means algorithm aims to minimize J with respect to r_{ik} and μ_k iteratively alternating 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:

$$\boldsymbol{r_{ik} \rightarrow \begin{cases} 1, & if\ k = \underset{j}{\operatorname{argmin}} \left\| x^{(i)} - \mu_j \right\|^2 \\ 0, & otherwise \end{cases}}$$

- **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 $\boldsymbol{\sum_{i=1}^{m} r_{ik} = |C_k|}$ (number of data points in the k-th cluster C_k).

K-Means – Random Initialization: The algorithm's performance depends heavily on the initial placement of the centroids. Since the centroids are chosen randomly, different runs can lead to different results, potentially converging to **local minima** instead of the global optimum. To mitigate this, K-means is often **run multiple times** with different initializations and the best clustering solution is chosen based on the lowest cost function 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.

Q: How does the K-Medoids algorithm work and what are its advantages and disadvantages?

K-Medoids is a clustering algorithm that is conceptually similar to K-means, but with a crucial difference: K-medoids uses actual data points as the centers (medoids), in contrast to K-means where the center of a cluster is not necessarily one of the input data points (it is just the mean between the points in the cluster).

This makes K-medoids more robust to noise and outliers compared to K-means because the medoid is a more centrally located point within the cluster and is not influenced by extreme values as much as the centroid in K-means.

The algorithm follows these steps:

- **Initialization**: K points from the dataset are randomly selected as initial medoids, initializing M, i.e. the set of medoids.

- **Assignment of points to clusters**: each point $x^{(i)} \in X$ is assigned to the cluster of the closest medoid:

$$C_k^{(i)} := arg \min_{k \in \{1,\dots,K\}} d\left(x^{(i)}, \mu_k\right)$$

  where $d\left(x^{(i)}, \mu_k\right)$ represents the distance (i.e. Euclidean distance) between the point and the medoid.

- **Cost calculation**: the total cost of clustering is calculated as the sum of distances between points and their assigned medoids:

$$Cost = \sum_{k=1}^{K} \sum_{x^{(i)} \in C_k} d\left(x^{(i)}, \mu_k\right)$$

- **Update of medoids**: 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 cost undo the swap, otherwise (confirm the swap) update the cost as equal to this lower NewCost.

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 $NewCost$*

            if *$(NewCost > Cost)$* ***then*** *undo swap*

            else $Cost = NewCost$

K-Medoids – PROS/CONS:

- PROS
    - It is less sensitive to outliers than K-means.
- CONS:
    - 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.

Q: Explain how a Gaussian Mixture Model works and how it finds the best parameters estimates.

A **Gaussian Mixture Model (GMM)** is a clustering technique that models data as a **mixture of multiple Gaussian distributions**, where each Gaussian represents a distinct cluster. Unlike K-means, which assigns each data point to a single cluster (hard assignment), GMM provides a **soft assignment**, meaning each data point has a probability of belonging to multiple clusters rather than being assigned to just one. This makes it ideal for datasets with overlapping clusters or complex distributions.

The probability of a single data point $x^{(i)}$ belonging to the mixture of the K Gaussians is given by:

$$p(x^{(i)}|\theta) = \sum_{k=1}^{K} \pi_k \, \mathcal{N}(x^{(i)}|\mu_k, \Sigma_k)$$

where:

- π_k is the **mixing coefficient** of the k-th Gaussian. These coefficients satisfy $\sum_{k=1}^{K} \pi_k = 1$.
- $\mathcal{N}(x^{(i)}|\mu_k, \Sigma_k)$ represents the probability density of the data point $x^{(i)}$ to belong to the k-th multivariate Gaussian.

Each of the multivariate gaussian component under the mixture perspective can be defined as:

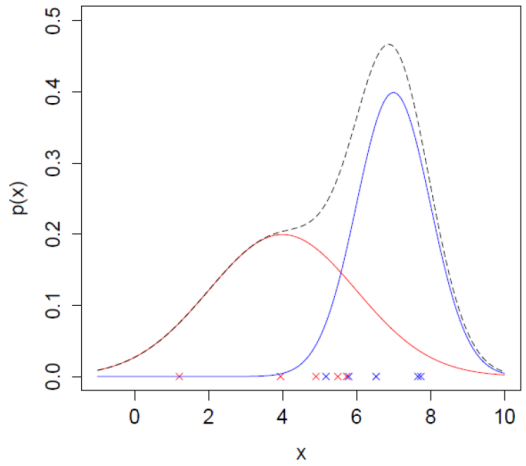
$$\mathcal{N}(x^{(i)}|\mu_k, \Sigma_k) = \frac{e^{\left(-\frac{1}{2}(x-\mu_k)^T \Sigma_k^{-1}(x-\mu_k)\right)}}{\sqrt{(2\pi)^n |\Sigma_k|}}$$

where:

- μ_k is the vector of the means, representing the center of the cluster.
- Σ_k is the covariance matrix, defining the shape and orientation of the cluster.

The main purpose of the GMM is to determine the parameters $\theta = [\pi_k, \mu_k, \Sigma_k]$ of the K gaussians of the mixture that **maximize the likelihood** of the data:

$$L(\theta) = \prod_{i=1}^{m} p(x^{(i)}|\theta)$$



Since direct maximization is difficult, to find these parameters GMM uses the **Expectation-Maximization (EM)**, an iterative algorithm which works as follows:

1. *Initialize the parameters θ (μ_k, Σ_k and π_k) for each of the K gaussians to random values.*

2. *Repeat {*

    1. ***(E) Expectation step**: we fix the parameters θ^{t-1} and for each cluster-point pair, we compute a **responsibility coefficient r_{ik}** which 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.*

    $$r_{ik} = \frac{\pi_k \, p_k(x^{(i)}|\theta_k^{t-1})}{\sum_{l=1}^{K} \pi_l \, p_l(x^{(i)}|\theta_l^{t-1})}$$

    ***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:*

    $$\pi_k = \frac{1}{m} \sum_{i=1}^{m} r_{ik}$$

$$\mu_k = \frac{\sum_i r_{ik} x^{(i)}}{\sum_i r_{ik}}$$

$$\Sigma_k = \frac{\sum_i r_{ik} \left((x^{(i)} - \mu_k)(x^{(i)} - \mu_k)^T \right)}{\sum_i r_{ik}}$$

} until convergence

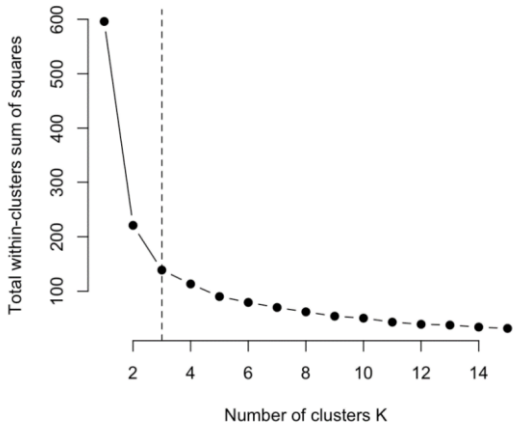
Gaussian Mixture Models – PROS/CONS:

- 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.

Q: How can we determine the optimal number of clusters (K) in clustering algorithms?

There are several methods that can guide the selection of the optimal number of clusters (**K**), a crucial hyperparameter for a lot of clustering algorithms:

1. **Elbow Method** – It involves plotting the **sum of squared distances** between data points and their assigned cluster centroids (y-axis) for **different values of K** (x-axis). The optimal number of clusters K is identified by the **elbow** of the graph.

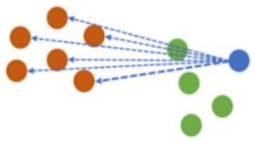


2. **Silhouette Method** – Measures the quality of separation between clusters considering cohesion and separation. Specifically, the **Silhouette Coefficient** is a measure of how similar a data point is within-cluster (**cohesion**) compared to other clusters (**separation**).

   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:

$$\boldsymbol{s_i} = \mathbf{1} - \frac{\boldsymbol{coh}}{\boldsymbol{sep}} \quad if\ coh < sep$$

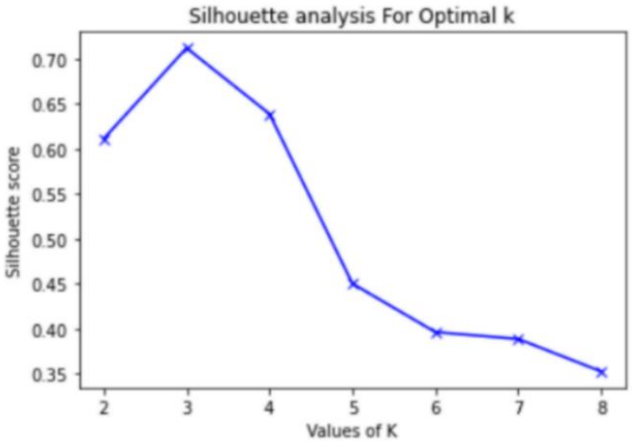
The value of the silhouette coefficient s_i is between $[-1, 1]$:

- 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 of K, we can plot a graph of silhouette scores versus number of clusters. The **optimal K is the one that maximizes the average silhouette score**.



3. **Kullback-Leibler (KL) Divergence** – Measures as much as the distribution found by the clusters differs from the true distribution:

$$\boldsymbol{D_{KL}(P||Q) = -\sum_i P(i)\log\frac{Q(i)}{P(i)} = \sum_i P(i)\log\frac{P(i)}{Q(i)}}$$

where $\boldsymbol{P}$ is the true distribution and $\boldsymbol{Q}$ is the model distribution. A KL divergence value close to 0 indicates good similarity between the distributions.

Once the KL divergences have been calculated for different values of K, the K that minimizes KL divergence must be chosen. This method is useful in probabilistic contexts, but requires a true distribution or an accurate estimate of this distribution.

4. **Information Criteria** – These metrics assess how well the clustering model fits the data while penalizing unnecessary complexity (i.e. to avoid overfitting). Some of them are:

    - **AIC (Akaike Information Criterion)**: is based on a penalty for the complexity of the model

    $$AIC = 2K - 2\ln(\hat{L})$$

    where K is the number of parameters (in clustering context they can be the number of clusters) and $\hat{L}$ is the maximum-likelihood estimate of the model.

    The model that minimizes AIC is selected.

    Furthermore, we can consider a different version of AIC, namely **corrected AIC (AICc)**, which adds a correction for small datasets

    $$AICc = AIC + \frac{2K(K+1)}{m-K-1}$$

    where m is the number of samples. Thanks to this, AICc is more reliable when the sample size is small relative to the number of model parameters.

    - **BIC (Bayesian Information Criterion)**: Similar to AIC, but with a stronger penalty for complexity

    $$BIC = \ln(m)K - 2\ln(\hat{L})$$

    The model that minimizes BIC is selected.

    This method favors simpler models, but is only valid if $m \gg K$.

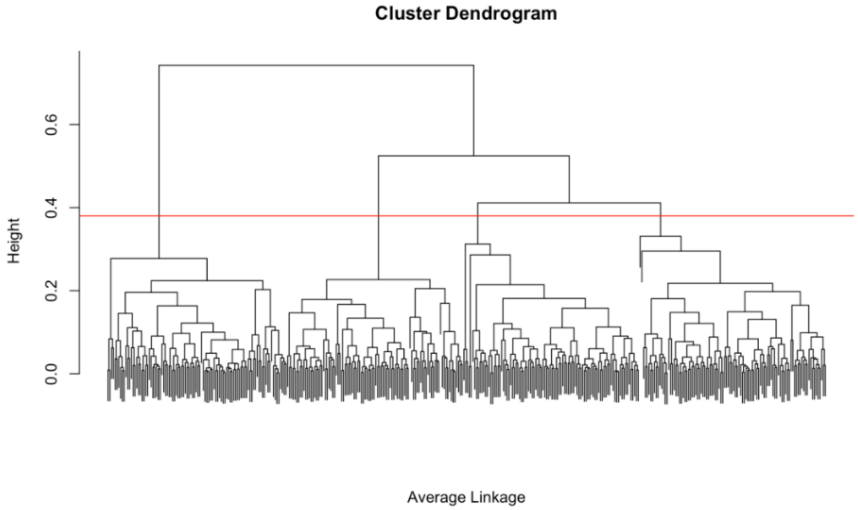
Q: What is Hierarchical Clustering and what are its advantages and disadvantages?

Hierarchical Clustering involves building a hierarchy of clusters. This hierarchy can be of two types:

- **Agglomerative ("bottom-up")**: each point starts as a separate cluster, and then iteratively merge the most similar clusters, up to form a single cluster that includes all the points.
- **Divisive ("top-down")**: we start from a single cluster that includes all points and divides iteratively until each point is an individual cluster.

Once the hierarchy is built, the final number of clusters can be chosen by "**cutting**" the hierarchy at a specific height.

Hierarchical clustering can be visualized as a **Dendrogram**, a tree-like diagram that records the sequences of merges or splits.



To decide how to merge or split clusters, it is necessary to define a metric of distance or similarity.

Specifically, 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.

An advantage of hierarchical clustering is that it is a versatile approach that does not require specifying the number of clusters a priori (K). However, it can be computationally expensive and depends on the choice of the metric and linkage criterion adopted.

Q: What is The Single Linkage criterion and how it works?

The **Single Linkage** method is one of the most well-known hierarchical clustering algorithms and works as follows:

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** between any two points in different clusters.

Q: Explain how DBSCAN clustering algorithm works and what are its advantages and disadvantages.

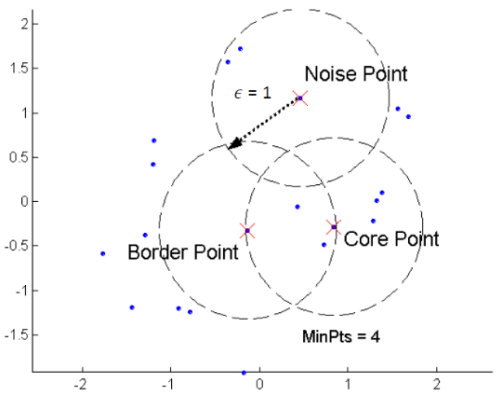
DBSCAN, which stands for "**D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise," is a **density-based** algorithm. This means that it works by grouping data points based on regions of high density, separating these regions from areas of low density.

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). Moreover, with **ϵ-Neighborhood** of a point we refer 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*.
- **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).
- **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. All the other points not considered are treated as Noise points and 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 – 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.
- 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.

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 datasets 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.

Q: Explain how HDBSCAN works.

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.

HDBSCAN works following these steps:

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 as 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:

Pros:

- Can handle varying densities
- The algorithm is stable over runs and hyperparameter choices
- Robust to outliers

Cons:

- Some important hyperparameters must be set by hand $(k, minClusterSize)$

Q: How can Gaussian Mixture Models (GMM) be used for anomaly detection?

Anomaly detection involves identifying unusual data points, referred to as **anomalies**, which deviate significantly from the majority of observations in a dataset referred to as **normal data** (**unbalanced dataset**).

In this context, **Gaussian Mixture Model (GMM)** can be used to detect anomalies in new data. The approach works by training a GMM only on **non-anomalous data** so that it learns the underlying distribution of the normal data. 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:

$$p(x_{test}^{(i)}|\theta) = \sum_{j=1}^{K} \pi_j \, p_j(x_{test}^{(i)}|\theta_j)$$

If a data point has a low probability (i.e., it does not fit well with the learned normal distribution), it is considered an anomaly. To classify a point as an **anomaly**, a threshold ϵ is set. If the probability of the data point is lower than this threshold, it is flagged as an anomaly:

$$p(x_{test}|\theta) < \epsilon \rightarrow flag\ anomaly$$

$$p(x_{test}|\theta) \geq \epsilon \rightarrow OK$$

The value of the threshold ϵ is a hyperparameter that needs to be carefully tuned. A well-chosen threshold ensures accurate anomaly detection without misclassifying too many normal data points as anomalies.

Q: What is the curse of dimensionality, and how does it affect machine learning models?

The **curse of dimensionality** refers to the phenomenon where increasing the number of features (dimensions) in a dataset leads to a drastic increase in the volume of the feature space, causing the data to become sparse. This sparsity makes it difficult for machine learning models to generalize well and can lead to decreased performance.

Initially, adding more features can improve classification, but after reaching an **optimal number of dimensions**, further increasing the dimensionality without adding more training samples **reduces model performance**. This happens because:

- The available data points become more **sparse**, making it harder to estimate model parameters accurately.
- **Overfitting** becomes more likely, as each new feature increases the number of possible data point combinations, requiring **exponentially more data** to maintain good coverage (which is often impractical).

To mitigate the curse of dimensionality, **dimensionality reduction techniques** are often employed to reduce the number of features while retaining the most important information.

Q: What is Dimensionality Reduction, and why is it useful?

Dimensionality reduction is the process of reducing the number of features (dimensions) in a dataset while preserving the most relevant information. This is done to address challenges caused by the **curse of dimensionality** and to improve the performance of machine learning models.

The key benefits of dimensionality reduction include:

1. **Reduced Data Complexity** – Eliminates redundant or less informative features, making models simpler and more efficient.
2. **Improved Model Performance** – Helps prevent overfitting by reducing the number of parameters to estimate, especially when training data is limited.
3. **Better Data Visualization** – High-dimensional data cannot be visualized directly. By reducing dimensions to **2D** or **3D**, we can graphically represent data, helping to identify patterns or clusters.

Q: What is Principal Component Analysis (PCA) and how does it work?

Principal Component Analysis (PCA) is a dimensionality reduction technique that reduces the dimensionality of a dataset while preserving as much variance as possible. It does this by finding a new coordinate system, defined by **principal components (PCs)**, which are the directions in the data that maximize variance.

More specifically, PCA finds **orthonormal** principal components such that:

- The **first principal component** u_1 captures the most variance in the data.
- The **second principal component** u_2 captures the next highest variance, ensuring it is orthogonal to u_1 (accounting only for the variance not already captured by the first component)
- etc.

Then PCA selects the **first k principal components**, condensing most of the original variance into this reduced set of dimensions.

Mathematically, PCA minimizes the **reconstruction error**, which is the average squared projection error between the original data points and their projections onto a lower-dimensional space. Moreover, it can be demonstrated that **minimizing the reconstruction error is equivalent to maximizing the variance of the projected data**.

The optimal principal components (PCs) that maximize the variance of the projected data are 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:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$x_j^{(i)} \leftarrow x_j^{(i)} - \mu_j \qquad for\ i = 1, \dots, m\ \ \&\ \ j = 1, \dots, n$$

   If the features have different scales, it is useful to normalize them:

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j} \qquad for\ i = 1, \dots, m\ \&\ j = 1, \dots, n$$

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) $\boldsymbol{u_1, u_2, \dots, u_k}$ **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 equation:

$$\hat{\Sigma} u_j = \lambda_j u_j$$

   for every principal component u_j with $j = 1, \dots, k$.

PCA can also be more efficiently computed using **Singular Value Decomposition (SVD)**.

Given a dataset $X = \sum_{i=1}^{m} x^{(i)}$ of dimensions $n \times m$, we first ensure the data is properly centered and, if necessary, normalized as we did above.

Once the data is preprocessed, we perform the SVD of the data matrix X:

$$X = USV^T$$

*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**.*

It can be demonstrated that the right singular vectors (columns) of U are the **eigenvectors** of Σ and the singular values s_j are the **square roots of the eigenvalues** of Σ (already ordered by increasing variance).

Therefore, to obtain the top k principal components we simply need to select the first k columns of U, obtaining U_{reduce}.

Given it, we can project the data onto the new k-dimensional space by:

$$Z = U_{reduce}^T X$$

Optionally we can reconstruct the data starting from the compressed one as:

$$\tilde{X} = U_{reduce} Z$$

obtaining an **approximation** of the original data.

Q: How can we determine the optimal number of principal components k in PCA?

To determine the optimal number k of principal components we can proceed as follows:

1. We perform a **single** Singular Value Decomposition (SVD) on the dataset, decomposing the data matrix X (already centered or normalized) as:

   $$X = USV^T$$

   where S contains the singular values, which are directly related to the amount of variance captured by each principal component.

2. Iteratively we select values of k, starting from $k = 1$ and increasing gradually ($k < n$), to compute the **cumulative explained variance** and check if it exceeds a predefined **variance threshold** (i.e. **0.95**):

   $$\frac{\sum_{j=1}^{k} s_j}{\sum_{j=1}^{n} s_j} \geq variance\_treshold$$

   where s_j are the singular values.

The **cumulative explained variance** gives us a measure of how much of the total variance is retained as we include more principal components. With this approach we determine the smallest number of principal components required to capture a predefined level of variance in the data.

Q: What is Kernel PCA, and how does it address the limitations of traditional PCA in handling non-linear structures in data? Provide the full mathematical derivation, including the normalization of the kernel matrix.

Traditional Principal Component Analysis (PCA) is effective for identifying linear manifolds in a dataset. However, it struggles to distinguish non-linear structures from no structure (random noise), making it inadequate for many real-world applications where data exhibits complex non-linearity. **Kernel PCA** overcomes this limitation by using a nonlinear kernel function $k\left(x^{(i)}, x^{(j)}\right)$.

Formally, 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.

Assuming the projected vectors in this new space are **zero-mean** (a condition we will address later), the covariance matrix in the Φ-space is given by:

$$\hat{\Sigma} = \frac{1}{m}\sum_{i=1}^{m} \Phi(x^{(i)})\Phi(x^{(i)})^T$$

We want to solve the eigenvalue problem:

$$\hat{\Sigma}u_j = \lambda_j u_j \qquad with\ j = 1, \dots, n$$

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, each 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, substituting this representation of u_j back into the eigenvector equation, we obtain:

$$\frac{1}{m}\sum_{i=1}^{m} \Phi(x^{(i)})\Phi(x^{(i)})^T \left(\sum_{l=1}^{m} \alpha_j^{(l)} \Phi(x^{(l)})\right) = \lambda_j \sum_{l=1}^{m} \alpha_j^{(l)} \Phi(x^{(l)})$$

$$\rightarrow \frac{1}{m}\sum_{i=1}^{m} \Phi(x^{(i)}) \left(\sum_{l=1}^{m} \alpha_j^{(l)} \Phi(x^{(i)})^T \Phi(x^{(l)})\right) = \lambda_j \sum_{l=1}^{m} \alpha_j^{(l)} \Phi(x^{(l)})$$

Now to express this in terms of the kernel function $k\left(x^{(i)}, x^{(l)}\right) = \Phi\left(x^{(i)}\right)^T \Phi\left(x^{(l)}\right)$ we can multiply both sides by $\Phi\left(x^{(k)}\right)^T$ to the left:

$$\frac{1}{m}\sum_{i=1}^{m} \Phi\left(x^{(k)}\right)^T \Phi\left(x^{(i)}\right)\left(\sum_{l=1}^{m} \alpha_j^{(l)} \Phi\left(x^{(i)}\right)^T \Phi\left(x^{(l)}\right)\right) = \lambda_j \sum_{l=1}^{m} \alpha_j^{(l)} \Phi\left(x^{(k)}\right)^T \Phi\left(x^{(l)}\right)$$

$$\rightarrow \frac{1}{m}\sum_{i=1}^{m} k\left(x^{(k)}, x^{(i)}\right)\left(\sum_{l=1}^{m} \alpha_j^{(l)} k\left(x^{(i)}, x^{(l)}\right)\right) = \lambda_j \sum_{l=1}^{m} \alpha_j^{(l)} k\left(x^{(k)}, x^{(l)}\right)$$

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 obtaining:

$$\boldsymbol{K\alpha_j = \lambda_j m \alpha_j}$$

where K is the **kernel matrix**, whose ij-th element is $k\left(x^{(i)}, x^{(j)}\right)$.

Thus, $\boldsymbol{\alpha_j}$ **is actually an eigenvector of** $\boldsymbol{K}$ with eigenvalue λ_j (scaled by m).

This eigenvalue problem formulation is what we solve during the training phase of kernel PCA to compute the eigenvectors α_j. These eigenvectors encode the principal components in the feature space implicitly, without ever computing $\Phi(x)$ explicitly.

Similar to standard PCA, after solving for these eigenvectors, we sort the eigenvalues of the kernel matrix K in descending order and select the top k eigenvectors $\alpha_1, \ldots, \alpha_k$ corresponding to the largest eigenvalues λ_j. These selected $\alpha_k s$ vectors represent the principal components in kernel PCA.

This means that given an eigenvector α_k of K, the corresponding eigenvector in the Φ -space can be obtained as:

$$u_k = \sum_{i=1}^{m} \alpha_k^{(i)} \Phi\left(x^{(i)}\right)$$

and for a new data point x, its **projection onto the** $\boldsymbol{k}$ **principal component** can be therefore obtained by:

$$\boldsymbol{z_k = u_k^T \Phi(x)} = \sum_{i=1}^{m} \alpha_k^{(i)} \Phi\left(x^{(i)}\right)^T \Phi(x) = \sum_{\boldsymbol{i=1}}^{\boldsymbol{m}} \boldsymbol{\alpha_k^{(i)} k\left(x^{(i)}, x\right)}$$

Normalization: So far, we have assumed that the projected data given by $\Phi\left(x^{(i)}\right)$ has zero mean in the Φ-space, 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. However, we can work around this by formulating the centering operation **entirely in terms of the kernel function**, so we never need to work directly in the feature space.

The projected data point after centralizing, denoted $\bar{\Phi}\left(x^{(i)}\right)$, is 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 element of the kernel matrix is given by:

$$\bar{K}_{ij} = \bar{k}\left(x^{(i)}, x^{(j)}\right) = \bar{\Phi}\left(x^{(i)}\right)^T \bar{\Phi}\left(x^{(j)}\right) =$$

$$= \Phi(x^{(i)})^T \Phi(x^{(j)}) - \frac{1}{m}\sum_{l=1}^{m} \Phi(x^{(i)})^T \Phi(x^{(l)}) - \frac{1}{m}\sum_{l=1}^{m} \Phi(x^{(l)})^T \Phi(x^{(j)}) + \frac{1}{m^2}\sum_{r=1}^{m}\sum_{l=1}^{m} \Phi(x^{(r)})^T \Phi(x^{(l)}) =$$

$$= k(x^{(i)}, x^{(j)}) - \frac{1}{m}\sum_{l=1}^{m} k(x^{(i)}, x^{(l)}) - \frac{1}{m}\sum_{l=1}^{m} k(x^{(l)}, x^{(j)}) + \frac{1}{m^2}\sum_{r=1}^{m}\sum_{l=1}^{m} k(x^{(r)}, x^{(l)})$$

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 $1/m$.

As a result, the only modification we need to make in our eigenvalue problem is to **replace the original kernel matrix K with the centered kernel matrix $\bar{K}$ during training**. The rest of the algorithm remains the same: we compute the eigenvalues and eigenvectors of $\bar{K}$, sort them, and use the top k eigenvectors to define the principal components in the feature space.

Q: What is the Independent Component Analysis (ICA) technique? Explain it in detail discussing its key assumptions, the ambiguities and how the algorithm works.

Independent Component Analysis (ICA) is a technique used to decompose mixed signals into their **independent** sources.

As ICA 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 is to separate the original speech signals from these mixtures.

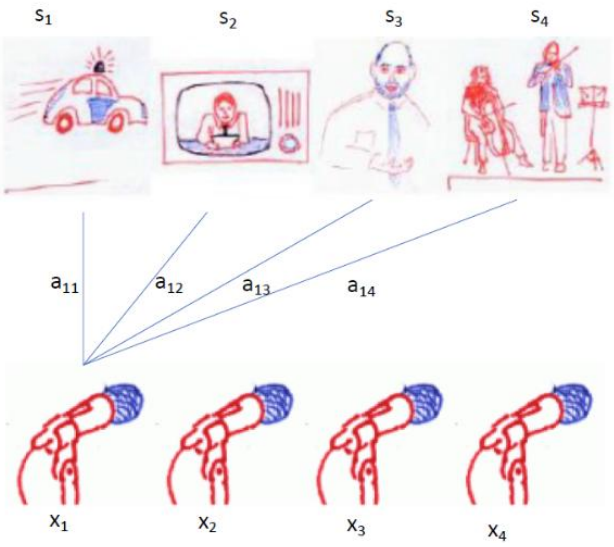
Mathematically, let $s^{(i)}$ represent the speakers' speech signals and $x^{(i)}$ represent the recorded mixed signals captured by the microphones (the mixtures). Assuming the number of microphones equals the number of speakers (k), each recorded signal can be expressed as **a linear combination** of the source signals:

$$x^{(i)} = a_{i1}s^{(1)} + a_{i2}s^{(2)} + \cdots + a_{ik}s^{(k)}$$

where $a_{i1}, a_{i2}, \ldots, a_{ik}$ are coefficients determined by the distances between the microphone and the speakers. In matrix notation, this can be written as:

$$\boldsymbol{x = As}$$

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**.



The goal of ICA is to determine the ***unmixing matrix*** $\boldsymbol{W = A^{-1}}$ so that we can recover the sources using

$$\boldsymbol{s = Wx}$$

In order to make ICA work some **assumptions** have to be made:

1. The source signals must be **statistically *independent*** (meaning that the occurrence of one event makes it neither more nor less probable that the other occurs). For this reason, we call them **independent components**.
2. The independent components must have ***non-Gaussian*** distributions.
3. For simplicity, we assume that the number of independent components is equal to the number of observed mixtures leading to a ***squared*** **mixing matrix**.

ICA Ambiguities: In the context of use of the ICA algorithm, there are some inherent **ambiguities** in A that make it impossible to exactly recover $s^{(i)}$'s, given only the $x^{(i)}$'s:

- **Permutation Ambiguity**: ICA can effectively separate independent sources, but it is not able to understand which source corresponds to which specific original signal. This problem is due to the presence of **ambiguous permutations** in the mixing and unmixing processes.

  In fact, 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:

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad W = \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} \qquad x = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$

$$W \cdot x = \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 19 \\ 24 \end{bmatrix}$$

$$P \cdot W \cdot x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 5 \\ 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 4 & 6 \\ 3 & 5 \end{bmatrix} \cdot \begin{bmatrix} 3 \\ 2 \end{bmatrix} = \begin{bmatrix} 24 \\ 19 \end{bmatrix}$$

  This means that, **given only the $x^{(i)}$'s, there is no way to distinguish between W and $P \cdot W$.**

- **Scaling Ambiguity**: ICA cannot determine the precise scaling of sources and of A. If A is replaced by $2A$ and $s^{(i)}$ by $0.5 \cdot s^{(i)}$, the observed signal $x^{(i)}$ remains unchanged:

$$x^{(i)} = A \cdot s^{(i)}$$

$$x^{(i)} = 2A \cdot \left(0.5 \cdot s^{(i)}\right)$$

  This makes it impossible to determine **the precise scaling of the elements in the mixing matrix A or the sources $s^{(i)}$ based solely on the observed data $x^{(i)}$.**

- **Gaussianity Ambiguity**: If the source signals were Gaussian, ICA would fail to recover the correct mixing matrix.

  Consider sources $s \sim N(0, I)$, mixed by A such that $x = As$. Since linear transformations of Gaussian variables remain Gaussian, we get:

$$x \sim N(0, AA^T)$$

  Now, if we apply an **orthogonal transformation** R (where $RR^T = R^T R = I$), using a new mixing matrix $A' = AR$ still results in:

$$x' \sim N(0, AA^T)$$

Thus, **if sources were Gaussian there is no way for ICA to know if they were mixed with A or A'.**

This ambiguity arises because Gaussian distributions are **rotationally symmetric**, meaning any orthogonal transformation of a Gaussian distribution yields another Gaussian distribution with identical 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** which states that any linear combination of given random variables will have a distribution that is more Gaussian as compared to the original variables themselves.

So, if we have two independent Gaussian sources after mixing them, they become more Gaussian as central limit theorem state, which makes it impossible to recover the original sources.

ICA Algorithm: Supposing that s is a random variable drawn according to a density function $p_s(s)$ where $s = Wx$. Then the probability density of the transformed variable x, denoted as $p_x(x)$, is related to the density of s by:

$$\boldsymbol{p_x(x) = p_s(Wx) \cdot |W|}$$

where $W = A^{-1}$ and $|W|$ represents the determinant of the matrix W.

Since we assumed that the k sources $s^{(1)}, s^{(2)}, \dots, s^{(k)}$ are **independent** random variables we can express the **joint distribution** of the sources as a product of their individual densities:

$$p(s) = \prod_{j=1}^{k} p_s\left(s^{(i)}\right)$$

and as consequence we can also express $p(x)$ as:

$$p(x) = \prod_{j=1}^{k} p_s\left(w_j^T x\right) \cdot |W|$$

Now we aim to maximize $p(x)$, which means 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. 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.

In specifying the density p_s, we should avoid using the Gaussian density and we want that to be a monotonic function that increases from 0 to 1; therefore, we can think to use the sigmoid function $P(s) = sigmoid(s) = g(s)$ since we know that $sigmoid(s) \in [0,1]$ and is monotonic. The corresponding density is the derivative of the sigmoid:

$$p_s(s) = g'(s)$$

Substituting $p_s(s) = g'(s)$ into the log-likelihood function, we obtain:

$$l(W) = \sum_{i=1}^{m} \left(\sum_{j=1}^{k} \log g'(w_j^T x) + \log|W| \right)$$

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 **stochastic gradient ascent update rule**. For a given training example $x^{(i)}$, the update rule is:

$$W := W + \alpha \left(\begin{bmatrix} 1 - 2g(w_1^T x^{(i)}) \\ 1 - 2g(w_2^T x^{(i)}) \\ \vdots \\ 1 - 2g(w_k^T x^{(i)}) \end{bmatrix} x^{(i)} + (W^T)^{-1} \right)$$

where α is the learning rate.

Once the algorithm converges, we can recover the original independent sources $s^{(i)}$ by applying the learned unmixing matrix W to the observed data $x^{(i)}$:

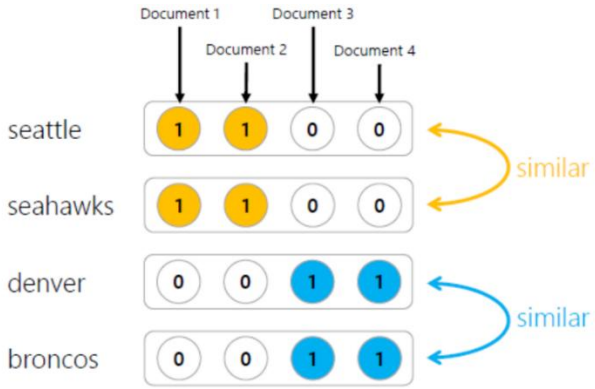
$$s^{(i)} = W x^{(i)}$$

Q: What is a Vector Space Model? List some common vectorial representations used in NLP.

A **Vector Space Model (VSM)** in the context of NLP is a mathematical representation that transforms text data into numerical vectors. This allows machine learning algorithms to process and analyze textual data effectively.

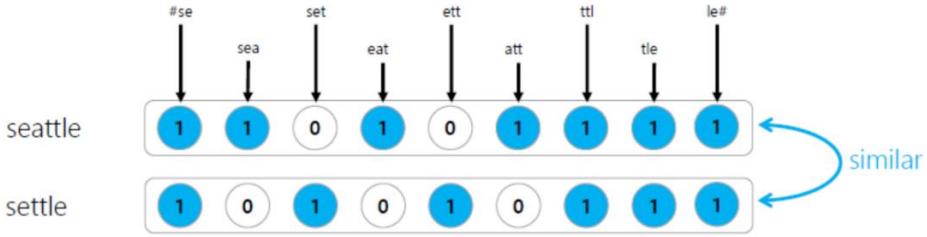
There are several ways to represent words as vectors:

1. **Document-based representation:** The vector can correspond to documents in which the word occurs. This helps capture **topical similarity**—words appearing in the same documents are considered related.



2. **Context-based representation:** The vector can correspond to neighboring word context. For example, if "seattle" and "denver" frequently appear in similar contexts, they are considered similar. This method captures **typical (by-type) similarity** between words.

3. **Character-based representation (trigrams):** The vector can correspond to character trigrams in the word. 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.



Once words are transformed into vectors, their similarity can be measured using techniques like **cosine similarity**, which calculates the angle between two word vectors. A smaller angle indicates higher similarity. However, the interpretation of this similarity depends on the chosen vector representation—topical, contextual, or character-based.

Q: What are word embeddings in NLP, and how are they learned?

Word embeddings are **dense**, low-dimensional vector representations of words that capture semantic relationships and contextual similarities. Unlike traditional high-dimensional, **sparse** vectors (meaning that most of their components are zeros), word embeddings efficiently encode word associations in lower-dimensional representations while preserving their meaningful relationships.

The process of generating word-embeddings generally involves word vectorization combined with dimensionality reduction.

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. Some algorithms that follow this approach are **LDA (Latent Dirichlet Allocation)** and **GloVe (Global Vectors for Word Representation)**.

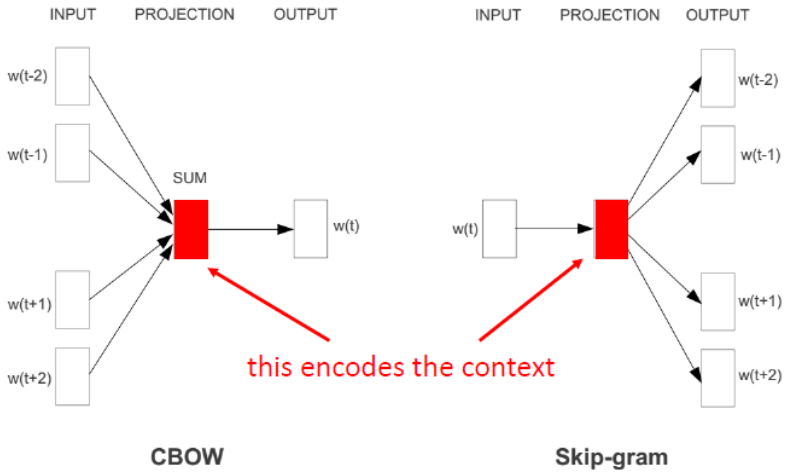
2. **Neural Networks:** These models learn embedding by mapping words to their contexts through an intermediate "bottleneck" layer, which compresses the information into dense vectors. A famous algorithm that follows this approach is **Word2Vec** which presents two architectures:
    - **Skip-Gram:** Predicts surrounding words given a target word.
    - **CBOW (Continuous Bag of Words):** Predicts a target word based on its surrounding words.

Q: What is Word2Vec and for what is used for?

Word2Vec is a set of models to transform words from a large text corpus into **word embeddings**. The key idea behind Word2Vec is that words appearing in similar contexts should have similar vector representations, therefore, to train the models makes use of the **context** of a word—the words surrounding it—to provide its meaning.

Word2Vec leverages two neural network architectures:

1. **Continuous Bag of Words (CBOW):** Predicts a target word based on its surrounding context words.
2. **Skip-Gram (SG):** Predicts the surrounding words given a target word.



Interestingly, Word2Vec models consist of just a two-layer neural network, but the embeddings learned via them are used as inputs for more complex DL models.

Q: What is the Continuous Bag of Words and how does it work?

The **Continuous Bag of Words (CBOW)** model is one of the two architectures used in Word2Vec to learn word embeddings. It predicts a target (middle) word based on its surrounding context words.

The model architecture consists of a two layers neural network:

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 are one-hot vectors of the context words.

The model maintains 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.

The key steps in CBOW are:

1. **Tokenization & Vocabulary Creation:**
    - The text corpus is split into individual words (tokens).
    - A vocabulary is created by extracting unique tokens.
    - Each word is mapped to a unique index and represented as a one-hot vector.
2. **Context-Target Pairs Formation:**
    - A window size n is defined to determine the number of surrounding words to consider. For example, if $n = 2$, for the target word "sat" the context words in *"The cat sat on the floor"* would be ["the", "cat", "on", "the"].
    - Context-target pairs are formed by converting both the context words and the target word into one-hot vectors. The context words one-hot vectors x_i and the target word one-hot vector y are each of size $V \times 1$.
3. **Embeddings and Prediction:**
    - The context words one-hot vectors are passed in input to the network:
        - **Hidden layer:** Transforms these context one-hot vectors into dense word embeddings using an 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 embeddings of the context words are averaged or summed to form a N-dimensional **aggregated** context embedding $\hat{v}$. For instance, performing the average we have:

          $$\hat{v} = \frac{1}{2n}\sum_{i=1}^{2n} v_i$$

        - **Output layer:** Uses another matrix W' to predict the target word. Specifically,
            - the aggregated context embedding $\hat{v}$ in output from the hidden layer is multiplied by W', resulting in a V-dimensional vector z representing the raw scores (**logits**) for each word in the vocabulary:

              $$z = W'^T \times \hat{v}$$

            - A **softmax function** converts these scores into a probability distribution:

              $$\hat{y} = softmax(z)$$

              The word with the highest probability is the model's predicted target word.
4. **Training with Backpropagation:**
    - The model optimizes the word embeddings by minimizing the **cross-entropy loss** between the predicted and actual target words:

      $$L = -\sum_{i=1}^{m} y_i \log(\hat{y}_i)$$

This loss is minimized using optimization techniques like Stochastic Gradient Descent (SGD).

- 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.

Q: What is the purpose of an autoencoder, and how does it work?

An **autoencoder** is a type of artificial neural network used for **dimensionality reduction**. Its goal is to learn a lower-dimensional **representation** of input data while preserving its most important information.

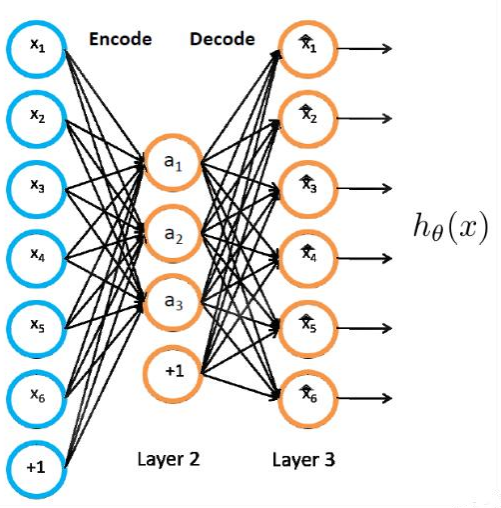
It consists of two main components:

1. **Encoder:** Reduce the input into a lower-dimensional representations also known as **latent representation**.

$$z = s(Wx + b)$$

2. **Decoder:** Reconstructs the original input from the latent representation

$$\tilde{x} = s(W'z + b')$$



The network is trained to make the output as close as possible to the input:

$$h_\theta(x) \approx x$$

However, this leads to a **trivial solution** (simply learn the identity mapping) unless **constraints** are applied:

- **Limiting the size of the latent space** forces meaningful compression (**standard autoencoder**).
- **Adding sparsity (sparse autoencoder):** Uses **L1 regularization** to ensure that only a few neurons are active at a time, forcing a more sparse representation which helps identify the most relevant features of the data.

The network is trained to minimize the **reconstruction error**, ensuring that the decoded output $\tilde{x}$ is as close as possible to the original input x:

$$\min_\theta \|\tilde{x} - x\|^2$$

or, in case of a sparse autoencoder, an $l1$ regularization penalty is added to the loss:

$$\min_{\theta} \|\tilde{x} - x\|^2 + \lambda \sum_i |a_i|$$

where a_i represents the activation of the i-th neuron and λ is the regularization parameter.

Q: What is layer-wise training and why was it used to train deep autoencoders?

Layer-wise training is a **pretraining technique** used in deep autoencoders to overcome training difficulties, such as the **vanishing gradient problem**. Vanish gradient is a phenomenon that 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.

Layerwise training works in the following manner:

1. Initially, a network with a **single hidden layer** is trained. 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. Optionally, after all hidden layers are trained, the network can be **fine-tuned globally**.

With this approach, **earlier layers learn meaningful representations before deeper layers are added** and as a result, weight updates are more stable suffering less from vanish gradient problem.

While it was essential in early deep autoencoders, modern advancements (e.g., **ReLU activations, Xavier initialization, and better optimizers**) have made **end-to-end training** the standard, eliminating the need for layer-wise pretraining.

Q: How do autoencoders compare to PCA in dimensionality reduction?

Kernel PCA applies a **predefined kernel function** to be able to capture nonlinearity in the data. Autoencoders , on the other hand, use **nonlinear activation functions** which make it possible to learn the nonlinearity in the data **adaptively** without requiring an explicit kernel function. This makes autoencoders **more flexible and effective** in capturing intricate data structures, allowing them to outperform Kernel PCA in many cases.

Q: What the Occam's Razor and the No Free Lunch (NFL) Theorem state?

These are principles that serve as guiding frameworks in machine learning.

Occam's razor is a guiding principle in machine learning for selecting the best algorithm among various options. It suggests that when choosing between models, the simplest one is usually the best. This principle encourages selecting models that balance accuracy with simplicity, reducing unnecessary complexity.

On the other hand, the **NFL Theorem** states that if an algorithm excels in a particular class of problems, it will necessarily perform worse on others. In essence, no algorithm is universally superior → No algorithm can give good results on all the problems. You should not generalize good results obtained on some problems to other problems.