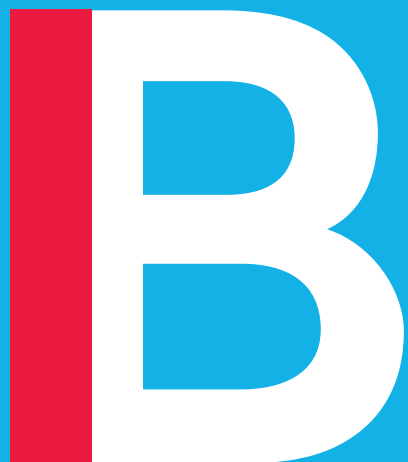


Machine Learning in Finance

Lecture 2 Introduction to Supervised Learning



Arnaud de Servigny & Jeremy Chichportich

Outline

- Introduction to the principles of Supervised Learning
- Linear Regression
- Logistic Regression
- Trees and Random Forests
- From Bagging to Boosting

Introduction to the principles of Supervised Learning

Categorisation of Supervised models

- In some instances, all explanatory features are considered on the same footing. This is typically the case with regressions, and related ones such as Logit (Parametric models).
- Alternatively, the explanatory features may be ordered in a successive manner in order to refine the selection effort (non Parametric models).
- In the first instance, fitting a model means finding the optimal weights applied to each feature. In the second instance finding the optimal model means ordering the most relevant features and finding the best cut-offs at each step.

Supervised Learning – Parametric Models

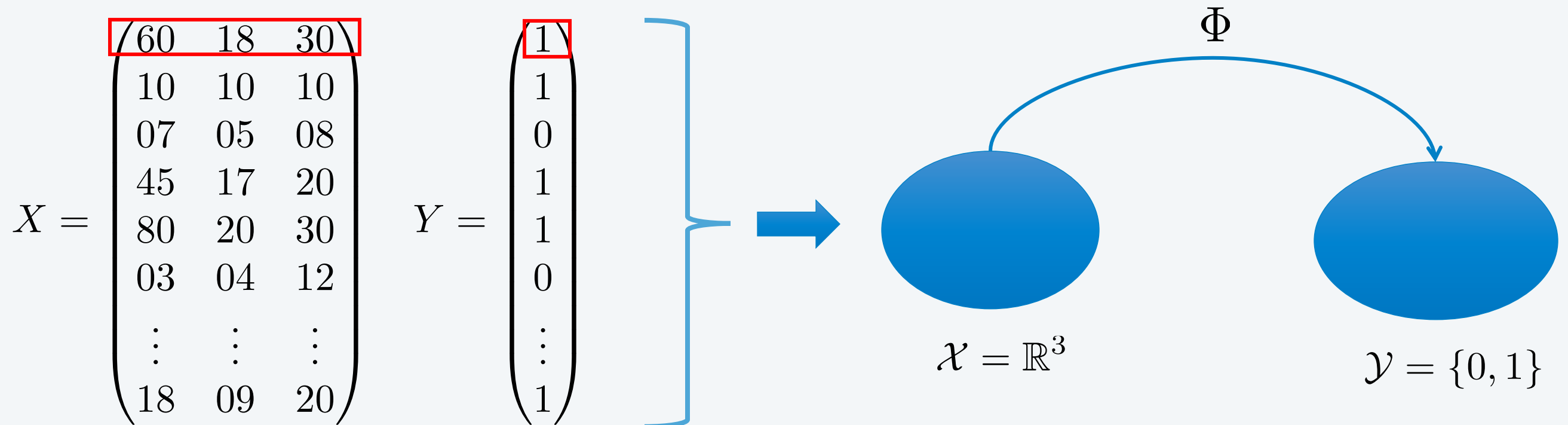
Setting up the Objective

- Supervised Learning is the process of learning a function which maps input data to an output based on several input-output pairs. Let's detail the process:
 - First, we have a dataset of pairs $\{features, target\} = \{(X_i, Y_i)_{1 \leq i \leq n}\}$ over $\mathcal{X} \times \mathcal{Y}$
 - Typically : $\mathcal{X} = \mathbb{R}^D$ and $\mathcal{Y} = \{0, 1\}$.
 - The pairs $\{(X_i, Y_i)_{1 \leq i \leq n}\}$ are assumed to be independent and identically distributed (i.i.d.) following an unknown distribution. It is important to mention here that we assume no sequentiality in the data.
- Example:
 - Let's consider this small dataset: We try to predict whether a student will fail or pass the final exam based on some feature values.
 - $Y_i = 1$ if the student pass, $Y_i = 0$ if he fails.
 - For each X_i , the first coordinate represents the number of hours spent on the course, the second coordinate is the average intermediary quiz mark and the third coordinate is the number of hours spent on the coursework.

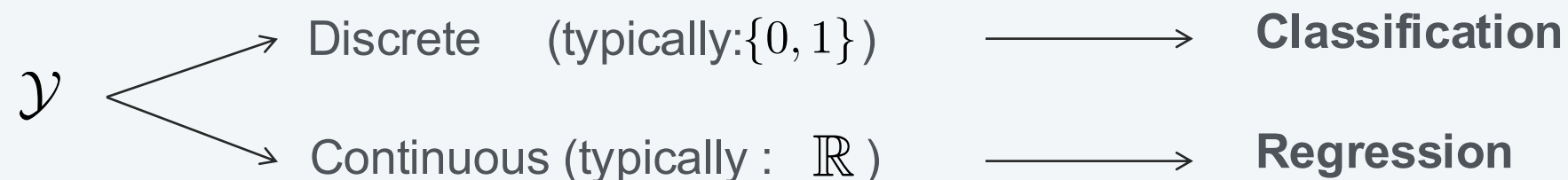
$$\begin{array}{lcl} X_1 = [60, 18, 30] & \longrightarrow & Y_1 = 1 \\ X_2 = [10, 10, 10] & \longrightarrow & Y_2 = 1 \\ X_3 = [07, 05, 08] & \longrightarrow & Y_3 = 0 \end{array}$$

Setting up the Objective

- A Supervised Algorithm is an algorithm that aims at building a predictor (i.e, a function $\Phi : \mathcal{X} \longrightarrow \mathcal{Y}$) which minimizes an error, based on the dataset.



- In the previous example, our objective was to predict a **discrete** value : pass or fail (1 or 0). This supervised task is called **classification**.
- We can also try to predict a **continuous** value: the final exam mark for instance. In that case, the task is called **regression**.



Setting up the Objective

- To define the error, we need first to define a **loss function** (i.e, a function $l : \mathcal{Y} \times \mathcal{Y} \longrightarrow \mathbb{R}$ which measures the "distance" between the the output of the predictor and the true labels $(Y_i)_{1 \leq i \leq n}$).
- For the loss function, we usually choose for all pairs (*ouptut, true label*), $(y, y') \in \mathcal{Y} \times \mathcal{Y}$:

$$l_{\text{Regression}}(y, y') = ||y - y'||_2^2 \quad \text{and} \quad l_{\text{Classification}}(y, y') = \delta_{y \neq y'} = \begin{cases} 1 & \text{if } y \neq y' \\ 0 & \text{if } y = y' \end{cases}$$

- We then define the following error / risk associated to the predictor Φ , the aggregate loss over the train set, which follows an unknown distribution \mathbb{P} :

$$\mathcal{R}_{\mathbb{P}}(\Phi) = \mathbb{E}^{\mathbb{P}}[l(Y, \Phi(X))]$$

- Our objective is to find the optimal predictor among all the possible functions defined by the modeler (e.g. Logit, linear regression, etc) $\mathcal{F}(\mathcal{X}, \mathcal{Y})$:

$$\text{Finding } \phi_{\mathbb{P}}^* = \underset{\Phi \in \mathcal{F}(\mathcal{X}, \mathcal{Y})}{\operatorname{argmin}} \mathbb{E}^{\mathbb{P}}[l(Y, \Phi(X))] = \underset{\Phi \in \mathcal{F}(\mathcal{X}, \mathcal{Y})}{\operatorname{argmin}} \mathcal{R}_{\mathbb{P}}(\Phi)$$

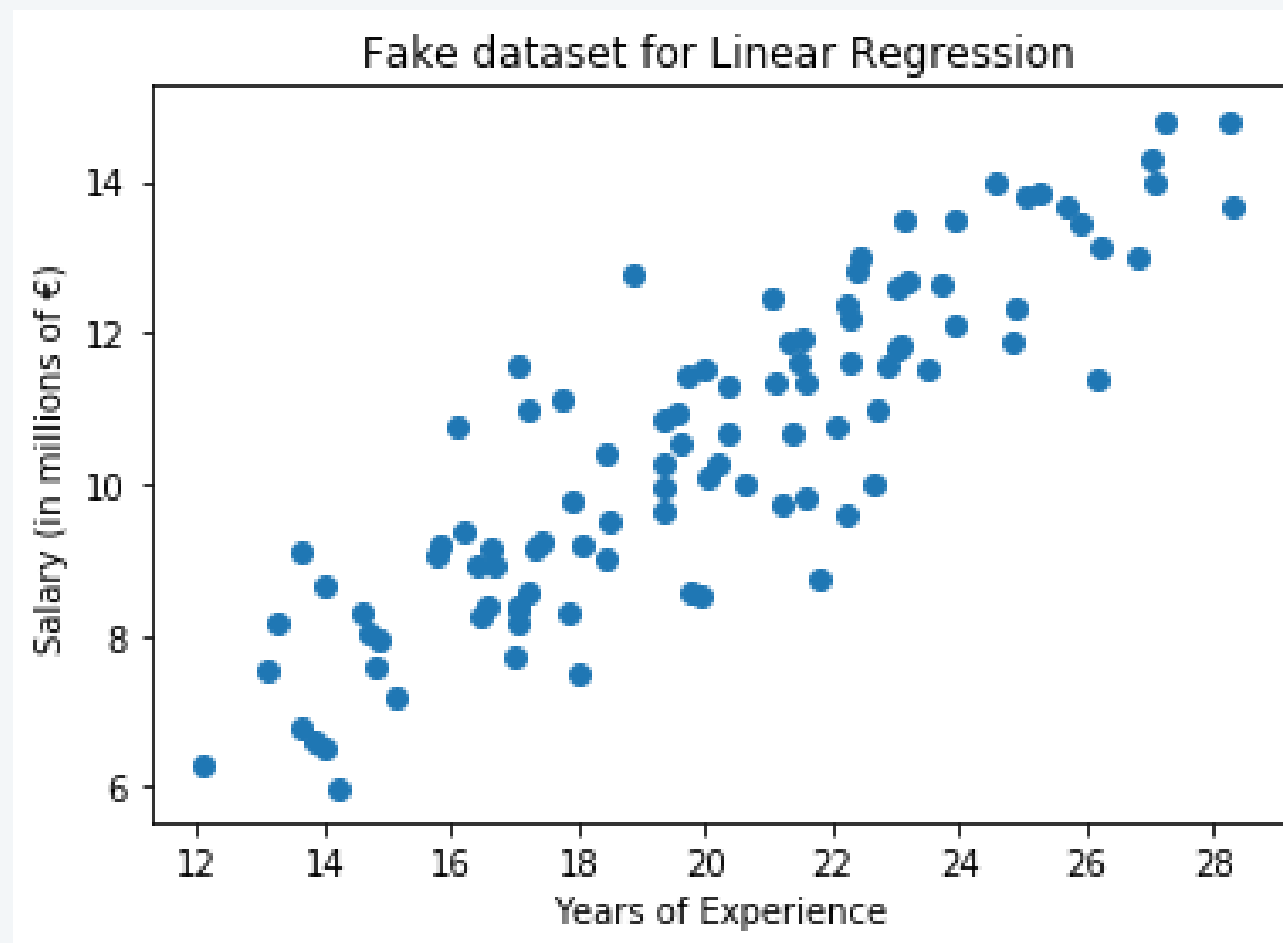
- Since \mathbb{P} is unknown, we optimize (minimize) the empirical cumulative risk $\mathcal{R}_n(\Phi)$:

$$\mathcal{R}_n(\Phi) = \frac{1}{n} \sum_{i=1}^n l(Y_i, \Phi(X_i))$$

Linear Regression

Introduction:

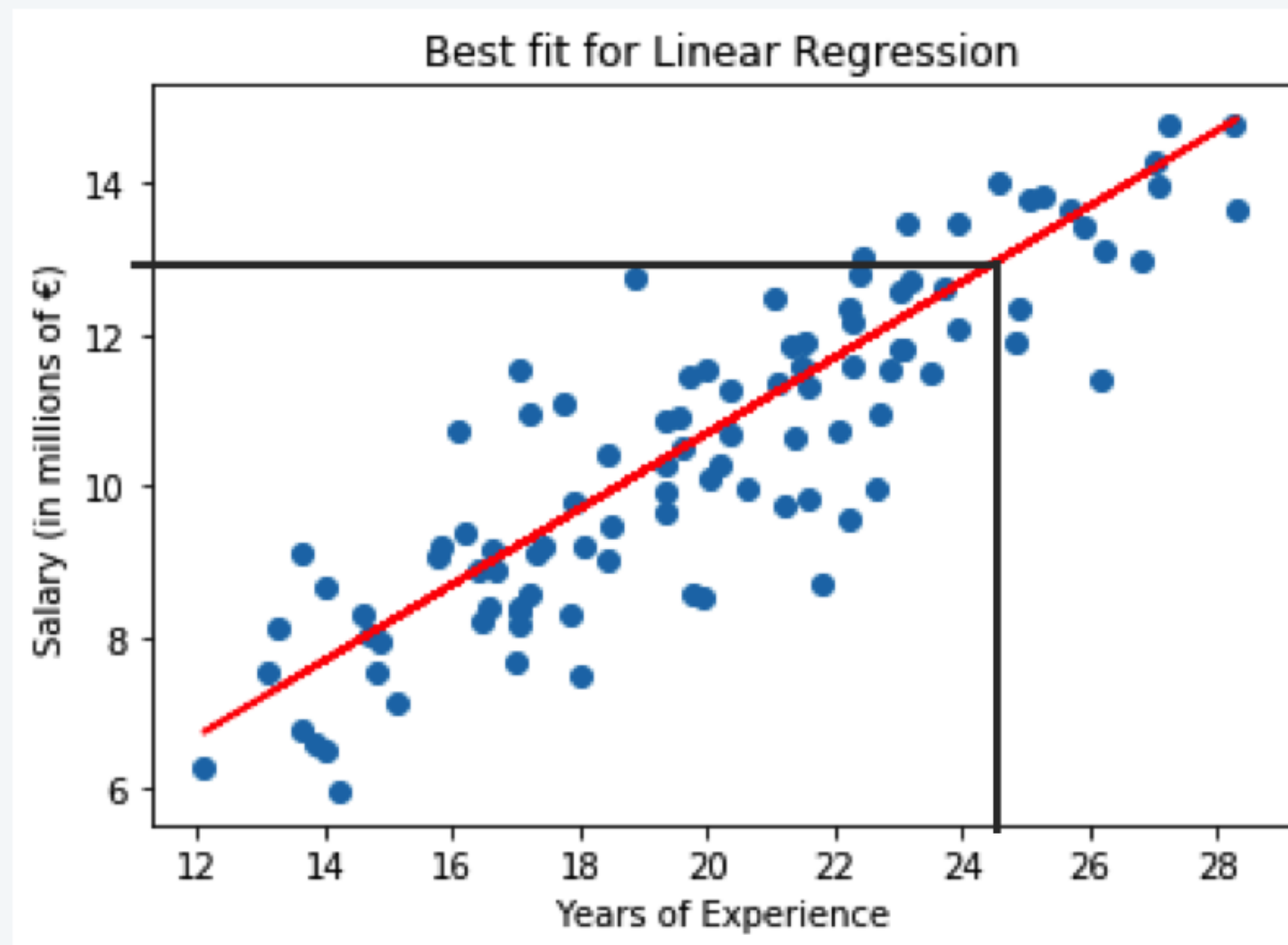
- Let us start with the simplest regression model : **Linear Regression**.
- Consider the following (fake) dataset representing the salary (in the x axis) of some (fake) employees according to the number of years of experience (in the y axis).



- From the plot, we can see that the salary and the experience variables exhibit a clear linear dependence.

Setting up the Objective:

- We would like to find a way to define the red line from the pairs (experience, salary) represented by the blue points.
- In that way, we could assign an estimated salary to each value of the experience variable.



Linear Regression: a Mathematical Perspective

- Mathematically speaking, it consists in modeling the conditional distribution of $Y_i \in \mathbb{R}$ given $X_i = x_i \in \mathbb{R}^D$, parametrized by the set of parameters $\theta = (w, b)$, as follows:

$$\forall i \in \{1, \dots, N\} \quad Y_i | X_i = x_i \sim \mathcal{N}(w^T x_i + b, \sigma^2) \quad \text{with} \quad w \in \mathbb{R}^D \quad \text{and} \quad b \in \mathbb{R}$$

- In other words,

$$\forall i \in \{1, \dots, N\} \quad Y_i = w^T X_i + b + \epsilon \quad \text{with} \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

- Now that the model is described, we will have to maximize the likelihood of the dataset in order to obtain the optimal parameters.

The training process :

- Let $\{(X_i, Y_i)_{1 \leq i \leq N}\}$ be the dataset used for training.

- The model:

$$\forall i \in \{1, \dots, N\} \quad Y_i = w^T X_i + b + \epsilon \quad \text{with} \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

- The parameter b can be ignored thanks to a normalisation of the training set :

$$\forall i \in \{1, \dots, N\} \quad Y_i = \hat{w}^T \hat{X}_i + \epsilon \quad \text{with} \quad \hat{X}_i = \begin{pmatrix} X_i \\ 1 \end{pmatrix}, \hat{w} \in \mathbb{R}^{D+1} \quad \text{and} \quad \epsilon \sim \mathcal{N}(0, \sigma^2)$$

- The training process consists in maximizing the log-likelihood w.r.t the parameters $\hat{w} \in \mathbb{R}^{D+1}$

- Since the dataset is i.i.d, the likelihood of the training set can be expressed as follows:

$$L(\hat{w}) = \prod_{i=1}^N p(y_i | x_i; \hat{w}) = \prod_{i=1}^N \left(\frac{1}{\sigma \sqrt{2\pi}} \exp\left(-\frac{1}{2} \frac{(y_i - \hat{w}^T x_i)^2}{\sigma^2}\right) \right)$$

The training process :

- In optimization matters, we usually prefer to minimize functions instead of maximizing them.
- Thus, we transform the **likelihood maximization problem** into the equivalent **cost minimization problem**, where the cost is the following **negative log-likelihood**:

$$-\log(L(\hat{w})) = -\sum_{i=1}^N \log(p(y_i|x_i)) = \frac{N}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \sum_{i=1}^N (y_i - \hat{w}^T x_i)^2$$

- The **training problem** can then be written as the following equivalent minimization problem:

$$\min_{\hat{w} \in \mathbb{R}^{D+1}} \underbrace{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{w}^T x_i)^2}_{J(\hat{w})}$$

Matrix Notation and Optimization :

- It is computationally more efficient to write the optimization problem in matrix notation.
- To that end, let's introduce the following matrix notation for the training data $\{(X_i, Y_i)_{1 \leq i \leq N}\}$

$$\hat{X} = \begin{bmatrix} - & x_1 & - & 1 \\ - & x_2 & - & 1 \\ \vdots & \vdots & \vdots & \vdots \\ - & x_N & - & 1 \end{bmatrix} \in \mathbb{R}^{N \times (D+1)} \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \in \mathbb{R}^N$$

- Then, we deduce the following prediction matrix P :

$$P = \hat{X} \hat{w} = \begin{bmatrix} \hat{w}^T x_1 \\ \hat{w}^T x_2 \\ \vdots \\ \hat{w}^T x_N \end{bmatrix} \in \mathbb{R}^N$$

- If we denote by $\| \cdot \|_2$ the \mathcal{L}^2 norm on \mathbb{R}^N (i.e : $\forall z \in \mathbb{R}^N \quad \|z\|_2^2 = z^T z$), we can then express the cost function J which we wish to minimize as follows :

$$J(\hat{w}) = \frac{1}{N} \| \hat{X} \hat{w} - Y \|_2^2$$

Using a Gradient Descent for Optimization

- We will use a **Gradient Descent** to find \hat{w}^* as follows:
 - Initialize randomly \hat{w}_0
 - Fix a number of iterations K and a learning rate η and repeat K times:

$$\hat{w}_{k+1} \leftarrow \hat{w}_k - \eta \nabla_{\hat{w}} J(\hat{w}_k)$$

Code for Gradient Descent using Numpy

```
max_iter = 100 # number of iterations for gradient descent
eta = 0.001 # learning rate

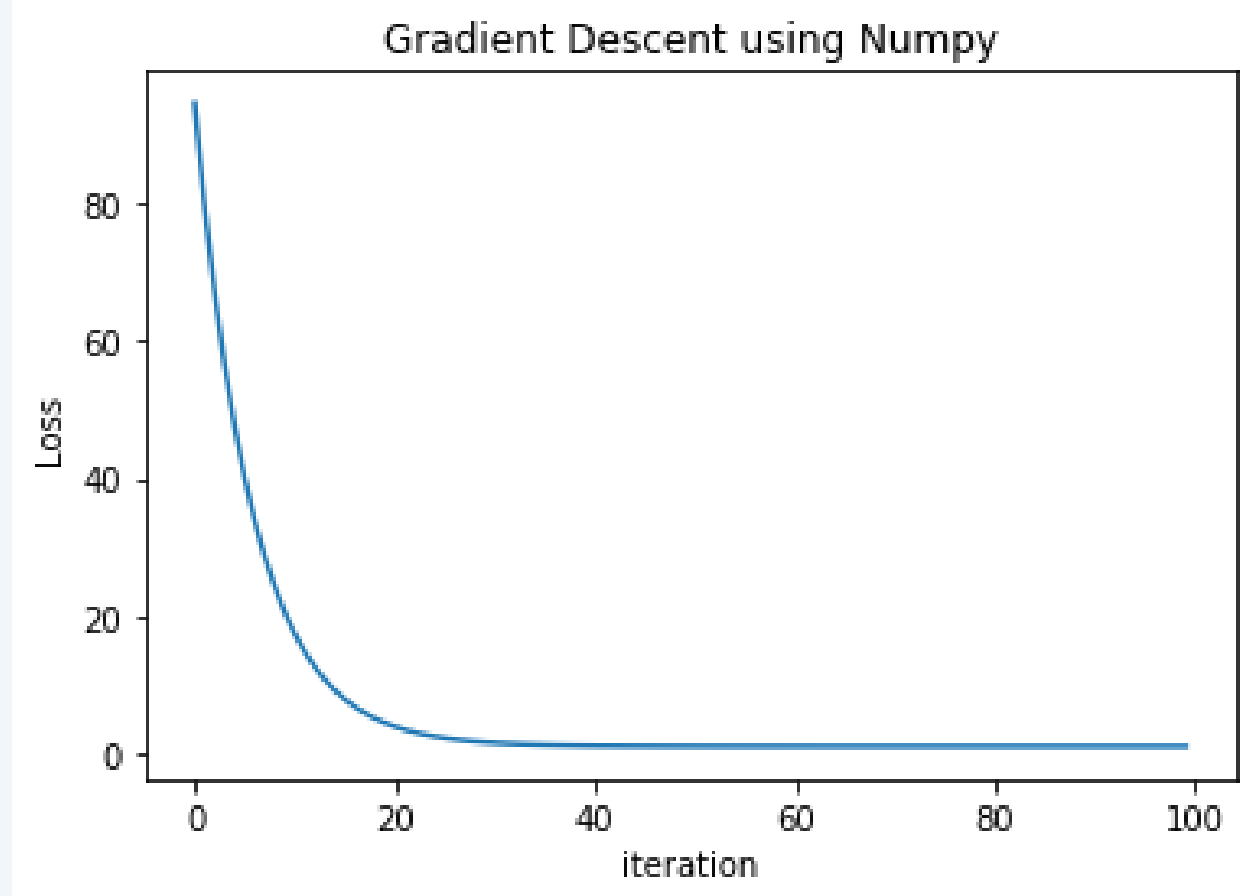
losses = []

# Gradient Descent Algorithm
W_hat = np.random.random((D+1, 1))
for i in range(max_iter):
    P = X_hat.dot(W_hat) # P = XW + b = X_hat W_hat
    loss = (1/N)*((Y - P).T).dot(Y - P) # loss = (1/N) || Y - P ||^2
    # gradient w.r.t W_hat : (2/N) (X_hat^T X_hat W_hat - X_hat^T Y)
    grad_W_hat = (2/N)*(X_hat.T.dot(X_hat.dot(W_hat)) - X_hat.T.dot(Y))

    # Gradient descent update:
    W_hat -= eta*grad_W_hat # W <- W - eta*grad_W

    # append losses
    losses.append(float(loss))

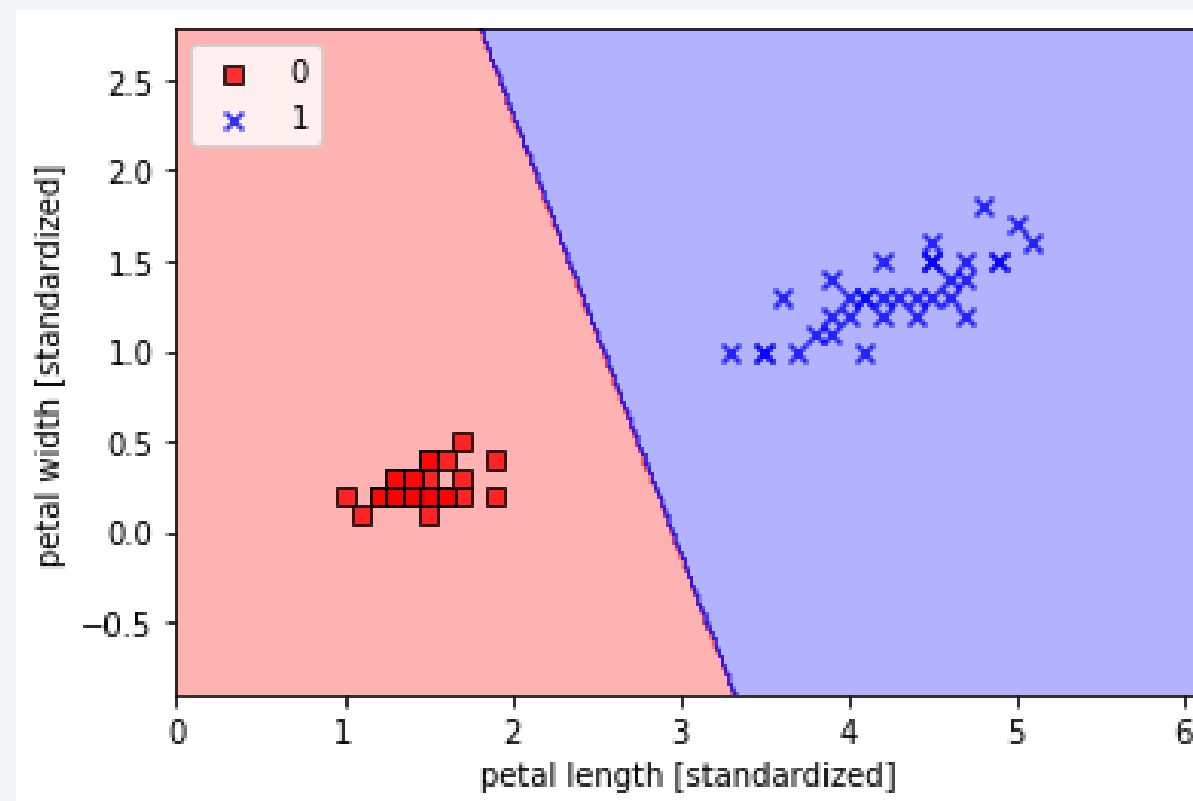
# plot losses
plt.plot(losses)
plt.xlabel("iteration")
plt.ylabel("Loss")
plt.title("Gradient Descent using Numpy")
plt.show()
```



Logistic Regression

Introduction:

- The **Logistic Regression** is one of the easiest **classification** models to implement. It also performs very well on linearly separable classes.
- We call **decision boundary** the hypersurface separating the space of input data between two subsets, one for each class. The classifier will classify all the points belonging in one side of the decision boundary as belonging in one class and all those on the other side as belonging in the other class.
- In the case of a Logistic Regression, the decision boundary is a **hyperplane**.
- The following scatterplot of the public Iris dataset shows a **linear decision boundary** associated with Logistic Regression.



Presenting the logit function :

- Before introducing the Logistic Regression as a probabilistic model, we should first introduce the **odds ratio**.

- The odds ratio (in favor of a particular event) can be written as $\frac{\theta}{1 - \theta}$ where θ stands for the probability of the positive event.

A high odds ratio means a very probable outcome.

- We can further define the **log-odds**, also called **logit function** $[0, 1] \rightarrow \mathbb{R}$, as the logarithm of the odds ratio:

$$\text{logit}(\theta) = \log\left(\frac{\theta}{1 - \theta}\right)$$

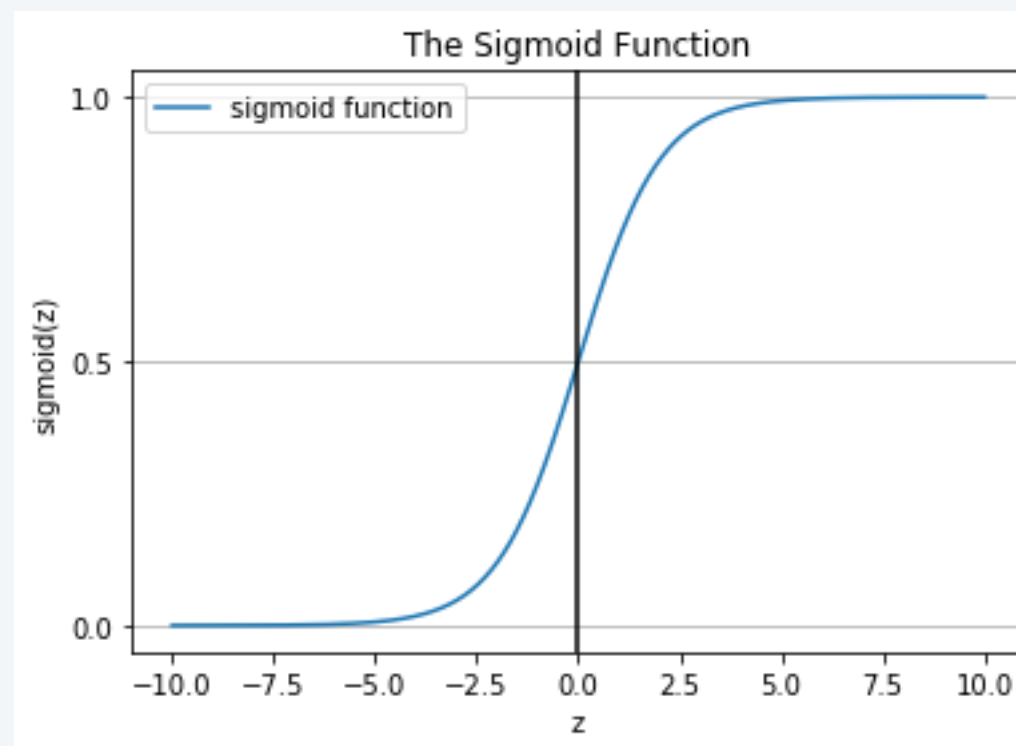
- The fact that we consider a log outcome means that poor outcomes matter more to us than positive outcomes. It is a strong preference choice. It also means that when the logit function outputs a high level, it means that it is even higher in reality.
- In the context of a Logistic Regression, we assume a linear relationship between feature values and the log-odds of the conditional probability that a particular sample belongs in the positive class given its features.

- Mathematically speaking: $\text{logit}(\mathbb{P}(Y_i = 1 \mid X_i = x_i)) = w^T x_i$

The sigmoid function and the Logistic Regression Model

- By inverting the logit function, we obtain: $\mathbb{P}(Y_i = 1 \mid X_i = x_i) = \sigma(w^T x_i)$

where σ refers to the **sigmoid function** $\sigma : z \mapsto \frac{1}{1 + e^{-z}}$



- In other words, a Logistic Regression consists in modeling the conditional distribution of $Y_i \in \{0, 1\}$ given $X_i = x_i \in \mathbb{R}^D$, parametrized by w . As the probability of the outcome Y being equal to 1 is the sigmoid function, while that of Y being equal to 0, is (1-sigmoid function), we are in the case of a Bernoulli distribution applied to the sigmoid function.

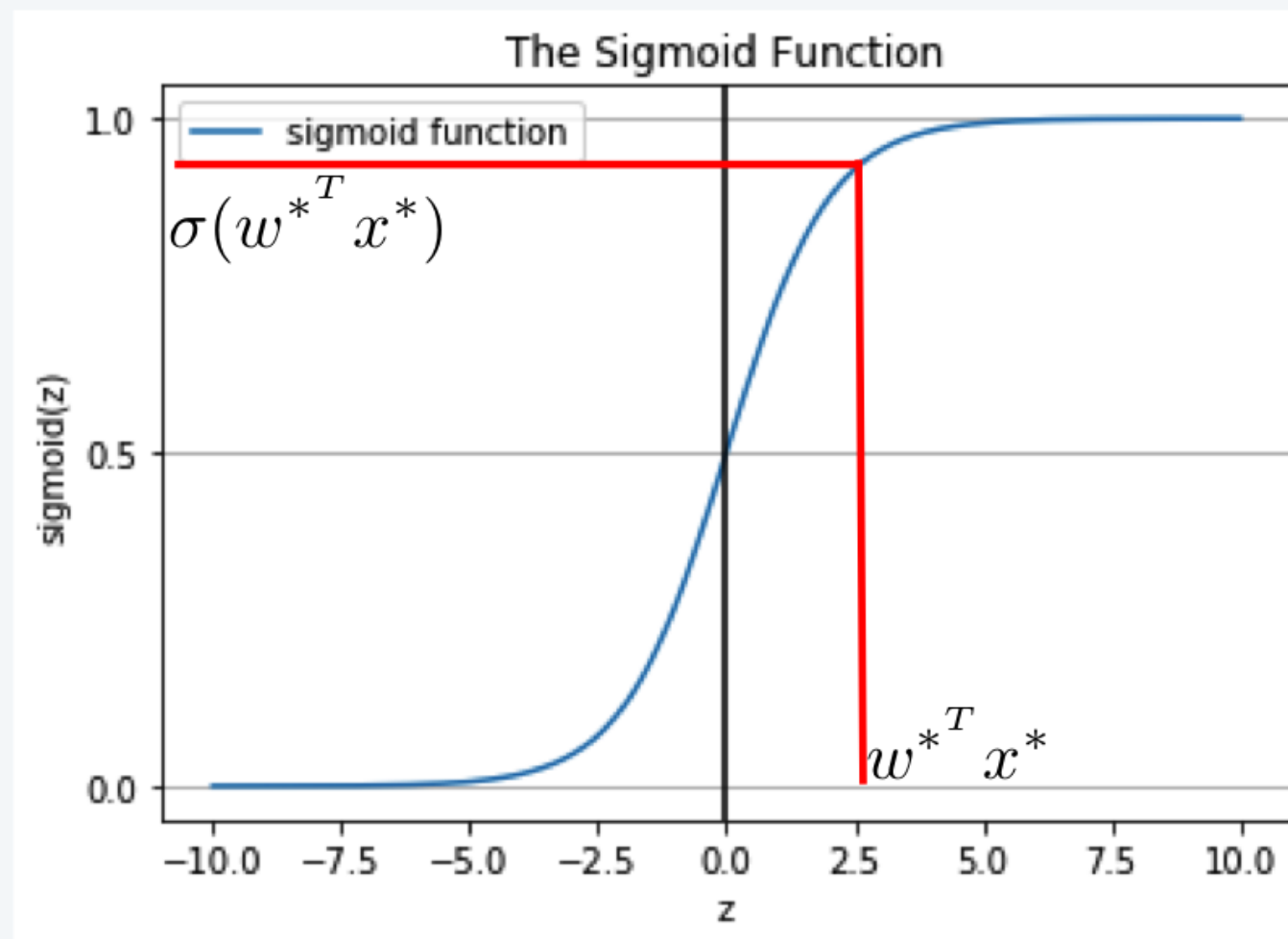
$$\forall i \in \{1, \dots, N\} \quad Y_i | X_i = x_i \sim \mathcal{B}(\sigma(w^T x_i))$$

where \mathcal{B} stands for the **Bernoulli** distribution.

The prediction phase after training:

- The training process consists in finding the parameter w^* that maximizes the likelihood of the dataset.
- After training the model, given a new data point $x^* \in \mathbb{R}^D$, we can determine the probability of being in the positive class

$$P(Y^* = 1 \mid X^* = x^*) = \sigma(w^{*T} x^*)$$



The Training Process: finding the optimal w

- Let $\{(X_i, Y_i)_{1 \leq i \leq N}\}$ be the dataset used for training.

- The model: $\forall i \in \{1, \dots, N\} \quad Y_i | X_i = x_i \sim \mathcal{B}(\sigma(w^T x_i))$

- The likelihood of the dataset can be expressed as follows:

$$L(w) = \prod_{i=1}^N p(Y_i = y_i \mid X_i = x_i; w) = \prod_{i=1}^N \sigma(w^T x_i)^{y_i} (1 - \sigma(w^T x_i))^{1-y_i}$$

- Hence, the **normalized negative log likelihood** (or **cost function**) is:

$$J(w) = -\frac{1}{N} \log(L(w)) = -\frac{1}{N} \sum_{i=1}^N (y_i \log(\sigma(w^T x_i)) + (1 - y_i) \log(1 - \sigma(w^T x_i)))$$

- The **training problem** can then be written as the following minimization problem:

$$\min_{w \in \mathbb{R}^D} \underbrace{-\frac{1}{N} \sum_{i=1}^N (y_i \log(\sigma(w^T x_i)) + (1 - y_i) \log(1 - \sigma(w^T x_i)))}_{J(w)}$$

Using a Gradient Descent for Optimization

- We will use **Gradient Descent** to find w^* as follows:
 - Initialize randomly w_0
 - Fix a number of iterations K and a learning rate η and repeat K times:

$$w_{k+1} \leftarrow w_k - \eta \nabla_w J(w_k)$$

Code for Gradient Descent using Numpy

```
max_iter = 10000 # number of iterations of gradient descent
eta = 0.001 # learning rate for gradient descent

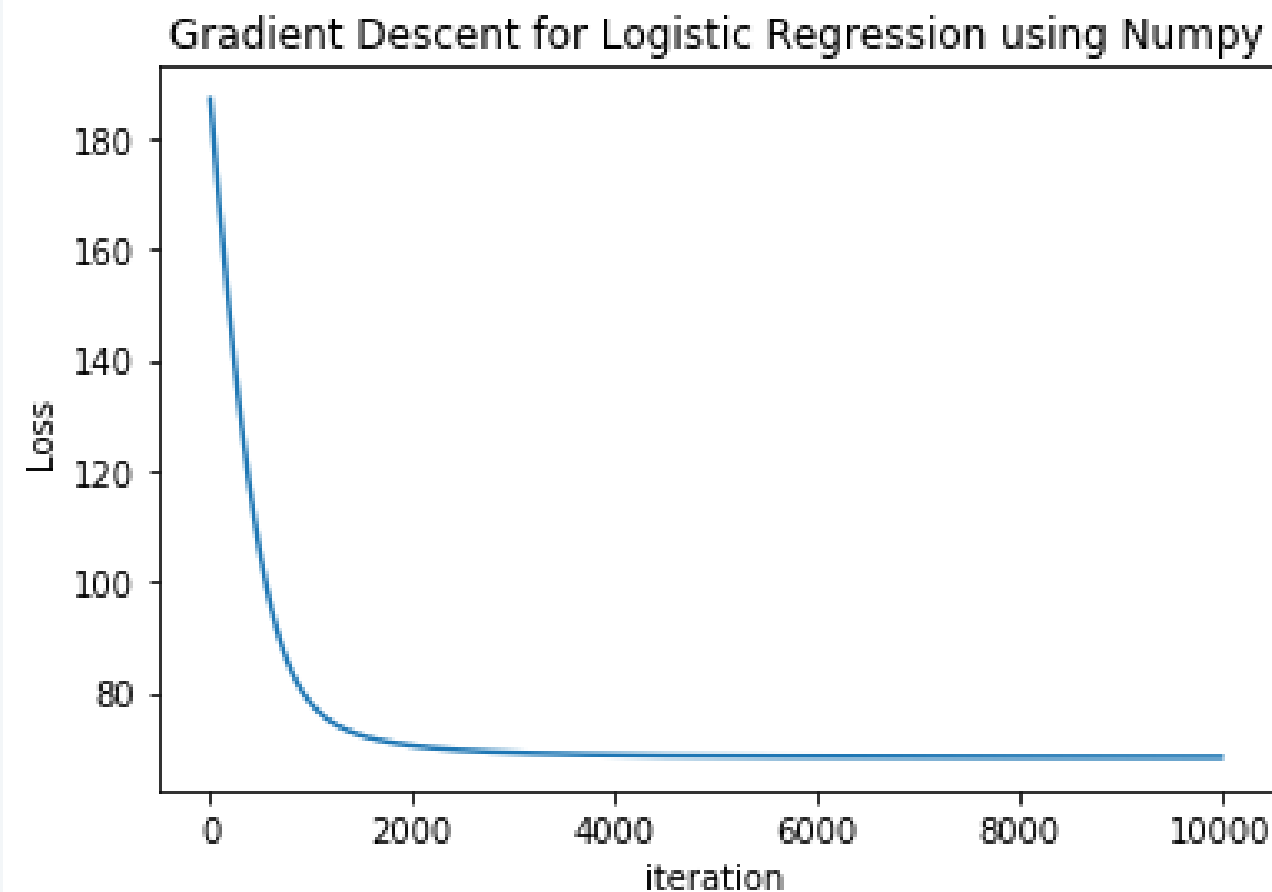
losses = []

W = np.random.randn(D)
for i in range(max_iter):
    P = expit(X.dot(W)) # P = sigmoid(XW)
    # Loss = -(1/N) (sum_i y_i log(sigmoid(w^T x_i))
    #          + (1-y_i) log(sigmoid(-w^T x_i)))
    loss = - (1/N)*np.sum((Y * np.log(P)) + ((1 - Y)*np.log(expit(-P))))
    epsilon = P - Y
    epsilon_hat = np.diag(epsilon[:, 0])
    grad_W = (1/N)* np.sum((epsilon_hat).dot(X), axis = 0) # gradient w.r.t W

    # Gradient descent update:
    W -= eta*grad_W # W <- W - eta*grad_W

    # append losses
    losses.append(float(loss))

plt.plot(losses)
plt.xlabel("iteration")
plt.ylabel("Loss")
plt.title("Gradient Descent for Logistic Regression using Numpy")
plt.show()
```



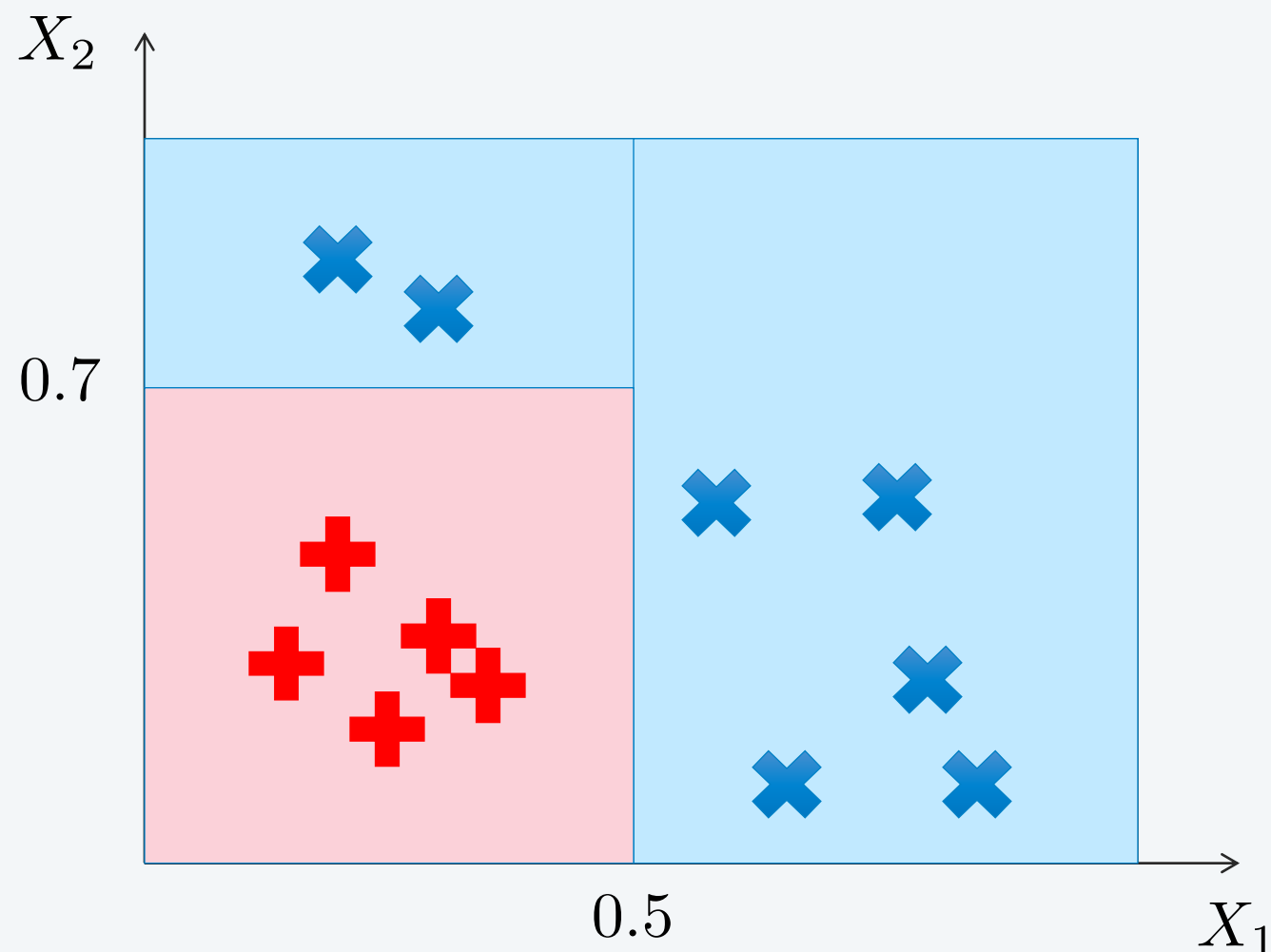
Supervised Learning – Non Parametric Models

Decision Trees and Random Forest

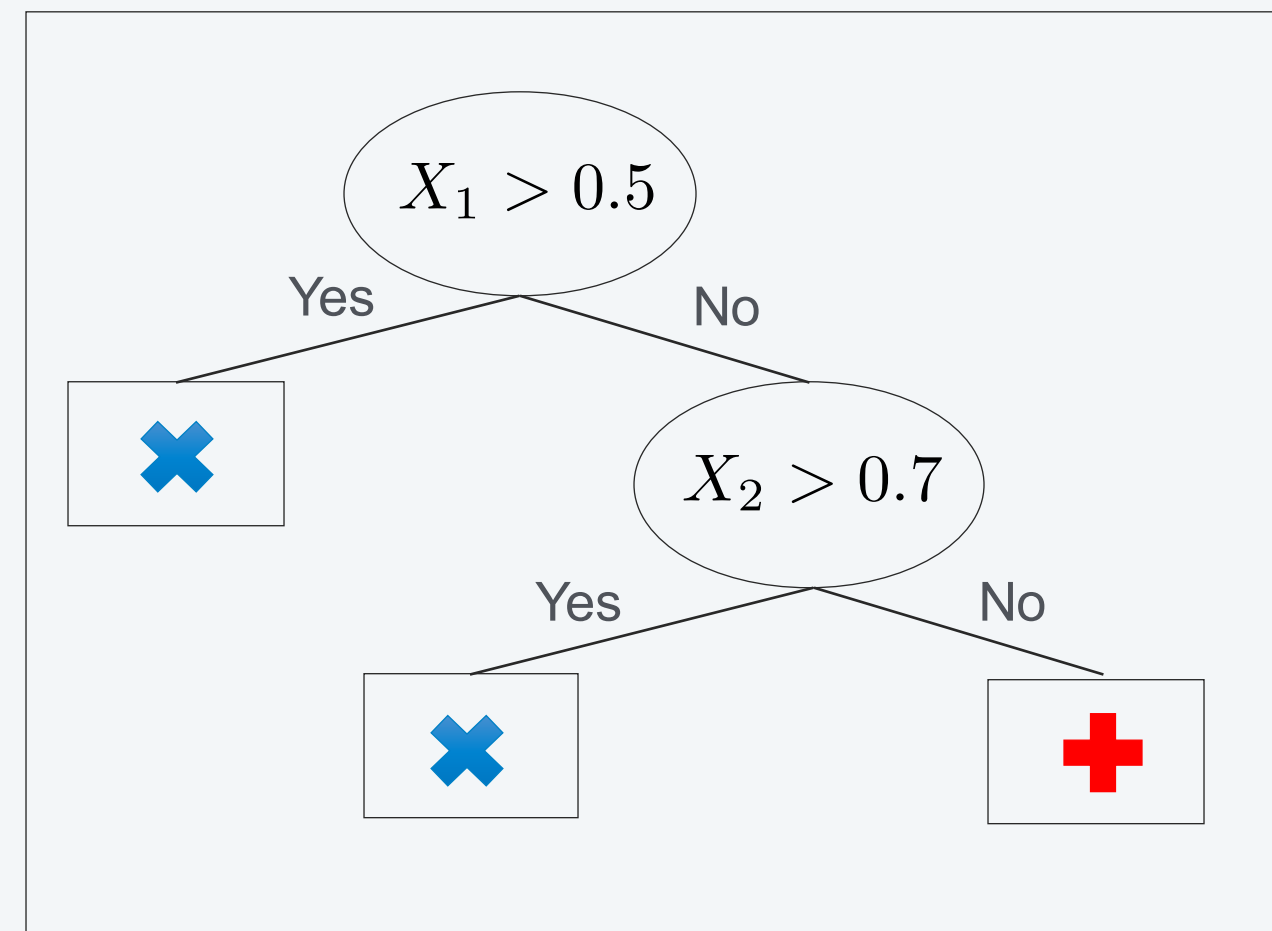
Introduction:

- The **Decision Tree** (DT) algorithm is an attractive model if we care about interpretability.
- As the name decision tree suggests, we can think of this model as breaking down our data by making a decision based on asking a series of questions.

Decision regions



Decision Tree Graph

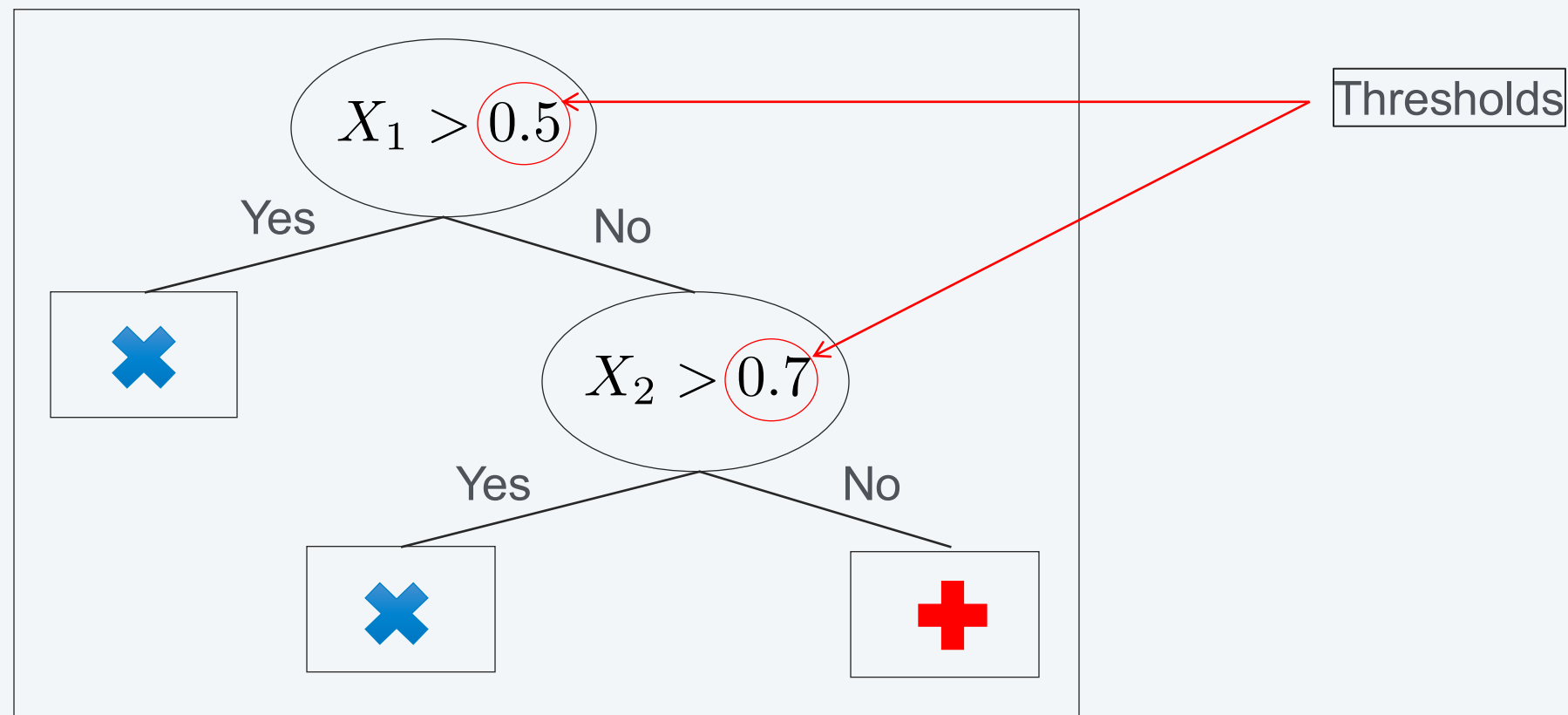


A high level description of the algorithm

- The DT algorithm is basically just a bunch of nested if-statements on the input features (also called **attributes**) in the training dataset.
- The decision algorithm:
 - We start at the tree root (with the whole dataset)
 - Then we split the dataset on the attribute that results in the largest **Information Gain** (IG).
 - We iterate the splitting procedure at each child node until the leaves are pure (which means that the samples at the leaves belong to the same class)
 - A very deep tree is prone to overfitting. To avoid that, we set a limit for the maximal depth of the tree.

Building Decision Trees

- First, we need to define an objective function that we want to optimize (Information Gain).
- Then, at each iteration, two challenges arise when trying to choose the best split.
 - How do we choose the best attribute responsible for the split ?
 - How do we choose the **threshold** when splitting based on the "best attribute" ?



Information Theory : Entropy – Part 1 –

- Let Y be a random variable taking values in the finite set \mathcal{Y}
- Let's denote $p(y) = \mathbb{P}(Y = y)$
- In information theory, the quantity $I(y) = \log_2\left(\frac{1}{p(y)}\right)$ can be interpreted as a quantity of information carried by the occurrence of y (sometimes called **self-information**).
- So, the **entropy** is defined as the expected amount of information of the random variable Y

$$H(Y) = \mathbb{E}_{p(y)}[I(Y)] = - \sum_{y \in \mathcal{Y}} p(y) \log_2(p(y))$$

Information Theory : Entropy – Part 2 –

- Let's take an example of a Bernoulli distribution $Y \sim \mathcal{B}(p)$

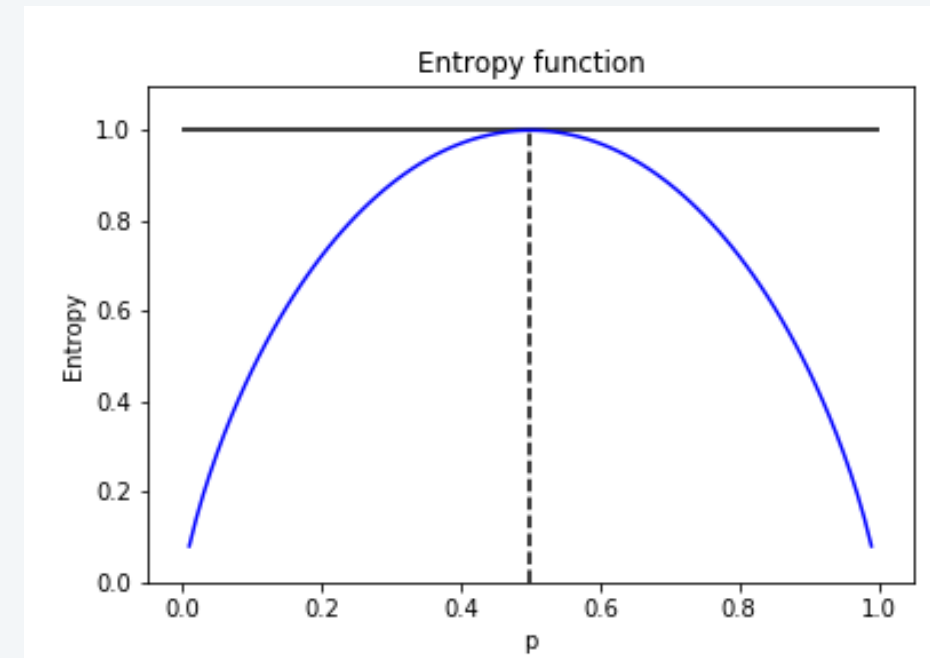
- The entropy of Y as a function of p is:

$$H(p) = -p \log_2(p) + (1 - p) \log_2(1 - p)$$

- $p = 0.5$ yields maximum entropy.
- So, the entropy is a measure of how much information we get from finding out the value of the random variable.
- We have the following inequalities:

$$H(Y) \geq 0 \quad \text{with equality if } Y \text{ is constant a.s.}$$

$$H(Y) \leq \log_2(\text{Card}(\mathcal{Y}))$$



Information Gain – Definition –

- We define the **information gain** at a split on the attribute c as follows:

$$IG(D_p, c) = I(D_p) - \frac{N_{\text{left}}}{N_p} I(D_{\text{left}}) - \frac{N_{\text{right}}}{N_p} I(D_{\text{right}})$$

- D_p refers to the dataset of the parent.
- D_{left} and D_{right} are the datasets of the left and right child nodes. (For simplicity, most libraries only implement binary decision trees).
- I is the **impurity measure**.
- N_p is the total number of samples at the parent node.
- N_{left} and N_{right} are the total number of samples at the right and left child nodes.

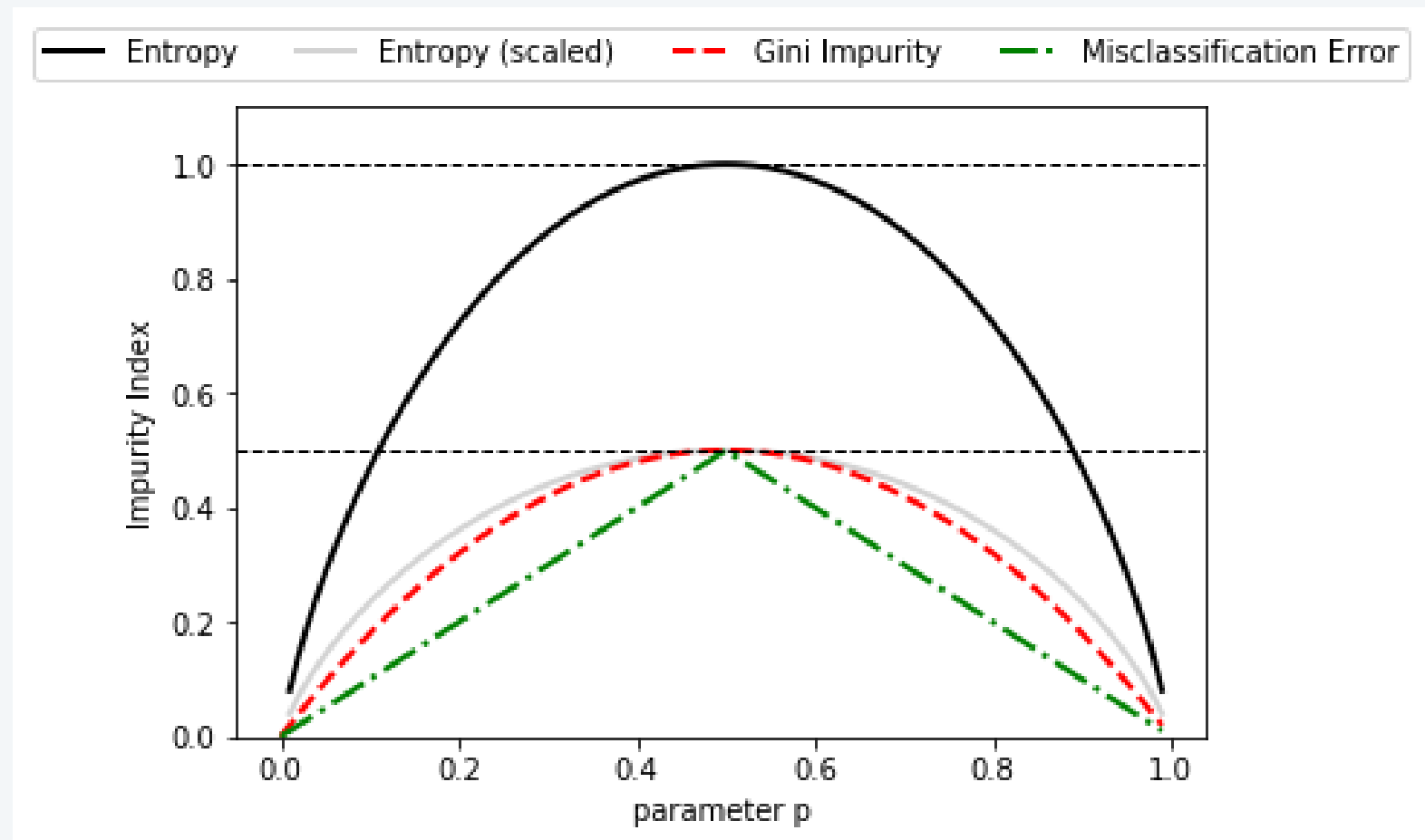
Different impurity measures

- The three impurity measures or splitting criteria that are commonly used in binary decision trees are **Gini Impurity** (I_G), **Entropy** (I_H), and the **Classification Error** (I_E).
- Let's denote p_k^m the proportion of the samples that belong to the class $m \in \{1, \dots, M\}$ for a particular node k . We define the above stated impurities as follows:

$$I_G(k) = - \sum_{m=1}^M p_k^m (1 - p_k^m)$$

$$I_H(k) = - \sum_{m=1}^M p_k^m \log_2(p_k^m)$$

$$I_E(k) = 1 - \max_{1 \leq m \leq M} p_k^m$$



A simple example – Description –

- Let's consider the following example:
 - We want to predict whether a person is sick or not ($Y = 1$ if it's the case and $Y = 0$ if not) based on the following 4 features:
 - $X_1 \in \{0, 1\}$ is whether the person was in contact with someone or not during the last month.
($X_1 = 1$ if it's the case and $X_1 = 0$ if not).
 - $X_2 \in [0, 100]$ is the number of times the person has touched their face per day.
 - $X_3 \in [0, 10]$ is the number of times the person has washed their hands per day.
 - $X_4 \in \{0, 1\}$ is whether the person is a man ($X_4 = 0$) or a woman ($X_4 = 1$).

A simple example – Dataset –

- We end up with the following small dataset of 10 samples.

In contact with a person	Number of touching face	Number of washing hands	Man/woman	Y
0	98	0	0	0
0	87	1	0	0
1	93	9	0	1
1	97	4	1	1
1	81	7	0	1
1	42	8	1	0
1	28	3	0	0
1	12	1	0	1
1	8	0	0	1
1	2	9	0	0

- Let's use the entropy as the impurity measure.

- We have at the root:
$$I(D_p) = H(Y) = -\frac{1}{2} \log_2\left(\frac{1}{2}\right) - \frac{1}{2} \log_2\left(\frac{1}{2}\right) = 1$$

A simple example – split on « man vs woman » column –

- Let's split on the attribute « man/woman » (X_4)

$X_4 = 0$

X_1	X_2	X_3	X_4	Y
0	98	0	0	0
0	87	1	0	0
1	93	9	0	1
1	81	7	0	1
1	28	3	0	0
1	12	1	0	1
1	8	0	0	1
1	2	9	0	0

$$I(D_{\text{left}}) = H(Y|X_4 = 0) = 1$$

$X_4 = 1$

X_1	X_2	X_3	X_4	Y
1	97	4	1	1
1	42	8	1	0

$$I(D_{\text{right}}) = H(Y|X_4 = 1) = 1$$

A simple example – split on « man vs woman » column –

- If we split on the attribute « man/woman » (X_4), we obtain the following information gain :

$$\begin{aligned} IG(D_p, X_4) &= I(D_p) - \frac{N_{\text{left}}}{N_p} I(D_{\text{left}}) - \frac{N_{\text{right}}}{N_p} I(D_{\text{right}}) \\ &= 1 - \frac{8}{10} \times 1 - \frac{2}{10} \times 1 \\ &= 0 \end{aligned}$$

- We conclude that there is absolutely nothing to gain from splitting on the « man/woman » attribute. Men and women seem to have the same likelihood to be infected by the disease.

A simple example – split on «in contact with a person » column –

- Let's split on the attribute «in contact with a person » (X_1)
- We obtain then the following information gain :

$$IG(D_p, X_1) = 1 - \frac{2}{10} \times 0 - \frac{8}{10} \times 0.95 = 0.24$$

X_1	X_2	X_3	X_4	Y
0	98	0	0	0
0	87	1	0	0
1	93	9	0	1
1	97	4	1	1
1	81	7	0	1
1	42	8	1	0
1	28	3	0	0
1	12	1	0	1
1	8	0	0	1
1	2	9	0	0

Finding the best split for a particular column

- In the last example, we made it easy : « man/woman » and «in contact with a person » columns have only two possible values each.
- For continuous data, we can find some rules that lead to a smaller set of possible values.
- So to find the best split for a continuous attribute X :
 - We sort the values of the attribute in order, and sort the target Y in the corresponding way.
 - We find all the boundary points (i.e, where Y changes from one value to an other).
 - We calculate the information gain when splitting at each boundary
 - We keep the split which gives the highest information gain.

A simple example – Split on « Number of washing hands » column at the root –

- We can see that the best split (the one with the highest information gain) on the « Number washing hands » column (X_3) is still very low.
- For the first split (at the root), it is better to do it on « in contact with a person » column (X_1) as the information gain is 0.24.

X_1	X_2	X_3	X_4	Y
0	98	0	0	0
1	8	0	0	1
1	12	1	0	1
0	87	1	0	0
1	28	3	0	0
1	97	4	1	1
1	81	7	0	1
1	42	8	1	0
1	2	9	0	0
1	93	9	0	1

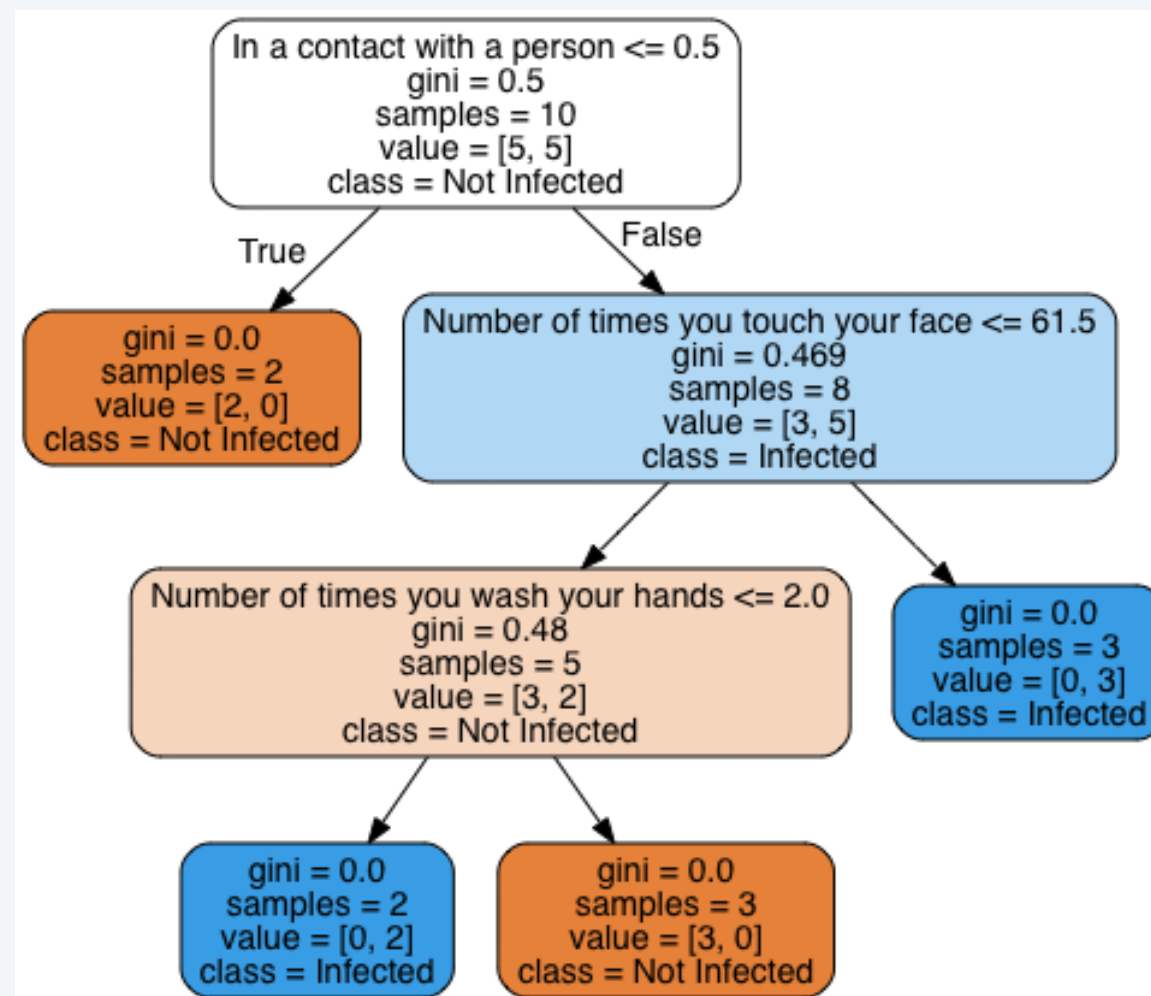
A simple example – Split on « Number of washing hands » column after some iterations –

- After some iterations, splitting on « Number of touching face » column (X_1) becomes interesting.

					IG		
Thresholds	2	→	X_1	X_2	X_3	X_4	Y
	5.5	→	1	8	0	0	1
	8.5	→	1	12	1	0	1
	3.5	→	1	28	3	0	0
		→	1	42	8	1	0
		→	1	2	9	0	0

Basic example after training

- When do we actually stop splitting ?
 - If a node is pure (all the samples of this node belong to the same category), we make it a leaf node and predict the class of the samples.
 - If the maximum information gain = 0, we gain nothing from splitting, we make the node a leaf node and predict the most likely class (majority vote).
 - To avoid overfitting, we set a limit to the depth of the tree.
- By applying the training algorithm on the basic example, we obtain the following tree:



Decision Trees on the Iris Dataset – Data –

- The data set consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica and Iris versicolor), so 150 total samples.
- Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters.



Setosa



Versicolor



Virginica

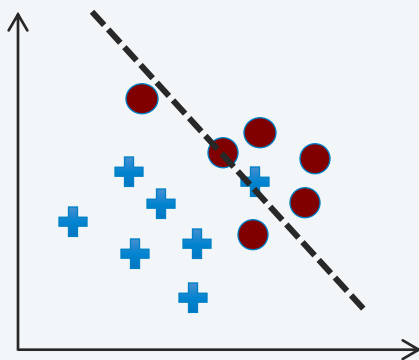
Decision Trees on the Iris Dataset – Model Selection –

- To find the optimal set of hyperparameters, we use a **grid search** method, which is a brute-force exhaustive search paradigm where we specify a list of values for different hyperparameters, and the algorithm evaluate the model performance (via cross validation) for each combination of hyperparameters.
- For decision tree algorithm, we tuned the following two hyperparameters:
 - **Impurity measure:** with the two possibilities :
 - Gini
 - Entropy
 - **The depth of the tree:** with values in $[1, 2, 3, 4, 5, 10, 20, 30, 40]$
- As a result, we obtain the following best parameters:
 - **Best impurity :** « gini »
 - **Best depth :** 5

Decision Trees on the Iris Dataset – Bias/Variance Tradeoff –

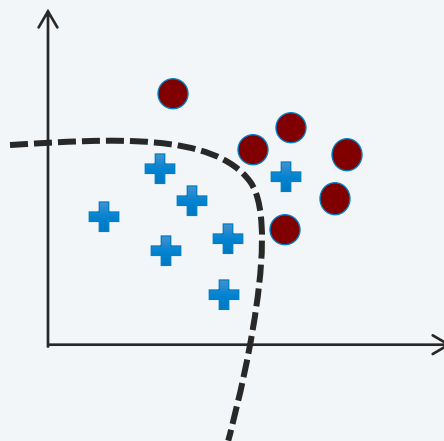
- By increasing the tree depth, we increase the model complexity.
- If the complexity is too small, it will affect the performance on both the training set and the test set (the model will suffer from high bias).
- If it's too big, we could achieve a perfect score on the training set but the performance will deteriorate on the test set (the model will suffer from high variance).

Underfitting



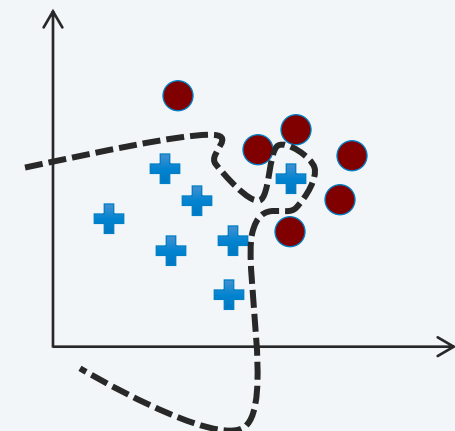
Complexity too small
(**high bias**)

Good Compromise



Complexity: neither
too big, nor too small.

Overfitting

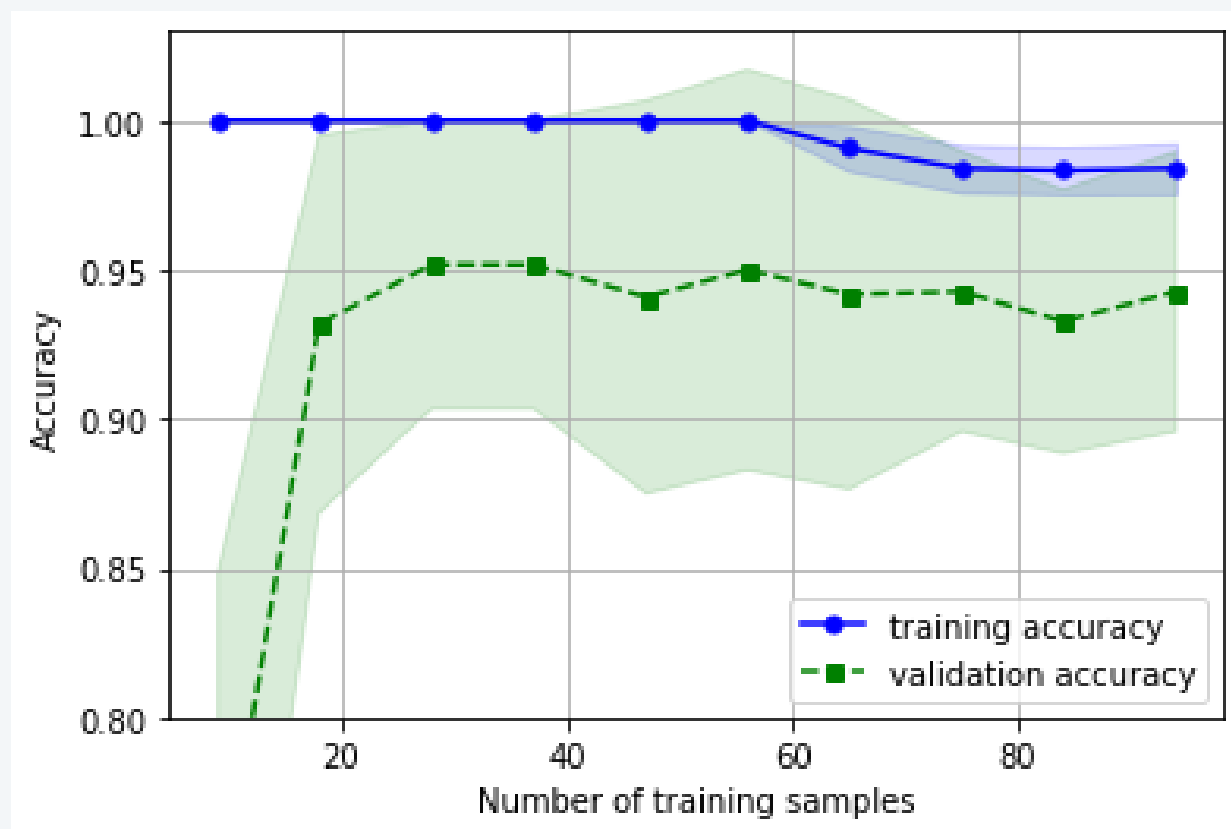


Complexity too big
(**high variance**)

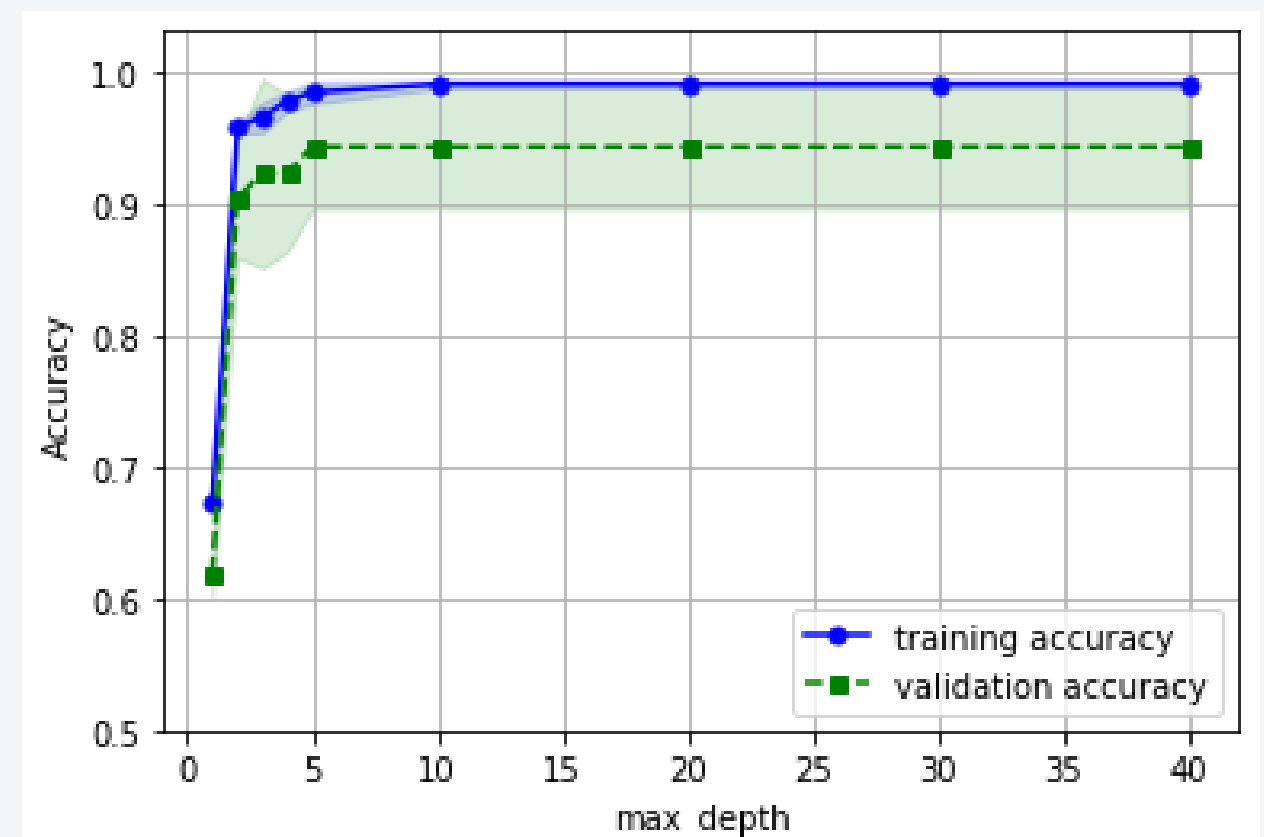
Decision Trees on the Iris Dataset – Overfitting –

- By plotting the model training and validation accuracies as functions of the training set size, we can detect whether the model suffers from high variance or high bias, and whether the collection of more data could help address this problem.

Accuracy vs training set size



Accuracy vs max depth



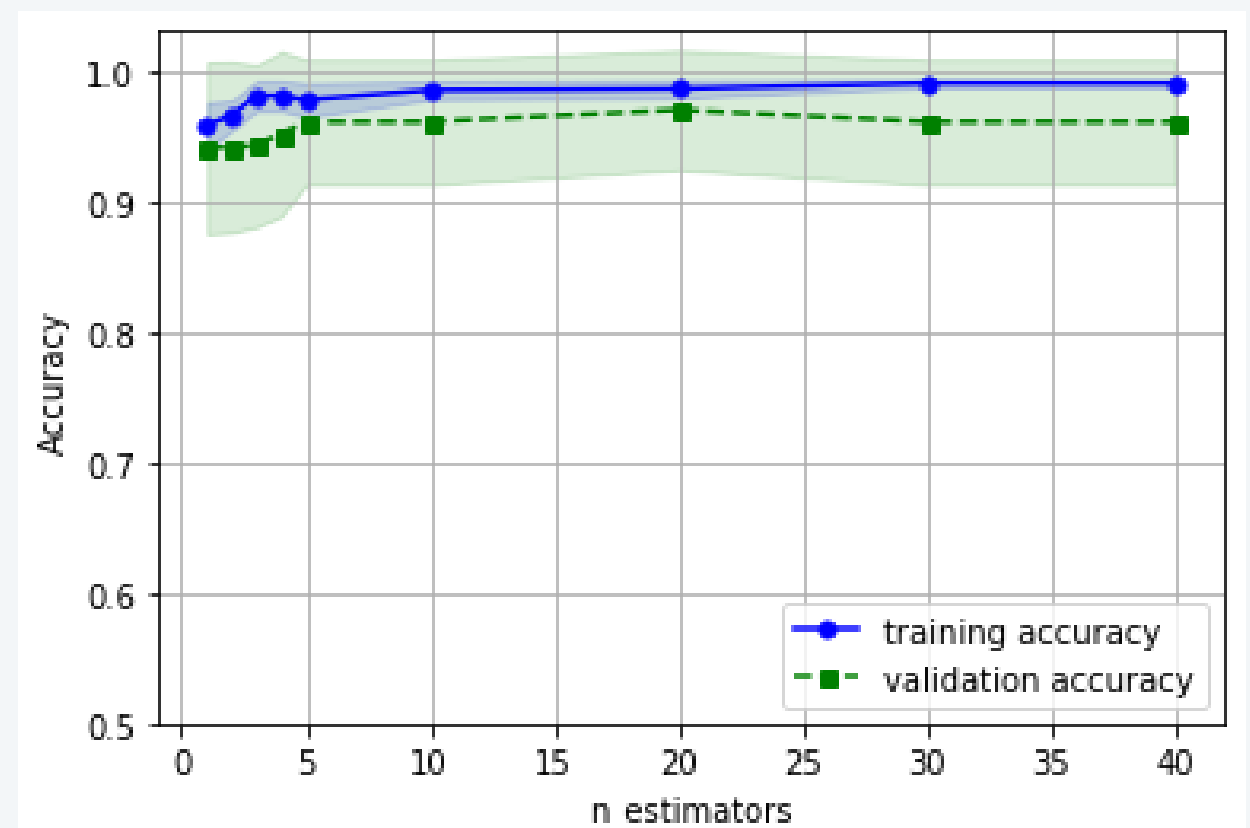
Combining multiple decision trees via Random Forest

- The Random Forest algorithm aims to average multiple decision trees that individually suffer from overfitting, in order to build a model with better generalization performance. It consists in the following steps:

1. Randomly choose n samples from the training set **with replacement** (bootstrap sample).
1. Grow a decision tree from the bootstrap sample. And at each node:
 - Randomly select d attributes without replacement.
 - Split the node using the attribute that provides the best split according to maximizing the information gain.
2. Repeat the first two steps K times.
3. Use the K trees to assign the class label by **majority voting**.

The Random Forest Algorithm on the Iris dataset

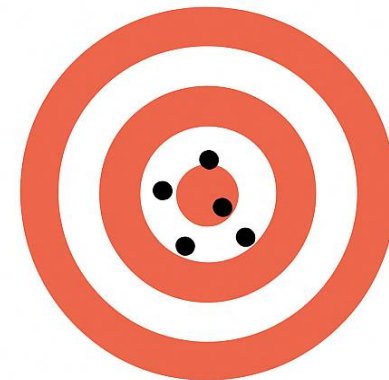
- The Hyperparameters of the Random Forest Algorithm:
 - The sample size n of the bootstrap controls **the bias-variance tradeoff** of the Random Forest.
 - In most libraries (including scikit-learn), the size of the bootstrap sample is chosen to be equal to the number of samples in the training dataset.
 - For the number of attributes d at each split, a reasonable default value (used in scikit-learn) is $d = \sqrt{D}$ where D is the number of attributes in the dataset.
 - So, one big advantage of Random Forest is that it requires very little tuning: The main hyperparameter to tune is the number of trees.
- By tuning the number of trees as a hyperparameter for the Iris dataset, we get 20 as the best value, which is confirmed by the following curve:



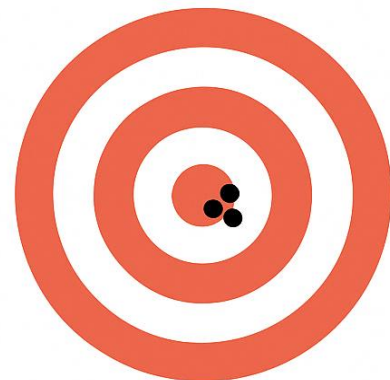
Random Forest Summary : A Bagging Based Ensemble

- The key benefits of random forest is to reduce variance without increasing bias.
- Random forest is known as a bagging method:
 - Uses bootstrap samples (sampling without replacement)
 - Trains independent trees in parallel
 - Agregates predictions:
 - Majority vote (classification)
 - Average (regression)
- 🎲 Extra Randomness for Better Performance
At each split, only a random subset of features is considered.
This reduces correlation between trees → more robust ensemble.




Decision Tree



Random Forest






Random Forest – Pros & Limitations

-  Strengths
 - Reduces overfitting by combining many trees (bagging).
 - Robust to noise and outliers.
 - Handles high-dimensional data well.
 - Simple to use, with good performance out-of-the-box..
-  Limitations – What's Missing?
 - Trees are built independently → they don't learn from each other.
 - No feedback loop: errors made by one tree aren't corrected by the next.
 - May require many trees to achieve high accuracy.
 - Doesn't explicitly focus on hard-to-classify examples.
-  Where Do We Go From Here?
 - “What if we could build a sequence of trees, where each one tries to fix the mistakes of the previous?”

From Bagging to Boosting

Introducing Boosting – Learning from Mistakes

-  Core Idea:
 - *Build a sequence of weak learners (e.g. decision trees), where each one focuses on the errors made by the previous.*
-  How It Works:
 - 1- Start with a simple model (tree #1).
 - 2- Measure its mistakes.
 - 3- Train the next model to correct those mistakes.
 - 4- Repeat, then combine all models into a strong predictor.
-  Goal:
 - Turn a series of weak learners into a strong ensemble.

⚡ AdaBoost: A Boosting Method

- AdaBoost, introduced by Freund and Schapire in 1996, was the first practical Boosting algorithm for binary classification. It builds a strong classifier by sequentially combining weak learners, each one focusing more on the errors made by its predecessors. It consists in the following steps:

1. Initialize uniform weights over all training samples.

2. For K iterations:

- Train a weak learner $f_k(\cdot)$ on the weighted training data.
- Compute the weighted classification error of the weak learner.
- Compute the learner's weight in the final model based on its performance:

$$\alpha_k = \frac{1}{2} \ln\left(\frac{1-\epsilon_k}{\epsilon_k}\right) \text{ where } \epsilon_k \text{ is the classification error.}$$




- Update the sample weights:
 - Increase weights for misclassified examples.
 - Decrease weights for correctly classified ones.
 - Normalize the weights to sum to 1.

3. Aggregate the K weak learners into a final prediction function:

$$F_K(\cdot) = \text{sign}\left(\sum_{m=1}^K \alpha_m f_m(\cdot)\right)$$





AdaBoost: Strengths and Limitations

-  Strengths
 - Strong performance on binary classification tasks, especially with clean data.
 - Interpretable: the final model is a weighted combination of weak classifiers..
 - Focuses learning on hard examples by reweighting misclassified instances.
-  Limitations – What's Missing?
 - Sensitive to noisy data and outliers — heavily penalizes misclassifications.
 - Only naturally supports binary classification (original formulation).
 - Harder to extend to regression tasks or to optimize with custom loss functions.
-  Where Do We Go From Here?
 - *What if we could boost by following the gradient of any loss function, not just the exponential one?*



Toward Gradient Boosting: A Functional Gradient Descent Perspective

-  Motivation
 - While AdaBoost reweights samples based on classification errors, Gradient Boosting takes a more general and powerful approach:
 - It minimizes any differentiable loss function by performing gradient descent in function space, not just exponential loss.
-  Key Idea:
 - We want to find a function $F(\cdot)$ that minimizes the empirical risk:
$$\min_F \sum_{i=1:n} L(y_i, F(x_i)) \text{ where } (x_i, y_i) \text{ is the couple (features, target) for } 1 \leq i \leq n$$

Instead of optimizing $F(\cdot)$ all at once, we build it additively in a greedy stage-wise manner.

Gradient Boosting Algorithm Inputs and Goal

Goal: Learn a function $F(x)$ that minimizes the empirical loss:

$$\mathcal{L}(F) = \sum_{i=1}^n \mathcal{L}(y_i, F(x_i))$$

We construct $F(x)$ additively:

$$F_M(x) = F_0(x) + \sum_{m=1}^M \nu \cdot \gamma_m \cdot h_m(x)$$

At each iteration, γ_m is a scalar coefficient (step size) selected to minimize the loss after fitting the weak learner:

$$\gamma_m = \arg \min_{\gamma} \sum_{i=1}^n \mathcal{L}(y_i, F_{m-1}(x_i) + \gamma \cdot h_m(x_i))$$

Input:

- Training data $\{(x_i, y_i)\}_{i=1}^n$
- Differentiable loss function $\mathcal{L}(y, F(x))$
- Number of iterations M
- Learning rate $\nu \in (0, 1]$

Gradient Boosting: Algorithm Steps

Algorithm:

1. Initialization:

$$F_0(x) = \arg \min_{\gamma} \sum_{i=1}^n \mathcal{L}(y_i, \gamma)$$

This is a constant prediction minimizing the total loss. For example:

- Squared loss: $\gamma = \frac{1}{n} \sum y_i$ (mean of targets)
- Log-loss (binary): $\gamma = \frac{1}{2} \log \left(\frac{p}{1-p} \right)$, where $p = P(y_i = 1)$

2. For $m = 1$ to M :

- Compute pseudo-residuals:

$$r_i^{(m)} = - \left. \frac{\partial \mathcal{L}(y_i, F(x_i))}{\partial F(x_i)} \right|_{F=F_{m-1}}$$

- Fit weak learner $h_m(x)$ to $(x_i, r_i^{(m)})$
- Line search to compute:

$$\gamma_m = \arg \min_{\gamma} \sum_{i=1}^n \mathcal{L}(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$$

- Update model:

$$F_m(x) = F_{m-1}(x) + \nu \cdot \gamma_m \cdot h_m(x)$$

3. Final model: $F_M(x)$



Gradient Boosting: Step-by-Step Intuition

- 1. Initialization:
 - We start with a simple constant prediction for all inputs.
 - This is the best constant that minimizes the loss function on the training data.
 - Think of this as our "baseline" guess before learning anything about x .
- 2. Compute Pseudo-Residuals:
 - These residuals represent the direction and magnitude of error made by the current model.
 - For squared error: residuals are simply : $y_i - F_{m-1}(x_i)$
 - For other losses: we compute the negative gradient of the loss with respect to current predictions.
 - This tells us how to adjust our model to reduce the loss.
- 3. Fit Weak Learner to Residuals:
 - We fit a simple model (e.g., decision tree) to predict the residuals.
- 4. Line Search for Step Size:
 - We find the best parameter γ_m that minimizes the loss after applying the new weak learner
 - This controls how strongly we trust the new learner
- 5. Update the Model:
 - We move in the direction suggested by the new learner
 - Each step improves the model a little, but in the right prediction



Summary:

Gradient Boosting is like iteratively correcting your mistakes, one small step at a time, in the direction of steepest descent.



Model Comparison: Summary of Strengths and Limitations

Model	Task Type	Pros	Cons
Linear Regression	Regression	<ul style="list-style-type: none">- Simple, interpretable- Fast to train	<ul style="list-style-type: none">- Assumes linearity- Sensitive to outliers
Logistic Regression	Classification	<ul style="list-style-type: none">- Probabilistic output- Interpretable coefficients	<ul style="list-style-type: none">- Linear decision boundary- Limited for complex patterns
Decision Tree	Both	<ul style="list-style-type: none">- Intuitive, interpretable- Handles mixed data types	<ul style="list-style-type: none">- Prone to overfitting- Unstable to small changes
Random Forest	Both	<ul style="list-style-type: none">- Reduces variance- Robust to overfitting- Feature importance	<ul style="list-style-type: none">- Less interpretable- Slower to predict
AdaBoost	Classification	<ul style="list-style-type: none">- Focuses on hard cases- Often better than single models	<ul style="list-style-type: none">- Sensitive to noise/outliers- Originally for binary tasks
Gradient Boosting	Both	<ul style="list-style-type: none">- Flexible loss functions- High predictive accuracy	<ul style="list-style-type: none">- Slower to train- Sensitive to hyperparameters



Modern Boosting Models: Extending Gradient Boosting in Practice

Model	Task Type	Pros	Cons
XGBoost	Both	<ul style="list-style-type: none">- Regularization (L1/L2)- Fast and scalable- Parallel tree construction	<ul style="list-style-type: none">- Many hyperparameters- Can overfit if not tuned properly
LightGBM	Both	<ul style="list-style-type: none">- Very fast on large data- Leaf-wise tree growth for accuracy- Categorical support	<ul style="list-style-type: none">- May overfit small data- Less interpretable
CatBoost	Both	<ul style="list-style-type: none">- Handles categorical variables natively- Stable with little tuning- Avoids target leakage	<ul style="list-style-type: none">- Slower than LightGBM- More recent, less adopted in some tools



Key Takeaways:

- All three are built on gradient boosting, but offer engineering optimizations for speed, memory, and accuracy.
- Best used in structured/tabular data, especially for competitions or production.



Choose based on:

- Data size
- Categorical feature handling
- Training speed needs
- Tuning flexibility

Appendix

A zoom on the notion of Gradient Descent

Position of the Problem:

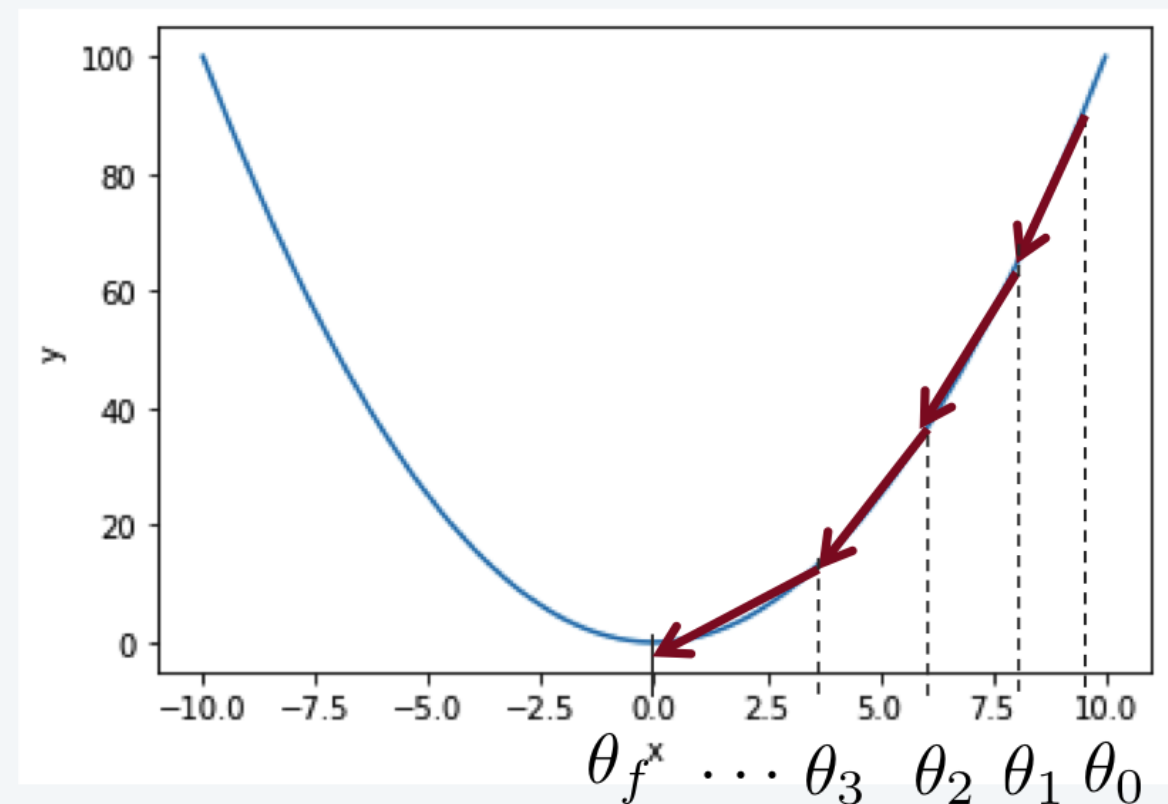
- The Gradient Descent algorithm is the backbone of Machine Learning.
- Let us start with a basic example.
- We want to minimize the function $f : \theta \mapsto \theta^2$ by using the gradient descent algorithm.

- As shown in the figure below, finding the minimum can be achieved by finding where the derivative (slope) is zero. To that end, we can do the following steps:

- Choose a random initial point θ_0
- Choose a learning rate $\eta > 0$ and repeat until convergence:

$$\theta_{k+1} \leftarrow \theta_k - \eta \frac{df}{d\theta}(\theta_k)$$

Gradient Descent



Gradient Descent Theorem:

- Before stating the gradient descent theorem, let's recall that the **gradient** is just the multidimensional equivalent of the **derivative**.

- For $f : \theta \in \mathbb{R}^d \mapsto \mathbb{R} :$
$$\nabla_{\theta} f(\theta) = \left(\frac{\partial f}{\partial \theta_1}(\theta), \frac{\partial f}{\partial \theta_2}(\theta), \dots, \frac{\partial f}{\partial \theta_d}(\theta) \right)$$

Gradient Descent Algorithm:

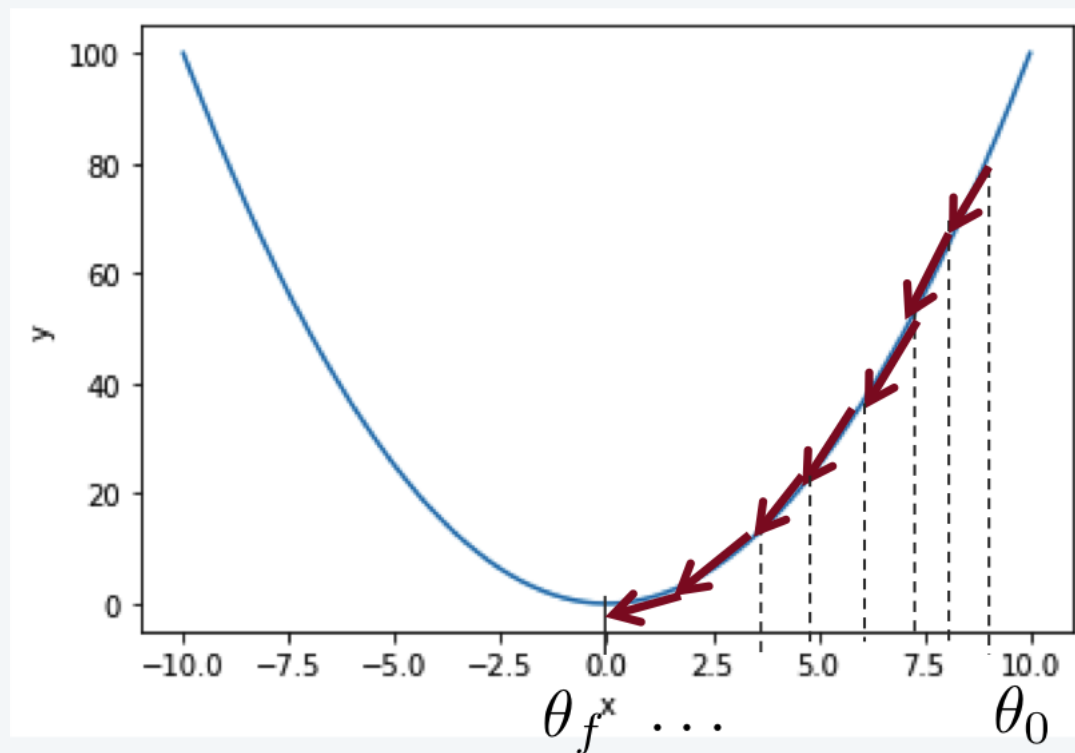
- Suppose we have a convex and differentiable function $f : \mathbb{R}^d \rightarrow \mathbb{R}$
- Suppose its gradient is (*Lipschitz*) continuous with constant ϵ . Let's fix the learning rate $\eta \leq \frac{1}{\epsilon}$
- Then, if we run the gradient descent algorithmn for K iterations:
 - Initialize randomly θ_0
 - Repeat K times
$$\theta_{k+1} \leftarrow \theta_k - \eta \nabla_{\theta} f(\theta_k)$$
- It will yield a solution which satisfies:
$$f(\theta_k) - f(\theta^*) \leq \frac{\| \theta_0 - \theta^* \|^2_2}{2\eta K}$$

where : $\| \cdot \|_2$ is the \mathcal{L}^2 norm on \mathbb{R}^d (i.e : $\forall z \in \mathbb{R}^d \quad \| z \|_2^2 = z^T z$)

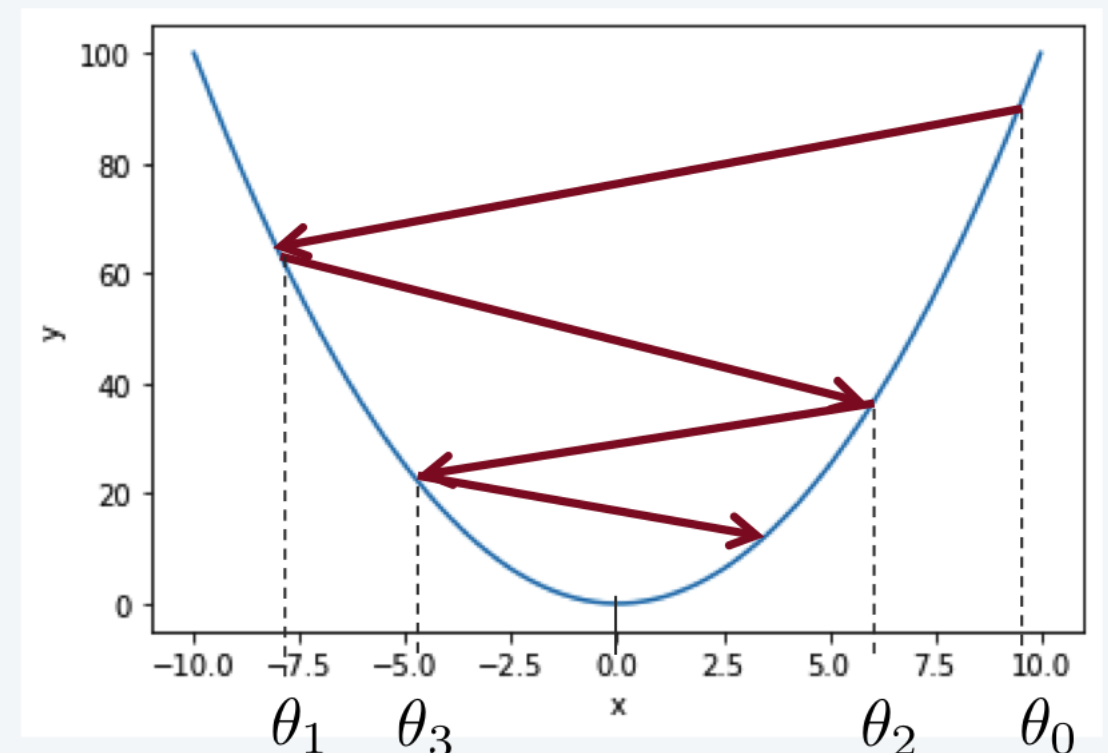
Choice of the learning rate in Gradient Descent:

- Under the assumptions of the gradient descent theorem, the algorithm is guaranteed to converge and it converges with a rate of $O(\frac{1}{K})$
- The learning rate η should not be too big so that the objective function does not blow up, but not too small so that it doesn't take too much time to converge.

Small learning rate



Big learning rate



- In this case, the learning rate is fixed, we will present in Lecture 4 other versions of the gradient descent algorithm but with a learning rate that decreases with iterations, when we are closer to the optimal parameters.

Using the gradient Descent in the case of a linear regression

The **training problem** has been written as the following equivalent minimization problem:

$$\min_{\hat{w} \in \mathbb{R}^{D+1}} \underbrace{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{w}^T x_i)^2}_{J(\hat{w})}$$

Matrix Notation and Optimization :

- It is computationally more efficient to write the optimization problem with matrix notation.
- To that end, let's introduce the following matrix notation for the training data $\{(X_i, Y_i)_{1 \leq i \leq N}\}$

$$\hat{X} = \begin{bmatrix} - & x_1 & - & 1 \\ - & x_2 & - & 1 \\ \vdots & \vdots & \vdots & \vdots \\ - & x_N & - & 1 \end{bmatrix} \in \mathbb{R}^{N \times (D+1)} \quad Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \in \mathbb{R}^N$$

- We then deduce the following prediction matrix P :

$$P = \hat{X} \hat{w} = \begin{bmatrix} \hat{w}^T x_1 \\ \hat{w}^T x_2 \\ \vdots \\ \hat{w}^T x_N \end{bmatrix} \in \mathbb{R}^N$$

- If we denote by $\|\cdot\|_2$ the \mathcal{L}^2 norm on \mathbb{R}^N (i.e : $\forall z \in \mathbb{R}^N \quad \|z\|_2^2 = z^T z$), we can then express the cost function J we look to minimize as follows :

$$J(\hat{w}) = \frac{1}{N} \|\hat{X} \hat{w} - Y\|_2^2$$

Matrix Notation and Optimization :

- To cost function $J(\hat{w}) = \frac{1}{N} || \hat{X}\hat{w} - Y ||_2^2$ is a differentiable convex function whose minima are thus characterized by the setting the gradient to zero.

- In order to differentiate the cost function, we will need the following formulas:

$$\begin{aligned} \forall z \in \mathbb{R}^n \quad \forall A \in \mathbb{R}^{n \times n} \quad \nabla_z(z^T A z) &= (A + A^T)z \\ \forall z \in \mathbb{R}^n \quad \forall \xi \in \mathbb{R}^n \quad \nabla_z(z^T \xi) &= \xi \end{aligned}$$

- So,
$$\begin{aligned} \nabla_{\hat{w}} J(\hat{w}) = 0 &\iff \nabla_{\hat{w}} \left(\frac{1}{N} (\hat{X}\hat{w} - Y)^T (\hat{X}\hat{w} - Y) \right) \\ &\iff \frac{1}{N} \left(\nabla_{\hat{w}} (\hat{w} \hat{X}^T \hat{X} \hat{w}) - \nabla_{\hat{w}} (\hat{w}^T \hat{X}^T Y) - \nabla_{\hat{w}} (Y^T \hat{X} \hat{w}) \right) \\ &\iff \frac{2}{N} (\hat{X}^T \hat{X} \hat{w} - \hat{X}^T Y) = 0 \\ &\iff \hat{X}^T \hat{X} \hat{w} = \hat{X}^T Y \end{aligned}$$

- We deduce the following expression of the gradient :

$$\nabla_{\hat{w}} J(\hat{w}) = \frac{2}{N} (\hat{X}^T \hat{X} \hat{w} - \hat{X}^T Y)$$

- If $\hat{X}^T \hat{X}$ is invertible, the optimal \hat{w}^* is given by the following closed form: $\hat{w}^* = (\hat{X}^T \hat{X})^{-1} \hat{X}^T Y$

Using the gradient Descent in the case of a logistic regression

- The **training problem** has been written as the following equivalent minimization problem:

$$\min_{w \in \mathbb{R}^D} \underbrace{-\frac{1}{N} \sum_{i=1}^N (y_i \log(\sigma(w^T x_i)) + (1 - y_i) \log(1 - \sigma(w^T x_i)))}_{J(w)}$$

Gradient Descent for Logistic Regression:

- Using these two properties of the sigmoid function:

$$\forall z \in \mathbb{R} \quad \sigma(-z) = 1 - \sigma(z)$$

$$\forall z \in \mathbb{R} \quad \sigma'(z) = \sigma(z)(1 - \sigma(z)) = \sigma(z)\sigma(-z)$$

- We can rewrite the cost function as follows:

$$\begin{aligned} J(w) &= -\frac{1}{N} \sum_{i=1}^N (y_i \log(\sigma(w^T x_i)) + (1 - y_i) \log(1 - \sigma(w^T x_i))) \\ &= -\frac{1}{N} \sum_{i=1}^N (y_i w^T x_i + \log(\sigma(-w^T x_i))) \end{aligned}$$

- Let's take the gradient w.r.t. w :

$$\begin{aligned} \nabla_w J(w) &= \nabla_w \left(-\frac{1}{N} \sum_{i=1}^N (y_i w^T x_i + \log(\sigma(-w^T x_i))) \right) \\ &= \frac{1}{N} \sum_{i=1}^N (\sigma(w^T x_i) - y_i) x_i \end{aligned}$$

Matrix Notation for the Gradient :

- let's introduce the following matrix notation for the training data $\{(X_i, Y_i)_{1 \leq i \leq N}\}$:

$$X = \begin{bmatrix} - & x_1 & - \\ - & x_2 & - \\ \vdots & \vdots & \vdots \\ - & x_N & - \end{bmatrix} \in \mathbb{R}^{N \times D}$$

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \in \{0, 1\}^N$$

- The prediction vector $P \in \mathbb{R}^N$ is then defined as follows:

$$P = \begin{bmatrix} \sigma(w^T x_1) \\ \sigma(w^T x_2) \\ \vdots \\ \sigma(w^T x_N) \end{bmatrix} \in \mathbb{R}^N$$

- And the error vector $\epsilon \in \mathbb{R}^N$ is defined as follows:

$$\epsilon = P - Y = \begin{bmatrix} \sigma(w^T x_1) - y_1 \\ \sigma(w^T x_2) - y_2 \\ \vdots \\ \sigma(w^T x_N) - y_N \end{bmatrix} \in \mathbb{R}^N$$

Matrix Notation for the Gradient :

- The error vector $\epsilon \in \mathbb{R}^N$ can be represented by $\hat{\epsilon}$: a diagonal $\mathbb{R}^{N \times N}$ matrix as follows:

$$\hat{\epsilon} = \begin{bmatrix} \sigma(w^T x_1) - y_1 & & & \\ & \sigma(w^T x_2) - y_2 & & \\ & & \ddots & \\ & & & \sigma(w^T x_N) - y_N \end{bmatrix} \in \mathbb{R}^{N \times N}$$

- As a result, the matrix $\hat{\epsilon}X \in \mathbb{R}^{N \times D}$ contains all the information needed to compute the gradient:

$$\hat{\epsilon}X = \begin{bmatrix} - & (\sigma(w^T x_1) - y_1)x_1 & - \\ - & (\sigma(w^T x_2) - y_1)x_2 & - \\ \vdots & \vdots & \vdots \\ - & (\sigma(w^T x_N) - y_1)x_N & - \end{bmatrix} \in \mathbb{R}^{N \times D}$$

- Indeed, if R_1, \dots, R_N are the rows $\hat{\epsilon}X$, the gradient of J w.r.t. w can be expressed as follows:

$$\boxed{\nabla_w J(w) = \frac{1}{N} \sum_{i=1}^N R_i}$$