Université IBN TOFAIL – Kénitra
ECOLE NATIONALE DES SCIENCES APPLIQUEES

Cours Intelligence Artificielle

CH4: Machine Learning - Apprentissage Supervisé: Arbres de Décision (Decision Trees) et Forêt aléatoire (Random Forest)

Par:

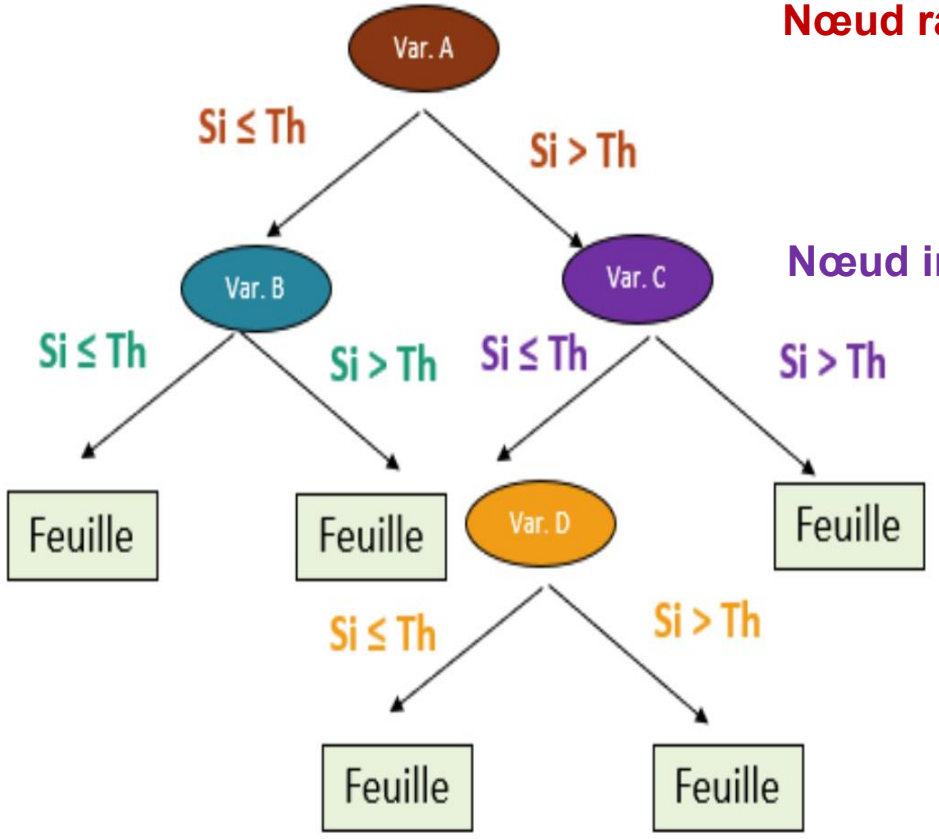
Hassan EL FADIL

elfadil.h@gmail.com

1. Arbres de Décision (Decision Tree)

1.1. Définition:

- Un arbre de décision (Decision tree) est un modèle d'apprentissage automatique utilisé pour **classer** ou **prédire** des données en divisant celles-ci en sous-groupes successifs basés sur des règles simples dérivées des attributs des données.
- Il est représenté comme un diagramme en forme d'arbre avec :
  - Un **nœud racine** (le point de départ).
  - Des **nœuds internes** ou **nœud de décision** (décisions ou tests sur les caractéristiques).
  - Des **feuilles** (résultats ou classes finales).



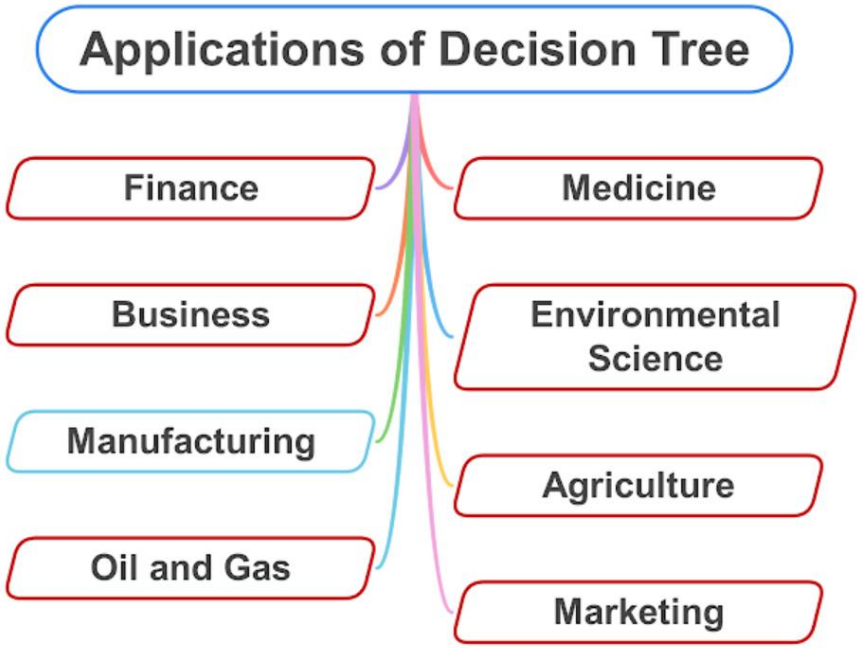
Nœud racine (root)

Nœud interne (node)

Feuille (Leaf)

1.2. Applications des arbres de décision

- **Classification :**
  - Détection de fraudes bancaires.
  - Prédiction de maladies en médecine (par exemple : cancer, diabète).
- **Régression :**
  - Estimation des prix immobiliers.
  - Prévisions des ventes dans le commerce.
- **Aide à la décision :**
  - Planification stratégique en entreprise.
  - Diagnostics dans les systèmes industriels.



1.3. Avantages et inconvénients

Avantages :

- Visualisation intuitive sous forme d'arbre.
- Gère les données **catégoriques** et **numériques**.
- Capable de capturer des relations complexes entre variables.
- Pas besoin de normalisation ou de mise à l'échelle des données.
- Gère automatiquement les valeurs manquantes.
- Efficace pour des **petits jeux** de données.

Inconvénients:

- **Surcharge (overfitting)** : Peut produire des modèles trop spécifiques aux données d'entraînement, au détriment de la généralisation.
- **Instabilité** : Sensible aux variations des données (petits changements dans les données peuvent entraîner un arbre très différent).
- Limitation pour des ensembles de **données volumineux**.

1.4. Fonctionnement d'un arbre de décision

1.4.1. Critères de Division et Algorithmes:

- Un **arbre de décision** se compose de nœuds et de branches (qui relient les tests aux décisions suivantes).
- Chaque branche divise les données en sous-ensembles selon les caractéristiques de l'entrée.
- L'algorithme choisit l'attribut qui permet de mieux séparer les données à chaque étape, et cela continue jusqu'à ce que des critères d'arrêt soient atteints.
- L'arbre est ensuite utilisé pour prédire la classe ou la valeur d'une nouvelle donnée en suivant les décisions jusqu'à la feuille.

Critères de division (splits)

- Entropie et Gain d'information.
- Indice de Gini.
- Réduction de variance (pour la régression).

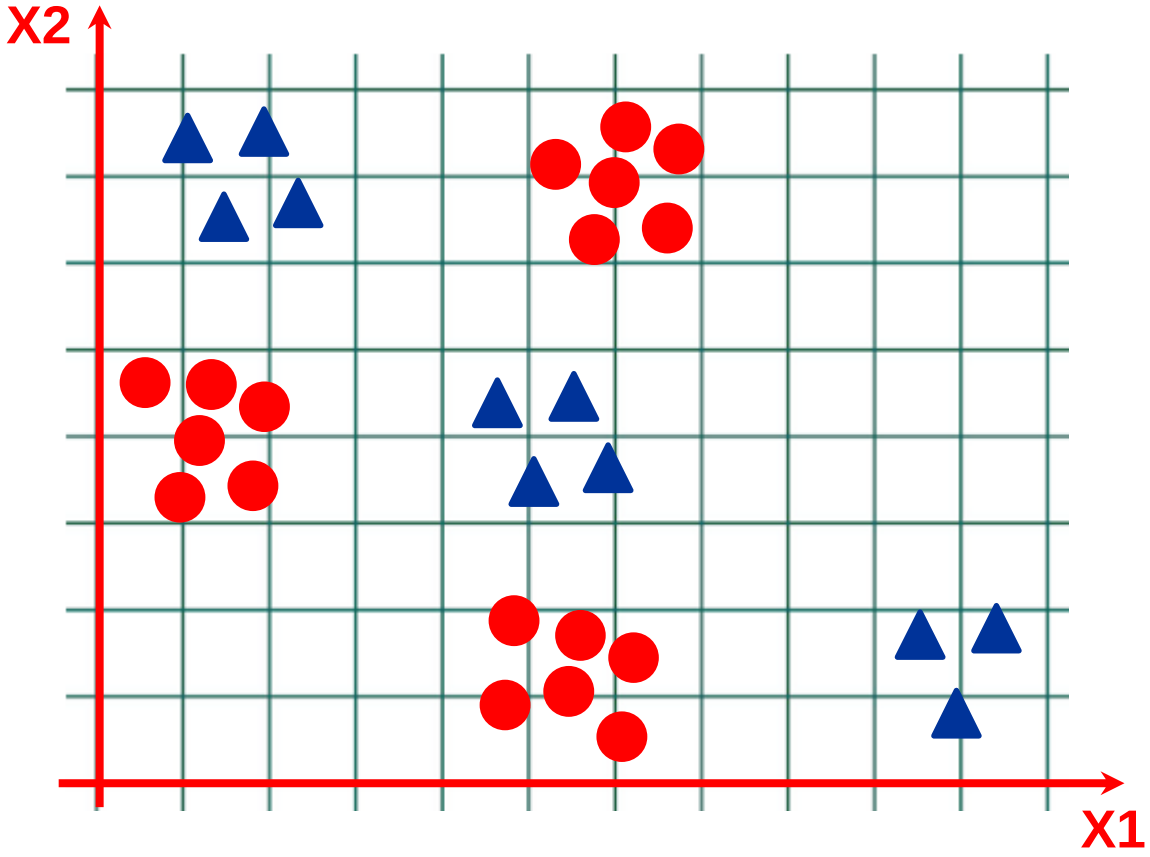
Algorithmes de construction

- CART (Classification and Regression Trees).
- ID3 (Iterative Dichotomiser 3).

1.4.2. Exemple illustratif (Cas de la Classification):

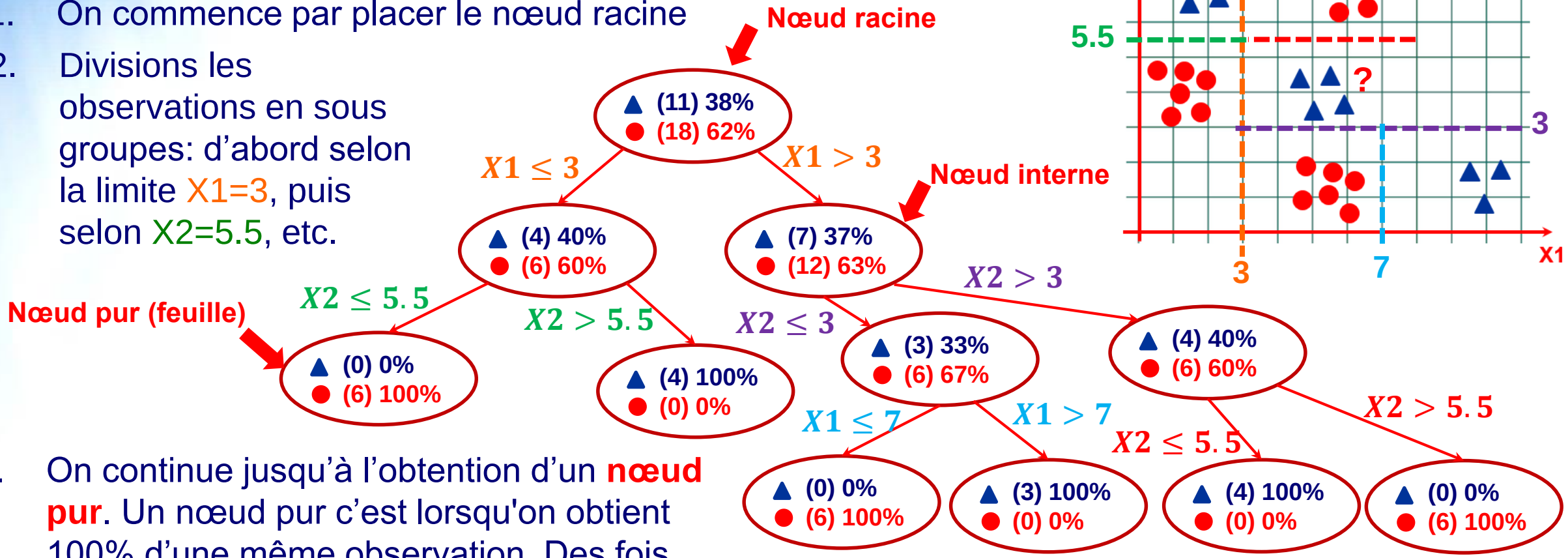
Considérons une base de données d'une ensemble d'observations représentés ici par des cercles et des triangles dans une espace à deux dimensions

- **Etiquettes: « labels »:**
  - ▲ : Triangles (total: 11)
  - ● : Cercles (total: 18)
- **Caractéristiques** : « **Features** »: variables d'entrée ou les attributs: **X1 et X2**



1.4.3. Construction de l'arbre de décision:

1. On commence par placer le nœud racine
2. Divisions les observations en sous groupes: d'abord selon la limite X1=3, puis selon X2=5.5, etc.



3. On continue jusqu'à l'obtention d'un **nœud pur**. Un nœud pur c'est lorsqu'on obtient 100% d'une même observation. Des fois on peut ne pas obtenir un nœud 100% pur!

Question: A quelle classe appartient le point de coordonnées (6.5; 4)? Il suffit de suivre l'arbre!!

1.4.4. Phase d'entraînement de l'arbre: Comment choisir les séparations?

- Dans l'arbre précédent, les séparations ont été positionnées de manière intuitive.
- **Question** : Comment l'algorithme détermine-t-il les séparations optimales afin d'assurer une meilleure performance ?

- L'algorithme sélectionne une caractéristique (variable d'entrée) pour diviser les données de manière optimale (X1 ou X2 dans l'exemple précédent).
- Le critère de sélection peut être basé sur des mesures comme :
  - **Entropie et gain d'information** (pour les données catégoriques: {Rouge, Bleu, Vert} ; {Petit, Moyen, Grand} ; {Ordinateur, Smartphone, Tablette}).
  - **Indice de Gini** (pour des classifications).
  - **Variance réduite** (pour les problèmes de régression).

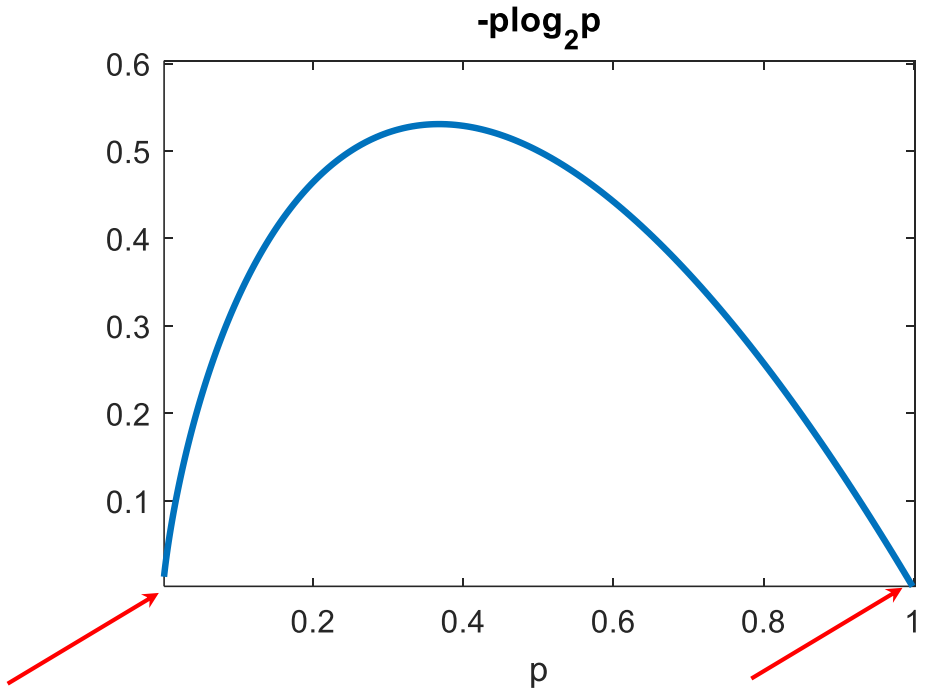
1.4.4.1. Entropie:

$$Entropy(S) = -\sum_{c \epsilon C} p(c)\log_2(p(c))$$

$p(c)$: Probabilité d 'appartenance à une classe

$\log_2$: Logarithme base 2:

$$\log_2(x) = y \Leftrightarrow 2^y = x$$



L'objectif de l'entropie est d'obtenir une probabilité aussi élevée que possible dans une feuille de l'arbre, tout en ayant, pour les autres classes du même nœud, des probabilités aussi faibles que possible.

L'entropie s'annule lorsqu'une probabilité d'appartenance se rapproche de 0 ou de 1.

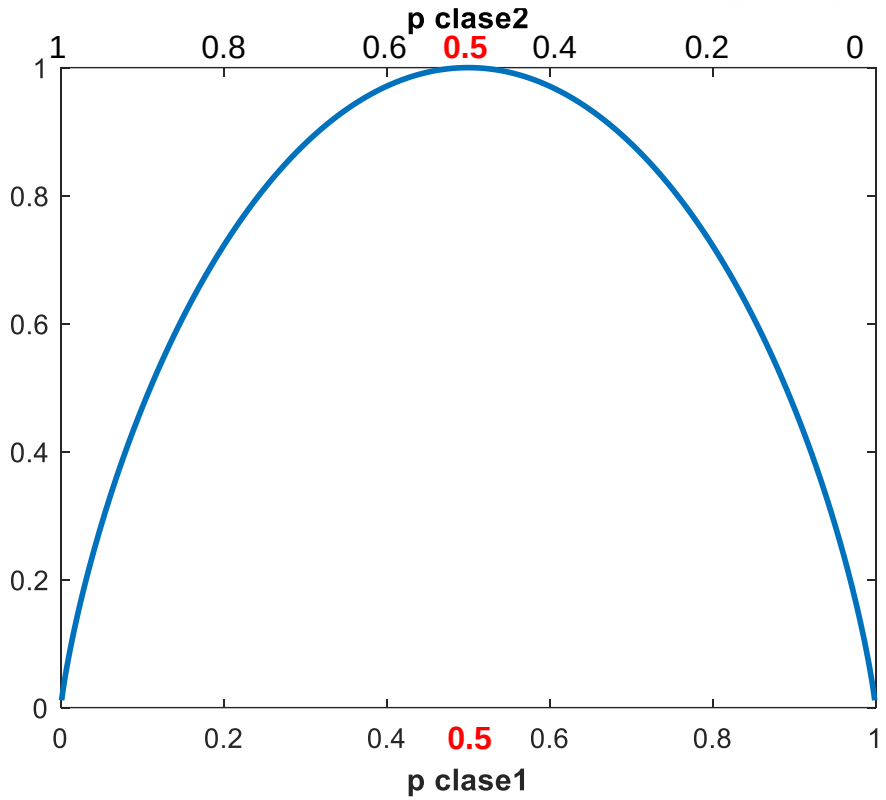
1.4.4.1. Entropie:

Entropie dans le cas d'une classification binaire (deux classes):

p : Probabilité d 'appartenance à la classe 1
$1 - p$: Probabilité d 'appartenance à la classe 2

$$Entropy = -p\log_2(p) - (1-p)\log_2(1-p)$$

- L'entropie est égale à **zéro** lorsqu'une des probabilités atteint 100 %, ce qui correspond à une séparation parfaite entre les deux classes.
- L'entropie atteint sa valeur maximale de 1 lorsque les deux probabilités sont égales à 50 %, indiquant une répartition totalement incertaine entre les classes.

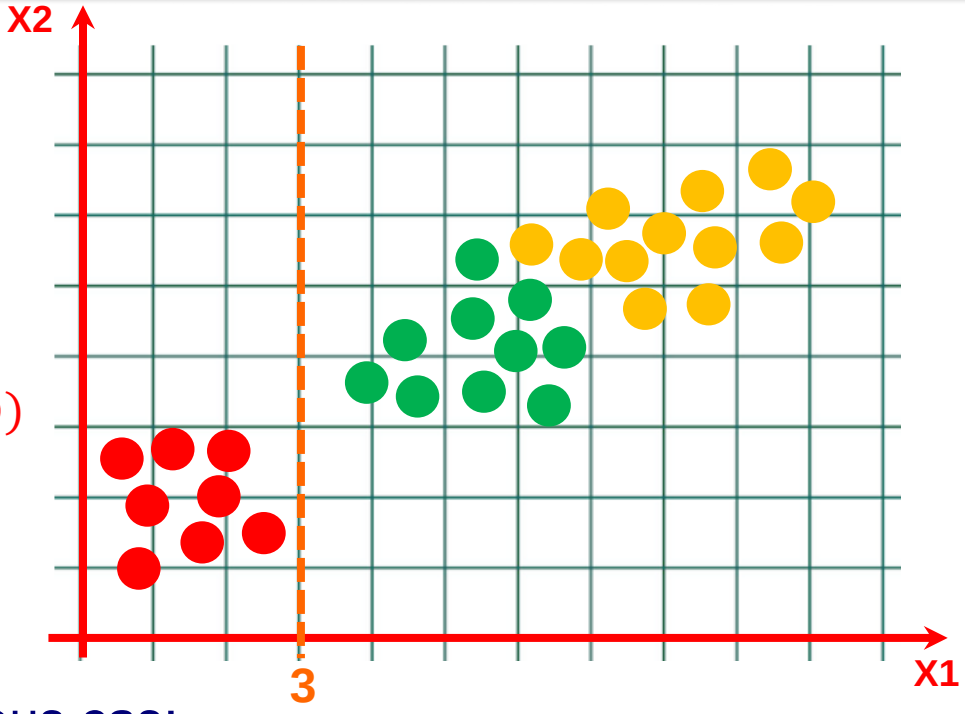


Nombres de classes	Entropie max ($\log_2(c)$)
2	$\log_2(2) = 1$
3	$\log_2(3) = 1.585$
4	$\log_2(4) = 2$
5	$\log_2(5) = 2.322$

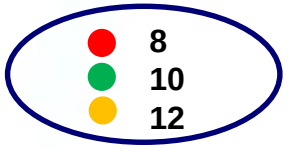
1.4.4.1. Entropie:

- Exemple: Base de données avec 3 classes différentes:
  - Classe rouges: 8
  - Classes verts: 10
  - Classe oranges: 12

$$Entropy(S) = -\sum_{c \epsilon C} p(c)\log_2(p(c))$$



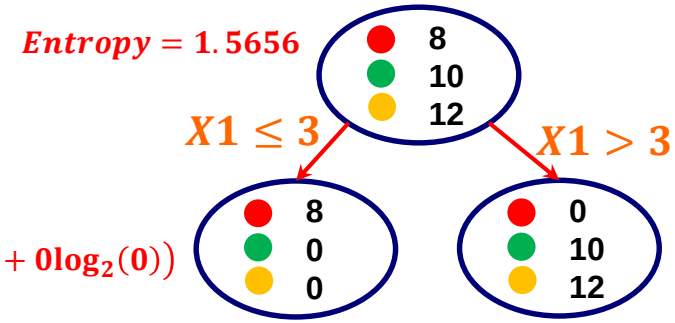
Nœud racine



$$Entropy = -\left(\frac{8}{30}\log_2\left(\frac{8}{30}\right) + \frac{10}{30}\log_2\left(\frac{10}{30}\right) + \frac{12}{30}\log_2\left(\frac{12}{30}\right)\right) = 1.5656$$

- Pour illustrer comment fonctionne l'entropie, Considérons deus cas:

- **1er cas:** Séparation par rapport à **X1=3**

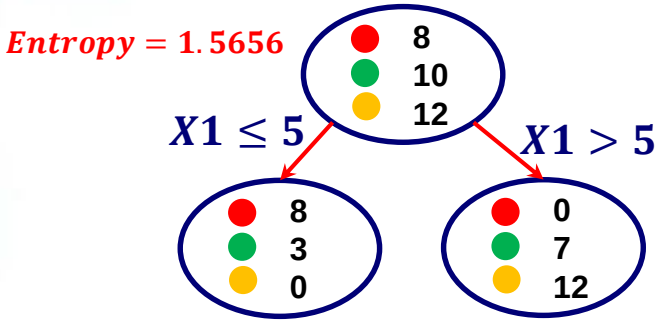


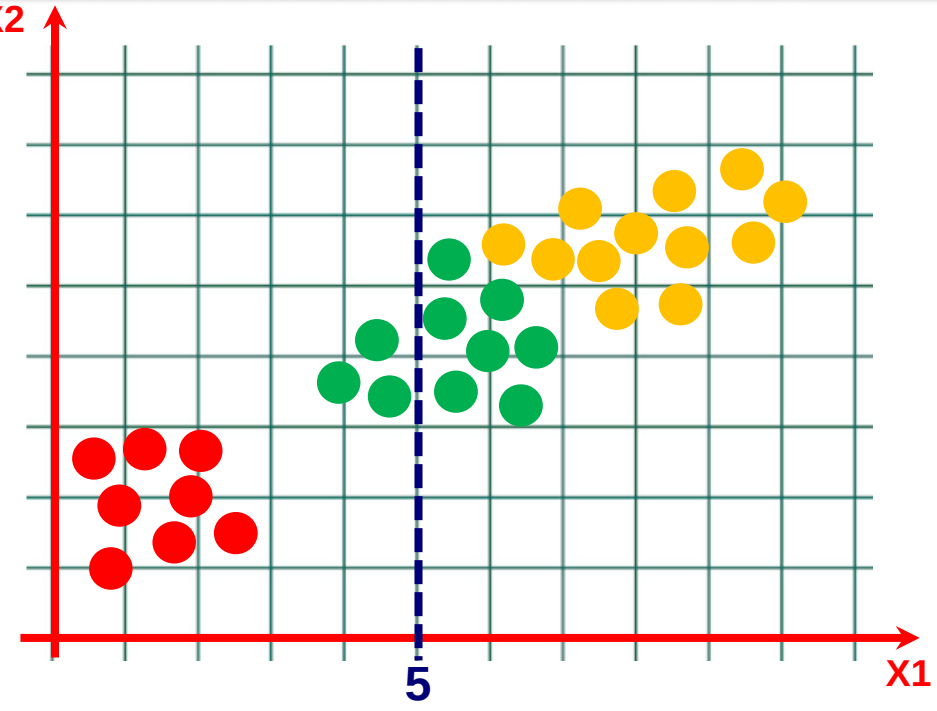
$$Entropy = -(1\log_2(1) + 0\log_2(0) + 0\log_2(0)) = 0$$

$$Entropy = -\left(0\log_2(0) + \frac{10}{22}\log_2\left(\frac{10}{22}\right) + \frac{12}{22}\log_2\left(\frac{12}{22}\right)\right) = 0.994$$

1.4.4.1. Entropie:

- **2ème cas:** Séparation par rapport à **X1=5**





$$Entropy = -\left(\frac{8}{11}\log_2\left(\frac{8}{11}\right) + \frac{3}{11}\log_2\left(\frac{3}{11}\right) + 0\log_2(0)\right) = 0.8454$$

$$Entropy = -\left(0\log_2(0) + \frac{7}{19}\log_2\left(\frac{7}{19}\right) + \frac{12}{19}\log_2\left(\frac{12}{19}\right)\right) = 0.9495$$

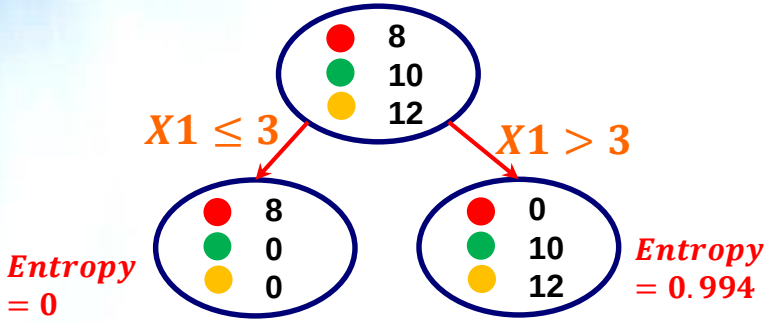
- Laquelle des deux séparation est meilleure?
- Pour cela on utilise l'entropie totale (Overall Entropy)

$$Overall\ Entropy = \sum_{i=1}^{2} p_i \times Entropy_i$$

p_i: Probabilité d 'appartenance au nœud *i*.
$Entropy_i$: Entropie du nœud *i*.

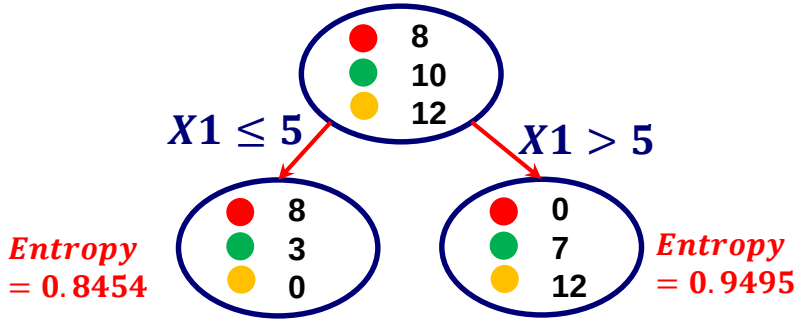
1.4.4.1. Entropie:

- **1er cas:** Séparation par rapport à **X1=3**



$$Overall\ Entropy = \frac{8}{30} \times 0 + \frac{22}{30} \times 0.994 = 0.7289$$

- **2ème cas:** Séparation par rapport à **X1=5**



$$Overall\ Entropy = \frac{11}{30} \times 0.8454 + \frac{19}{30} \times 0.9495 = 0.9113$$

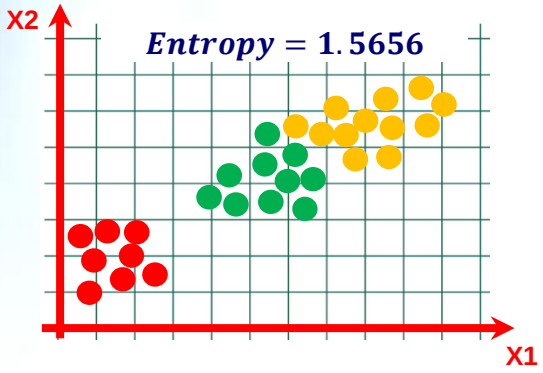
⇨ L'entropie totale est petite pour le 1er cas, la séparation **X1=3** est donc **meilleure** que la séparation **X1=5**

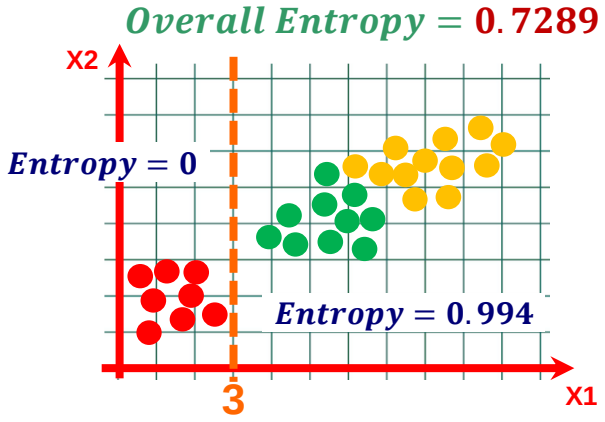
Ici, nous avons considéré deux cas. Cependant, l'algorithme explore plusieurs seuils de séparation en parcourant toutes les valeurs possibles de X1. À chaque seuil, il calcule l'entropie totale. La séparation finale retenue est celle qui minimise l'entropie totale.

1.4.4.2. Gain d'information (Information Gain):

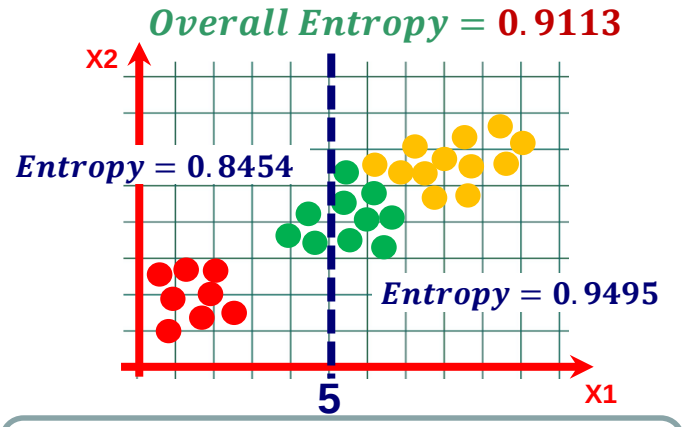
$$Information\ Gain = Entropy(S) - \sum_{i=1}^{k} \frac{|S_i|}{|S|} \times Entropy(S_i)$$
$$= Entropy(S) - Overall\ Entropy$$

- $Entropy(S)$: L'entropie de l'ensemble initial S.
- $Entropy(S_i)$: L'entropie des sous-ensembles S_i obtenus après la division.
- $\frac{|S_i|}{|S|}$: Le poids relatif du sous-ensemble S_i par rapport à l'ensemble initial S.
- k : Nombre de sous-ensembles (ou partitions) obtenus après une division de l'ensemble initial S





$$Information\ Gain = 1.5656 - 0.7289 = 0.8367$$



$$Information\ Gain = 1.5656 - 0.9113 = 0.6543$$

*La séparation optimale est celle qui **maximise le Gain d'information**. Dans ce cas, la meilleure séparation correspond à **X1=3***

1.4.4.3. Impureté de Gini (Gini Impurity):

$$Gini\ Impurity = 1 - \sum_{i=1}^{k} p_i^2$$

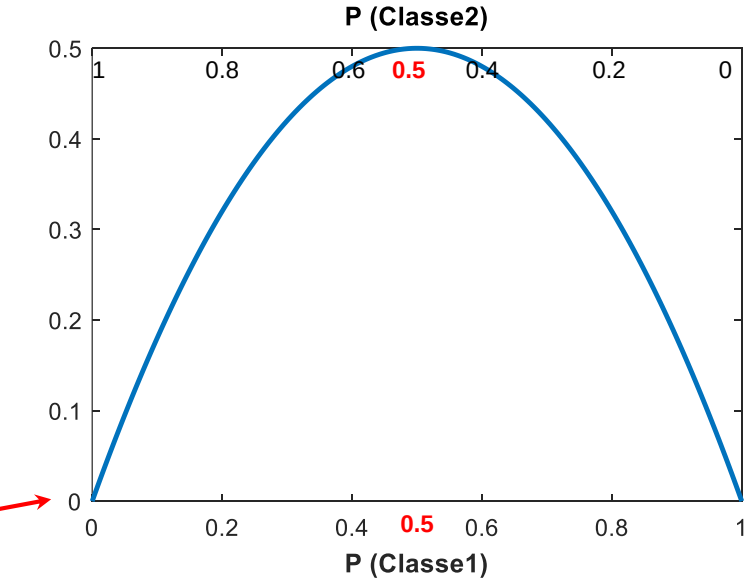
- p_i: est la probabilité d'appartenance à la classe *i* dans un nœud.
- k : est le nombre total de classes.

- L'impureté de Gini c'est une métrique qui permet à l'algorithme de déterminer les frontières de séparation optimales.

- Cas d'une classification binaire (deux classes):

  p : Probabilité d 'appartenance à la classe 1
  $1 - p$: Probabilité d 'appartenance à la classe 2

$$Gini\ Impurity = 1 - p^2 - (1-p)^2 = 2p(1-p)$$

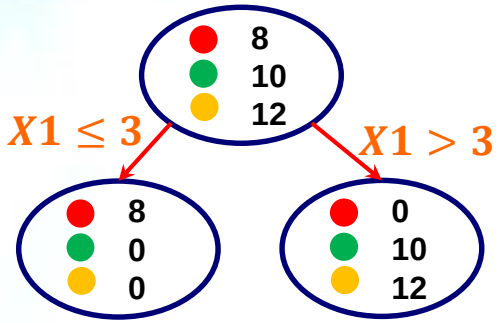


L'idéal est d'obtenir une impureté de Gini égale à 0, tout comme pour l'entropie.

1.4.4.3. Impureté de Gini (Gini Impurity):

Exemple précédent:

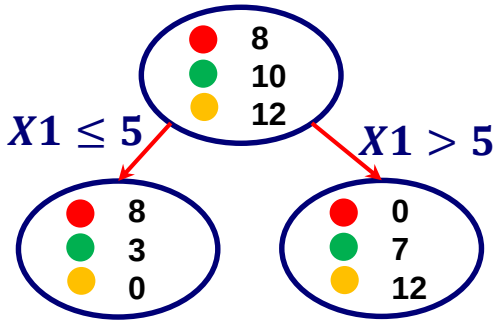
- **1er cas:** Séparation par rapport à **X1=3**



$$Gini\ Impurity = 1 - \left(\frac{8}{8}\right)^2 = 0$$

$$Gini\ Impurity = 1 - \left(\frac{10}{22}\right)^2 - \left(\frac{12}{22}\right)^2 = 0.4959$$

- **2ème cas:** Séparation par rapport à **X1=5**



$$Gini\ Impurity = 1 - \left(\frac{8}{11}\right)^2 - \left(\frac{3}{11}\right)^2 = 0.3967$$

$$Gini\ Impurity = 1 - \left(\frac{7}{19}\right)^2 - \left(\frac{12}{19}\right)^2 = 0.4654$$

- Pour savoir laquelle parmi les deux séparation est optimale, on calcule **Impureté de Gini totale**:

$$Total\ Gini = \sum_{i=1}^{2} p_i \times Gini_i$$

1. $$Total\ Gini = \frac{8}{30} \times 0 + \frac{22}{30} \times 0.4959 = 0.3637$$ ⇒ **Meilleure séparation**
2. $$Total\ Gini = \frac{11}{30} \times 0.3967 + \frac{19}{30} \times 0.4654 = 0.4402$$

1.4.4.4. Entropie vs. Impureté de Gini :

Entropie:

$$Entropy(S) = -\sum_{c \epsilon C} p(c)\log_2(p(c))$$

- Donne de meilleurs résultats.
- Nécessite des opérations logarithmiques, qui sont plus complexes et gourmandes en termes de ressources de calcul, notamment pour de grandes quantités de données.
- Préféré lorsque les classes sont non équilibrées

Impureté de Gini:

$$Gini\ Impurity = 1 - \sum_{i=1}^{k} p_i^2$$

- Plus rapide (moins coûteux).
- Critère par défaut dans des bibliothèques comme Scikit-learn.
- Préféré lorsqu'il y a un grand nombre de classes.

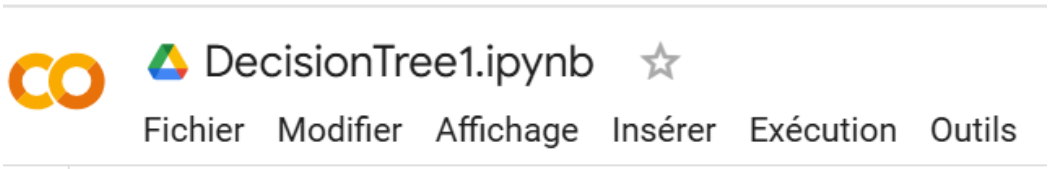
1.5. Application 1: Arbre de Décision pour la prédiction de prêt bancaire:

1.5.1. Objectif:

- Déterminer si un client est éligible pour un prêt bancaire en fonction de :
  - ***Revenu annuel*** *: faible, moyen, élevé*
  - ***Historique de crédit*** *: bon, mauvais*
  - ***Montant demandé*** *: faible, moyen, élevé*

1.5.2. Étape 1 : Préparer l'environnement

- Ouvrir Google Colab et créer un **Nouveau Notebook**.
  Renommer le **DecisionTree1.ipynb**
- Ajouter les bibliothèques nécessaires



```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, classification_report
import matplotlib.pyplot as plt
```

1.5.3. Étape 2 : Créer ou importer les données

- Créer un dataset fictif pour prédire si un client est éligible pour un prêt bancaire.

```
# Créer un dataset fictif
data = {
    "Revenu_annuel": ["faible", "moyen", "élevé", "faible", "moyen", "élevé", "moyen", "faible"],
    "Historique_credit": ["bon", "mauvais", "bon", "bon", "mauvais", "mauvais", "bon", "mauvais"],
    "Montant_demandé": ["faible", "moyen", "élevé", "faible", "moyen", "élevé", "moyen", "faible"],
    "Eligibilité": ["Non", "Non", "Oui", "Non", "Non", "Oui", "Oui", "Non"]
}
# Convertir en DataFrame
df = pd.DataFrame(data)
# Afficher les données
print(df)
```

```
  Revenu_annuel Historique_credit Montant_demandé Eligibilité
0        faible              bon          faible         Non
1         moyen          mauvais           moyen         Non
2         élevé              bon           élevé         Oui
```

1.5.4. Étape 3 : Prétraitement des données

- Convertir les variables catégorielles en numériques.
- Séparer les données en **features (X)** et **target (y)**.

"faible" → 0
"moyen" → 1
"élevé" → 2

```
# Encodage des données catégorielles en numériques
df_encoded = df.copy()
df_encoded["Revenu_annuel"] = df["Revenu_annuel"].map({"faible": 0, "moyen": 1, "élevé": 2})
df_encoded["Historique_credit"] = df["Historique_credit"].map({"mauvais": 0, "bon": 1})
df_encoded["Montant_demandé"] = df["Montant_demandé"].map({"faible": 0, "moyen": 1, "élevé": 2})
df_encoded["Eligibilité"] = df["Eligibilité"].map({"Non": 0, "Oui": 1})

# Séparer les features (X) et la cible (y)
X = df_encoded.drop("Eligibilité", axis=1)
y = df_encoded["Eligibilité"]

# Afficher les données transformées
print("Features (X) :\n", X)
print("Cible (y) complète :\n", y.values)
```

drop() permet de supprimer des colonnes ou des lignes.
"Eligibilité" est la colonne que tu veux supprime

```
Features (X) :
    Revenu_annuel  Historique_credit  Montant_demandé
0               0                  1                0
1               1                  0                1
2               2                  1                2
3               0                  1                0
4               1                  0                1
5               2                  0                2
6               1                  1                1
7               0                  0                0
Cible (y) complète :
 [0 0 1 0 0 1 1 0]
```

1.5.5. Étape 4 : Diviser les données en ensemble d'entraînement et de test

```
# Diviser les données en train (80%) et test (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

Remarque 1: `stratify=y` permet de garantir que la répartition des classes dans l'échantillon d'entraînement et l'échantillon de test **respecte la distribution initiale** des classes de y (surtout pour une distribution déséquilibrée des classes)

1.5.6. Étape 5 : Construire et entraîner un modèle d'arbre de décision

```
# Créer le modèle d'arbre de décision
model = DecisionTreeClassifier(criterion="entropy", random_state=42, max_depth=5)

# Entraîner le modèle
model.fit(X_train, y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(criterion='entropy', max_depth=5, random_state=42)
```

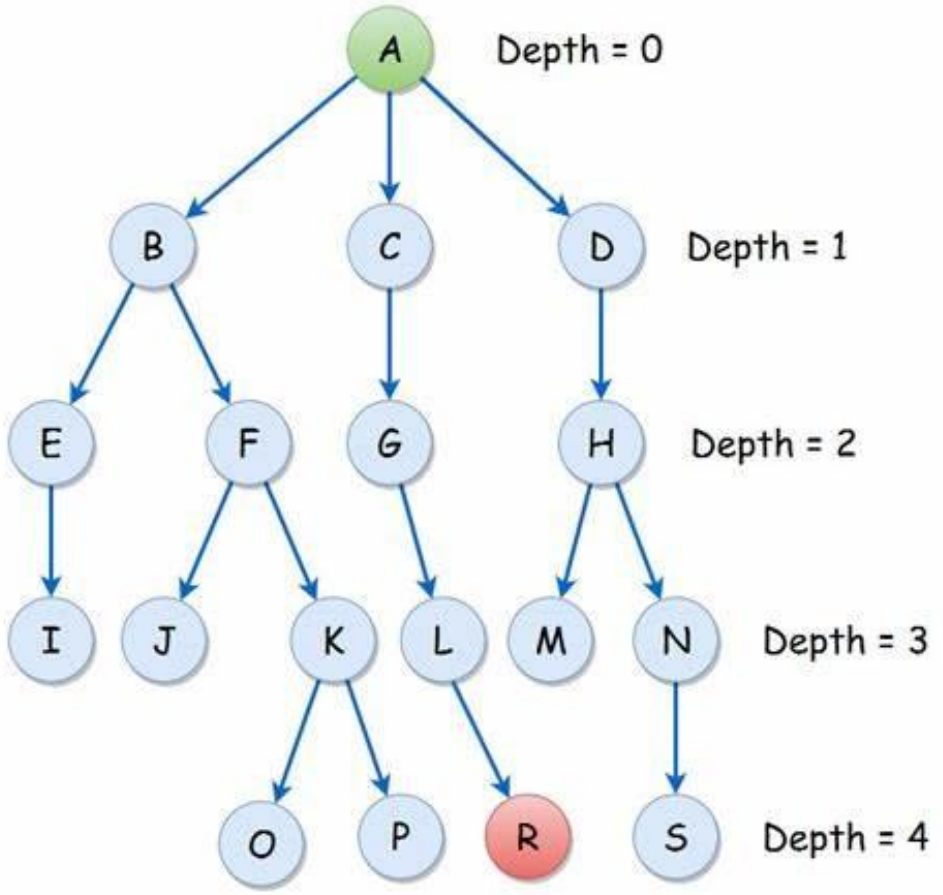
max_depth représente le nombre maximal de niveaux de l'arbre de décision

Remarque 2*: Vous pouvez utiliser le critère **d'Impureté de Gini** au lieu de **l'entropie**.*
Remplacer alors la première ligne par le code suivant :

```
model = DecisionTreeClassifier(criterion='gini', random_state=42, max_depth=5)
```

… Application 1

Remarque 3: **max_depth** représente le nombre maximal de niveaux de l'arbre de décision



Effet de max_depth sur l'apprentissage:

- **Si max_depth est trop petit → Sous-apprentissage (underfitting)**
  → L'arbre ne capture pas assez d'informations et les prédictions sont mauvaises.
  Exemple : max_depth=1 (juste une séparation)

- **Si max_depth est trop grand → Sur-apprentissage (overfitting)**
  → L'arbre apprend trop les détails du jeu d'entraînement et ne généralise pas bien.

1.5.7. Étape 6 : Évaluer le modèle (précision et rapport de classification)

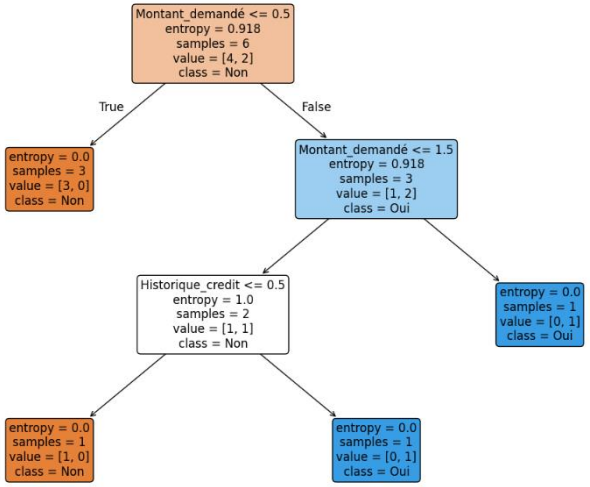
```
# Prédictions sur l'ensemble de test
y_pred = model.predict(X_test)

# Évaluer le modèle
accuracy = accuracy_score(y_test, y_pred)
print("Précision du modèle :", accuracy)
print("\nRapport de classification :\n", classification_report(y_test, y_pred))
```

```
Précision du modèle : 1.0

Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1

    accuracy                           1.00         2
   macro avg       1.00      1.00      1.00         2
weighted avg       1.00      1.00      1.00         2
```

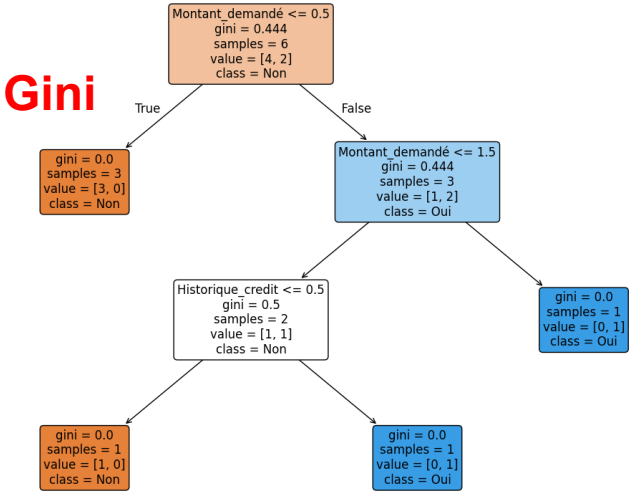
1.5.8. Étape 7 : Visualiser l'arbre de décision

```
# Visualiser l'arbre de décision
plt.figure(figsize=(15, 10))
plot_tree(model,
          feature_names=X.columns,  # Affiche les noms des features
          class_names=["Non", "Oui"],  # Nom des classes
          filled=True,  # Colore les noeuds en fonction des classes
          rounded=True,  # Rendre les noeuds arrondis pour une meilleure lisibilité
          fontsize=12)  # Taille de police pour une meilleure lisibilité
plt.show()
```

Cas de l'Entropie



Cas de l'Impureté de Gini



1.5.9. Étape 8 : Tester avec de nouvelles données

- Tester le modèle avec des exemples personnalisés.

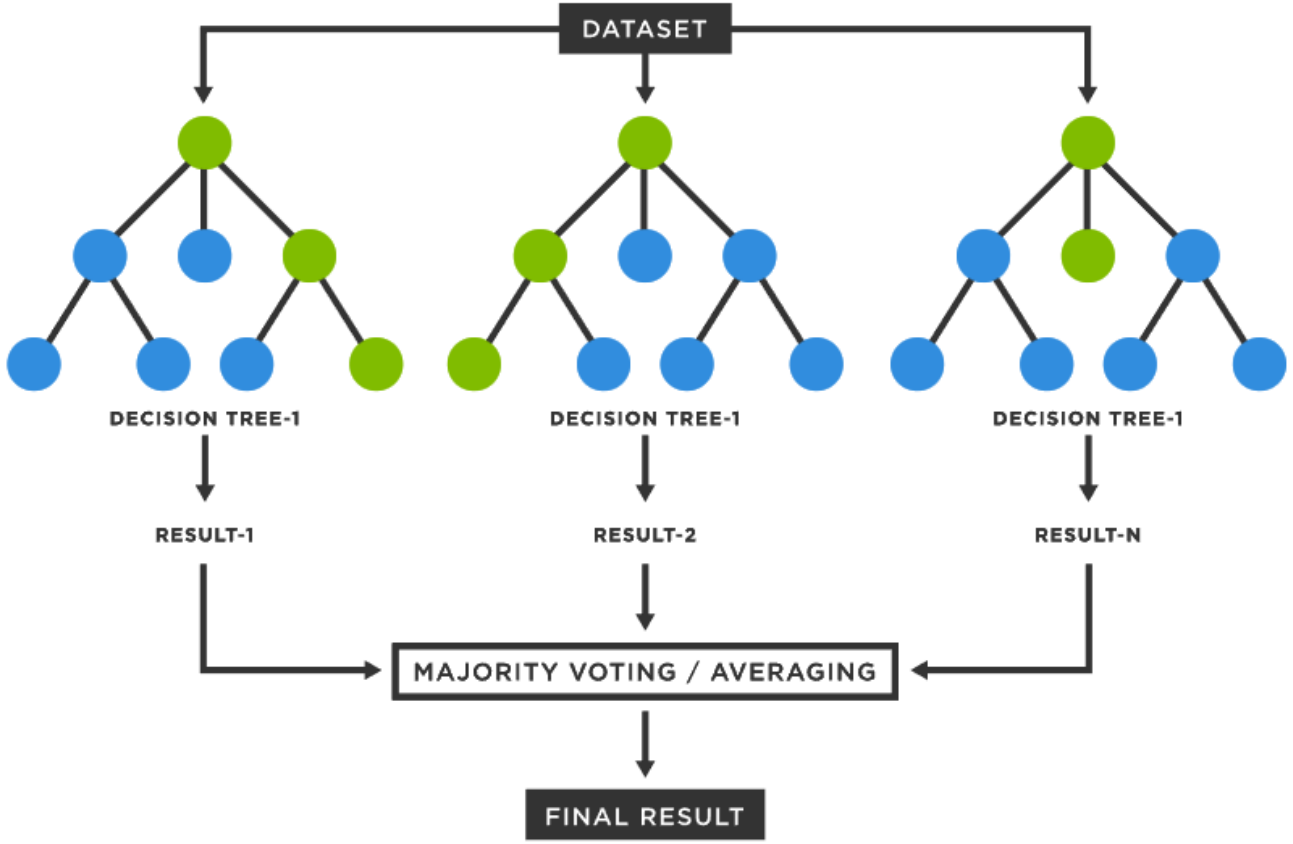
```
# Exemple de prédiction
nouveau_client = pd.DataFrame({
    "Revenu_annuel": [1],  # moyen
    "Historique_credit": [1],  # bon
    "Montant_demandé": [2]  # élevé
})

# Prédire pour le nouveau client
prediction = model.predict(nouveau_client)
print("Éligibilité du client :", "Oui" if prediction[0] == 1 else "Non")
```

```
Éligibilité du client : Oui
```

2. Forêt aléatoire (Random Forest)

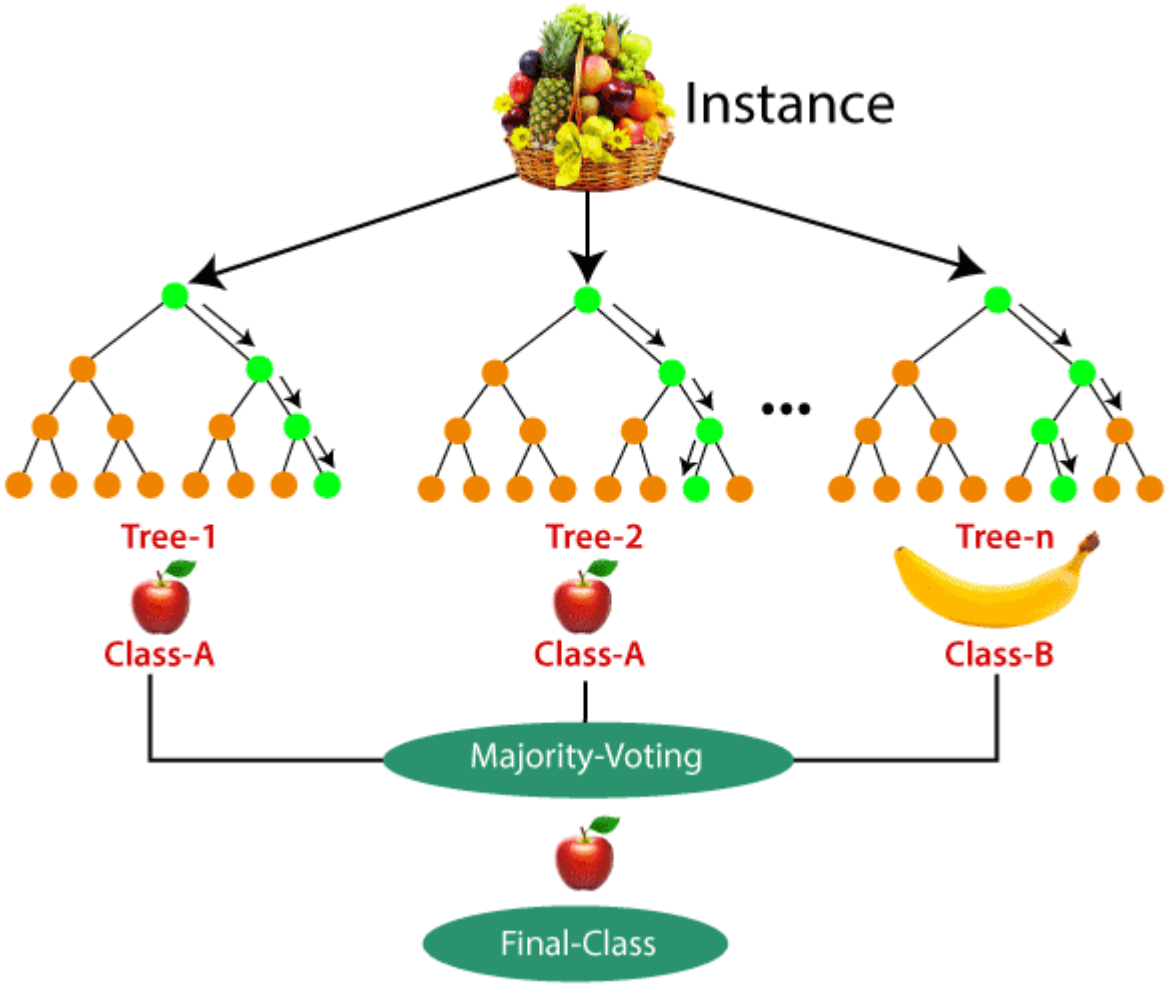
2.1. Définition

- Random Forest est un algorithme d'apprentissage supervisé utilisé pour des tâches de classification et de régression.
- Il fonctionne en construisant plusieurs arbres de décision (un ensemble) et en combinant leurs prédictions pour obtenir un résultat final robuste.



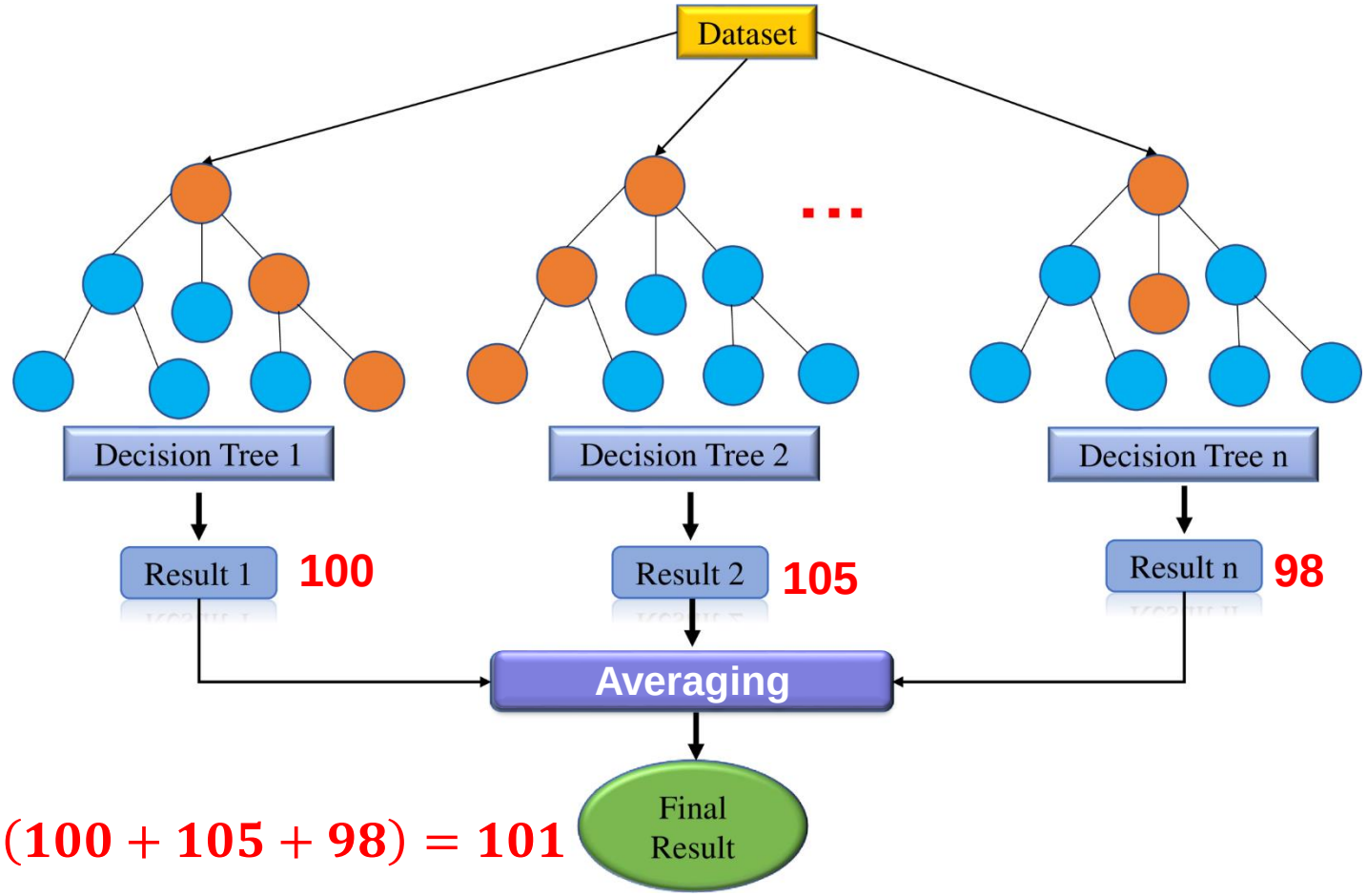
… Random Forest

- Exemple (**classification**): Un jeu de données d'images de fruits est donné au classificateur Random Forest, qui le divise en sous-ensembles pour chaque arbre de décision.
- Chaque arbre produit une prédiction,
- La décision finale est basée sur la majorité des résultats (**Voting**)
- Ici 2 pour la pomme et 1 pour la banane) Donc le résultat final de prédiction et « Pomme ».



… Random Forest

Exemple (**Regression**): Dans ce cas, Random Forest utilise **l'averaging** (moyenne) pour combiner les prédictions des différents arbres



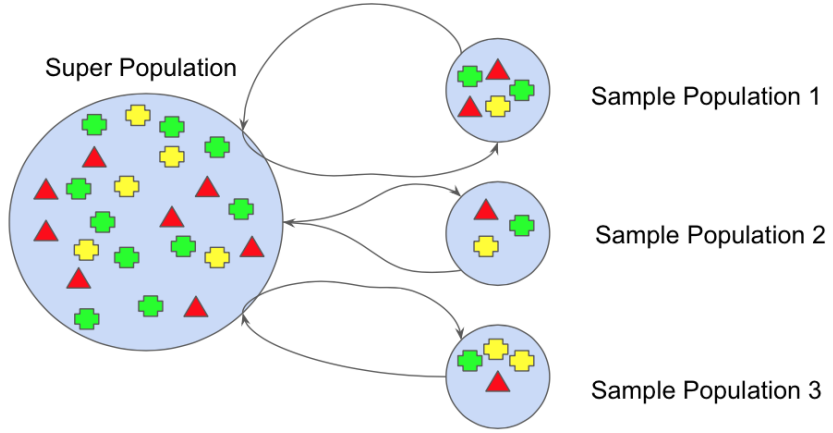
$$\textit{Prédiction finale} = \frac{1}{3}(100 + 105 + 98) = 101$$

2.2. Techniques d'échantillonnage

- Une **technique d'échantillonnage** consiste à sélectionner une partie d'un ensemble de données d'origine pour effectuer des analyses, des entraînements de modèles ou d'autres traitements.
- Techniques très utilisées: Bagging, Pasting
- **Bagging** (Bootstrap Aggregating) : Consiste à créer plusieurs sous-ensembles de données en échantillonnant **avec remplacement**. Chaque sous-ensemble peut donc contenir des doublons, et certains exemples du jeu de données d'origine peuvent être exclus.
- **Pasting** (échantillonnage sans remplacement): Chaque arbre est formé à partir d'un sous-échantillon tiré aléatoirement, mais sans remplacement. Cela signifie que chaque échantillon d'entraînement est utilisé une seule fois pour un arbre donné.
- **Le Bagging** est **beaucoup plus utilisé** que le **Pasting** dans la pratique, notamment dans le cadre des **forêts aléatoires**.

… Random Forest
Techniques d'échantillonnage

Aspect	Bagging	Pasting
Échantillonnage	Avec remplacement	Sans remplacement
Doublons	Possible dans les sous-ensembles	Aucun doublon dans un sous-ensemble
Utilisation	Jeux de données plus petits	Jeux de données plus grands



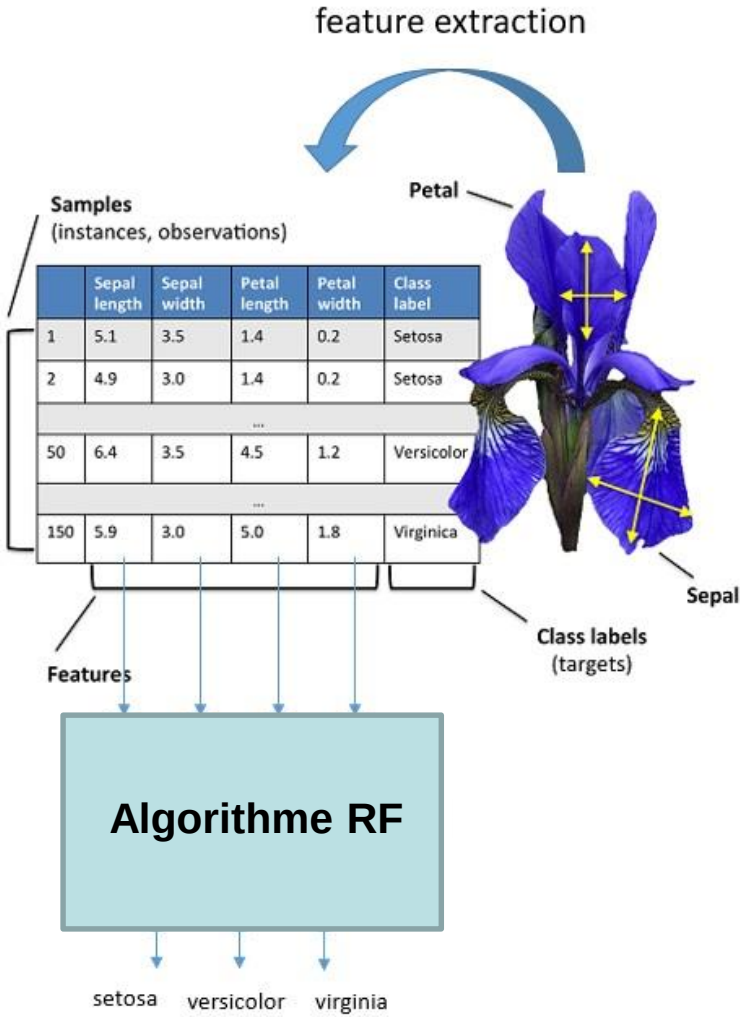
2.3. Application 2: Random Forest sur Iris:

2.3.1. Objectif:

Utilisez le jeu de données Iris, un jeu de données classique pour la classification.
Ce jeu contient 150 exemples de fleurs répartis en 3 espèces (Setosa, Versicolor, Virginica), avec 4 caractéristiques pour chaque fleur : la longueur et la largeur des sépales et des pétales.

Etapes :

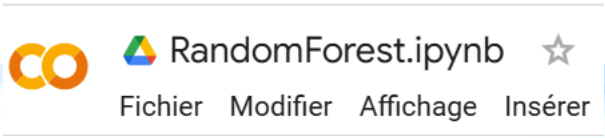
1. Charger le jeu de données.
2. Appliquer un modèle **Random Forest** pour effectuer une classification.
3. Évaluer la performance du modèle.
4. Analyser les résultats obtenus.



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

2.3.2. Nouveau Notebook:

- Ouvrir Google Colab et créer un **Nouveau Notebook**.
  Renommer le **RandomForest.ipynb**



2.3.3. Code à utiliser dans Google Colab: …….. 1/3

```
# Importation des bibliothèques nécessaires
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns

# 1. Charger le jeu de données Iris
iris = load_iris()
X = iris.data  # Caractéristiques
y = iris.target  # Étiquettes

# 2. Diviser les données en train/test (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

2.3.3. Code à utiliser dans Google Colab: …….. 2/3

```
# 3. Créer et entraîner le modèle Random Forest
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

# 4. Prédire sur l'ensemble de test
y_pred = rf.predict(X_test)

# 5. Évaluer les performances du modèle
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# 6. Afficher un rapport de classification
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

2.3.3. Code à utiliser dans Google Colab: …….. 2/3

$$Acuracy = \frac{Number\ of\ Correct\ Predictions\ (\sum TP + \sum TN)}{Total\ Population}$$

```
Accuracy: 1.00
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        10
           1       1.00      1.00      1.00         9
           2       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

classes du modèle:
0: Setosa
1: Versicolor
2: Virginica

Nombre d'échantillons de chaque classe dans y_test

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

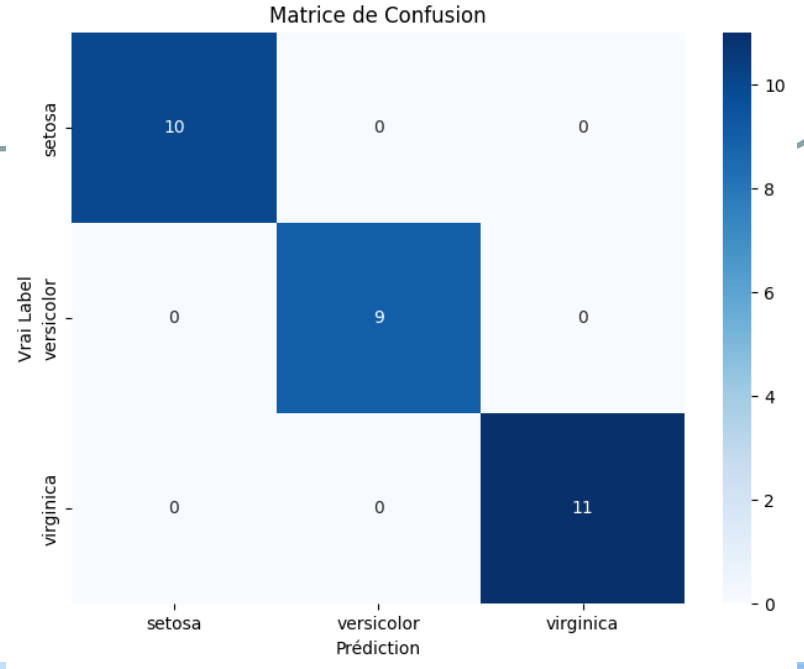
Mesure combien de prédictions positives sont correctes

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

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

2.3.3. Code à utiliser dans Google Colab: …….. 3/3

```
# 7. Matrice de confusion
conf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=iris.target_names, yticklabels=iris.target_names)
plt.title("Matrice de Confusion")
plt.ylabel('Vrai Label')
plt.xlabel('Prédiction')
plt.show()
```



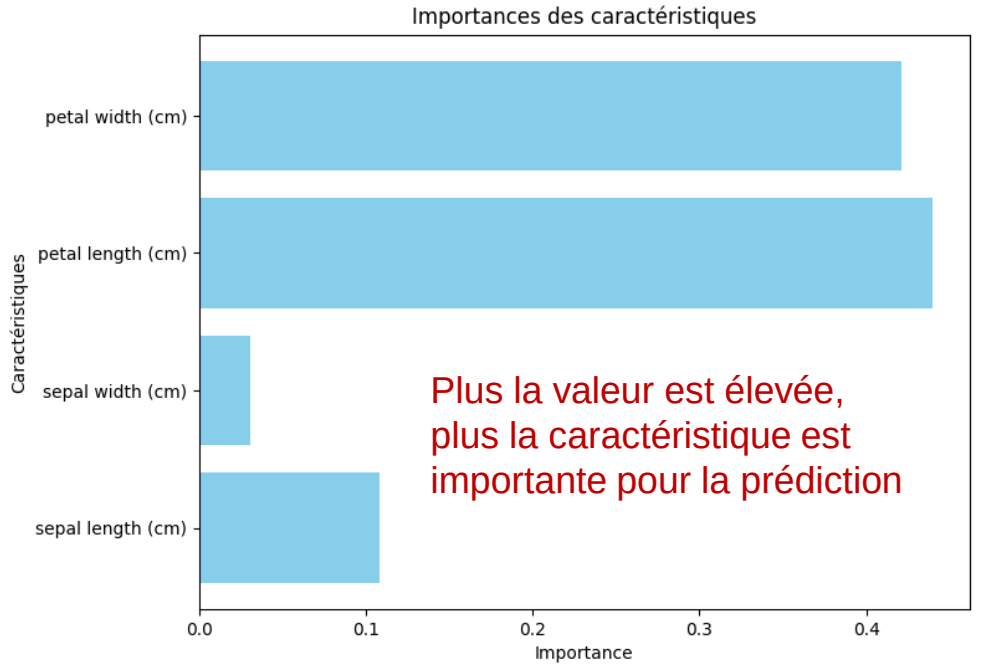
Ici, la matrice est **diagonale**, cela signifie que le modèle **classifie parfaitement** les échantillons.

Avantages de la matrice de confusion

- Permet d'analyser **les erreurs du modèle** (ex. confusion entre *Versicolor* et *Virginica*).
- Plus **détaillée** qu'une simple **précision globale**.
- Aide à améliorer le modèle en comprenant ses **points faibles**.

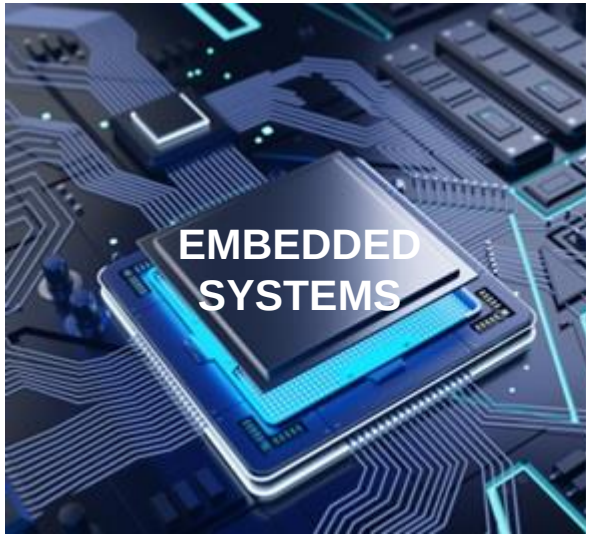
2.3.3. Code à utiliser dans Google Colab: …….. 3/3

```
# 8. Visualisation des importances des caractéristiques
importances = rf.feature_importances_
features = iris.feature_names
plt.figure(figsize=(8, 6))
plt.barh(features, importances, color='skyblue')
plt.title("Importances des caractéristiques")
plt.xlabel("Importance")
plt.ylabel("Caractéristiques")
plt.show()
```



Ce type de visualisation est utile pour :
Comprendre le modèle : Identifier quelles caractéristiques influencent le plus la décision.

- **Feature Selection** : Éliminer les caractéristiques peu importantes pour simplifier le modèle.
- **Interprétation** : Expliquer les résultats à des non-experts.



FIN !
Questions ?