

# Machine Learning

1

Filière Ingénierie Logicielle et Systèmes Intelligents (4A-ILSI)

ENSAM-Meknès

A.AHMADI

2025 / 2026

# Machine Learning

## Syllabus

2

### 1. Introduction au Machine Learning

- Définition et importance du Machine Learning.
- Types d'apprentissage : supervisé, non supervisé, semi-supervisé, par renforcement.

### 2. Préparation des Données

- Collecte, nettoyage et prétraitement des données.
- Techniques de normalisation et de réduction de dimensionnalité.

### 3. Régression linéaire simple et multiple

- Modèle et représentation
- La fonction coût
- Algorithme descente de gradient (Gradient Descent)
- Normalisation des données (features scaling)
- Régression polynomiale
- Résolution matricielle : équation normale
- Implémentation

### 4. Régression logistique

- Classification binaire
- Formalisation et fonction coût
- Descente de gradient
- Classification : cas multi-classes

### 5. Techniques d'Evaluation et Validation de Modèles

- Méthodes de validation croisées.
- Grid search
- Biais-Variance Tradeoff
- Régularisation L1 et L2
- Mesures de performance : précision, rappel, F1-score, AUC-ROC, etc.

### 6. Arbres de Décision et Forêts aléatoires

- Le problème sur/sous apprentissage
- Bias et variance
- Régularisation

### 7. Apprentissage Supervisé

- Clustering (K-Means, Clustering Hierarchique)

### 8. Réseaux de Neurones : Introduction

- Introduction aux réseaux de neurones artificiels.
- Modèle et représentation
- La fonction coût
- Retropagation du gradient (Backpropagation)

### 9. Applications Pratiques et Etudes de Cas

# Machine Learning

## Prérequis



1. Algèbre linéaire
2. Notions de Probabilité et de Statistiques
3. Programmation Python :
  - i. Site important : <https://learnpython.org/>
  - ii. Tutorial important : <https://docs.python.org/3/tutorial/>
  - iii. Librairies (Numpy, Pandas, Mathplotlib)

<https://colab.research.google.com/github/ageron/handson-ml3/blob/main/index.ipynb#scrollTo=Fss-TohBKDs5>

# Machine Learning

## Documentation

4

1. "*Artificial Intelligence: A Modern Approach*", Stuart Russell and Peter Norvig, 4th edition (Pearson).
2. "*Stephen Marsland's Machine Learning: An Algorithmic Perspective*", Stephen Marsland, 2nd edition (Chapman & Hall).
3. "*Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*", Aurélien Géron, 3d edition, (O'Reilly Media).
4. <https://www.coursera.org/learn/machine-learning/>
5. [https://scikit-learn.org/stable/user\\_guide.html](https://scikit-learn.org/stable/user_guide.html)
6. <https://www.dataquest.io/blog/>
7. "*Deep Learning with Python*", François Chollet , 2nd edition (Manning).

# Machine Learning

## Environnements de travail

5

### Environnement de travail du ML :

- Outils, plateformes et interfaces pour apprendre, coder et expérimenter les projets ML.
- Simplifie l'installation des bibliothèques.
- Favorise la reproductibilité des projets.
- Permet de collaborer facilement.
- Offre des interfaces adaptées à chaque profil (débutant, avancé, chercheur).
- Un bon environnement = gain de temps + moins d'erreurs.

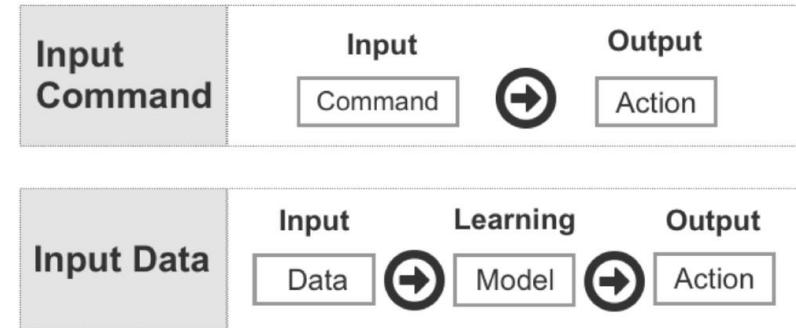
Environnement	Type	Niveau	Avantages
Jupyter Notebook	Local	Débutant	Intuitif, visuel
Google Colab	Cloud	Tous	GPU gratuit, partage facile
VS Code	Local	Intermédiaire	Flexible, léger
Anaconda	Local	Tous	Gestion d'environnement
Docker	Local/Cloud	Avancé	Reproductibilité, déploiement

**Kaggle Notebooks (en ligne) :**  
- Accès direct à des datasets  
- Communauté active  
- Exécution dans le cloud

- Commencer avec **Google Colab** ou **Jupyter Notebook**.
- Utiliser **Miniconda** (léger par rapport à Anaconda) pour gérer les environnements.
- Versionner les projets avec **Git**.
- Documenter chaque étape dans un **notebook**.

## I- Définition

- Le Machine Learning (ML) est un nouvel effort pour formaliser la connaissance humaine dans des systèmes automatisés.
- "Le ML donne aux ordinateurs la capacité d'apprendre sans être explicitement programmés" ([Arthur Samuel, 1959](#)).
- On dit qu'un programme informatique apprend de l'expérience E par rapport à une tâche T et à une mesure de performance P, si sa performance sur T, mesurée par P, s'améliore avec l'expérience E ([Tom Mitchell, 1997](#)).
- Le programme apprend à partir d'un ensemble d'entraînement. Il en extrait des modèles/statistiques pour faire des prédictions.
- Composants clés :
  - Tâche (T) : Ce qu'on veut accomplir.
  - Expérience (E) : Les données d'entraînement.
  - Performance (P) : Comment on évalue le modèle.
- Utilité :
  - Réduire la complexité du code.
  - S'adapter aux environnements changeants.
  - Résoudre des problèmes trop complexes pour l'humain.
  - Aider à analyser de grandes quantités de données.



[Source : Oliver Theobald](#)

## II- Applications du ML



- Application à une grande variété de tâches, (des images et du texte à l'audio, la fraude, la recommandation et même les jeux). Chaque tâche a des techniques adaptées, souvent basées sur des réseaux de neurones ou des modèles plus classiques :

### 1. Analyse et classification d'images

- Tâche : Identifier ou classer des objets dans une image.
- Exemples :
  - Classer des produits sur une chaîne de production → *Image classification*.
  - Déetecter des tumeurs dans des IRM → *Image segmentation*.
- Techniques : *CNN, Transformers*.

### 2. Traitement du langage naturel (NLP)

- Tâches :
  - *Classer* des articles ou commentaires → Text classification.
  - *Résumer* des documents → Text summarization.
  - Créer des *assistants vocaux* / chatbots → NLU, question-answering.
- Techniques : *RNN, CNN, Transformers* (très performants)

## II- Applications du ML



### 3. Prédiction de valeurs numériques

- Exemple : Prévoir les revenus d'une entreprise
- Tâche : Régression
- Techniques :
  - *Régression linéaire / polynomiale*
  - *SVM de régression, Forêts aléatoires*
  - *Réseaux de neurones*
  - *RNN, CNN, Transformers* (pour données séquentielles).

### 4. Reconnaissance vocale

- Tâche : Comprendre des commandes vocales.
- Techniques : *RNN, CNN, Transformers* (pour traiter l'audio, séquence longue).

### 5. Détection de fraude

- Tâche : Anomaly detection.
- Techniques : *Isolation Forest, GMM, Autoencodeurs*.

### 6. Segmentation de clients

- Tâche : Clustering
- Utilité : Adapter le marketing à différents profils
- Techniques : *K-means, DBSCAN...*

## II- Applications du ML



### 7. Visualisation de données complexes

- Tâche : Réduction de dimension.
- But : Représenter visuellement des données en haute dimension.

### 8. Systèmes de recommandation

- Tâche : Suggérer des produits à l'utilisateur
- Techniques :
  - *Réseaux de neurones* entraînés sur les historiques d'achats.

### 9. Jeux et agents intelligents

- Tâche : *Reinforcement Learning* (RL).
- Exemple : Créer un bot qui maximise ses gains dans un jeu (ex : AlphaGo).
- Principe : L'agent apprend à agir pour maximiser ses récompenses.

## III- Types de Machine learning



### 1. Apprentissage Supervisé

- Le modèle apprend à partir d'exemples labellisés. On lui donne une entrée  $X$  et une sortie correcte  $Y$ . Il apprend à prédire  $y$  à partir de  $x$ .
- Types de tâches :
  - *Classification* : Prédire une catégorie (ex : maladie ou non, type de fleur, spam ou ham...)
  - *Régression* : Prédire une valeur numérique (ex : température, prix d'un bien...).
- Exemples pratiques :
  - Diagnostiquer une maladie à partir de symptômes.
  - Prédire le prix d'une maison.
  - Reconnaître des visages ou objets.
- Méthodes courantes :
  - *Arbres de décision.*
  - *Régressions linéaires/logistiques.*

## III- Types de Machine learning

11

### 2. Apprentissage Non Supervisé

- Le modèle n'a pas de sortie à apprendre. Il découvre seul des structures ou relations dans les données.
- Types de tâches :
  - *Clustering* : regrouper les objets similaires (ex : regrouper les clients en segments de comportement).
  - *Réduction de dimension* : compresser les données tout en gardant les informations essentielles (ex : visualisation, nettoyage).
- Exemples pratiques :
  - *Détection de fraudes (anomalies)*
  - *Segmentation marketing*
  - *Compréhension de données textuelles (groupes de mots, sujets)*
- Méthodes courantes :
  - *K-means*
  - *DBSCAN*
  - *PCA (Analyse en Composantes Principales)*

## III- Types de Machine learning

12

### 3. Apprentissage par Renforcement

- L'algorithme apprend par *essais-erreurs* dans un *environnement dynamique*. Il prend des *décisions*, observe une *récompense*, puis ajuste ses actions pour *maximiser les gains* à long terme.
- Concepts clés :
  - *Agent* : prend des décisions
  - *Environnement* : contexte dans lequel agit l'agent
  - *Récompense* : retour donné selon la qualité de l'action
- Exemples pratiques :
  - *AlphaGo* (jeu de Go)
  - *Robotique autonome* (naviguer dans un espace)
  - *Optimisation de stratégie de trading ou de publicité*
- Méthodes courantes :
  - *Q-learning*
  - *Deep Q-Networks*
  - *Policy Gradients*

## III- Types de Machine Learning

13

### 4. Autres approches hybrides

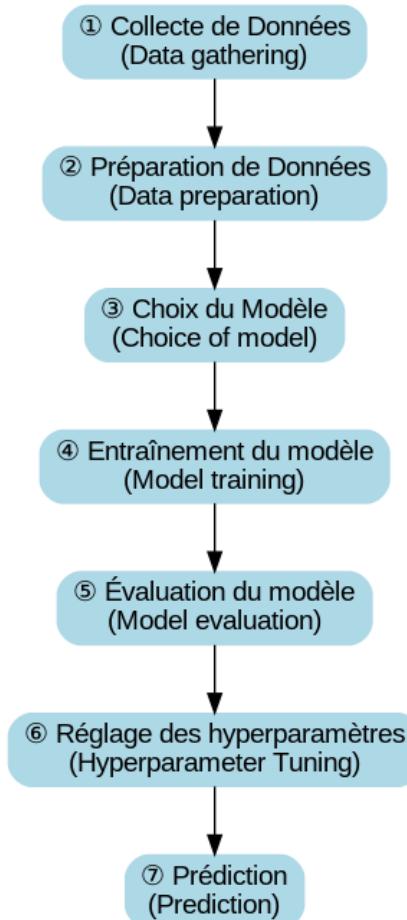
- ML évolue et des approches *hybrides* apparaissent pour des cas particuliers.
- Exemples :
  - **Semi-supervisé** : on a peu de données labellisées, et beaucoup de données non étiquetées. Le modèle s'en sert pour apprendre mieux.
  - **Auto-supervisé** : on crée automatiquement des labels depuis les données (très utilisé dans GPT, BERT, etc.).
  - **Apprentissage fédéré** : les données restent sur l'appareil (ex : téléphone), mais le modèle s'entraîne de manière distribuée et sécurisée (important pour la vie privée).

Objectif	Type de ML	Exemple
Prédire une sortie connue	Supervisé	Prédire un diagnostic
Explorer / structurer les données	Non Supervisé	Segmentation client
S'adapter en environnement dynamique	Renforcement	Jouer à un jeu

## IV- Les 7 étapes du Machine Learning

14

Un projet Machine Learning passe par 7 étapes principales :



Etape	Objectif principal
1. Collecte de données	Obtenir des données pertinentes
2. Préparation des données	Nettoyer et structurer les données
3. Choix du modèle	Sélectionner l'algorithme adapté
4. Entraînement	Apprendre à partir des données
5. Évaluation	Mesurer la performance réelle
6. Réglage des hyperparamètres	Optimiser les réglages du modèle
7. Prédiction	Utiliser le modèle pour des cas réels

## IV- Les 7 étapes du Machine Learning

15

### 1. Collecte de données (Data Gathering)

On rassemble les données nécessaires pour entraîner un modèle :

- Les données peuvent provenir de capteurs, bases de données, fichiers CSV, API, ou être extraites du web.
- Elles doivent être représentatives du problème à résoudre.
- Plus les données sont variées, riches et bien structurées, plus le modèle aura de chances d'apprendre efficacement.

Exemple : Pour prédire la consommation électrique, on collecte des données horaires, météo, historiques de consommation, etc.

### 2. Préparation des données (Data Preparation)

On nettoie et transforme les données pour qu'elles soient exploitable par un algorithme.

- Nettoyage : suppression des doublons, gestion des valeurs manquantes, correction des erreurs.
- Transformation : normalisation, standardisation, encodage des variables catégorielles.
- Séparation : division en ensembles d'entraînement, de validation et de test.

**NB:** Cette étape est souvent la plus longue et la plus critique : un bon modèle commence par de bonnes données.

## IV- Les 7 étapes du Machine Learning

16

### 3. Choix du modèle (Choice of Model)

On sélectionne l'algorithme le plus adapté au type de problème :

- Régression : prédire une valeur continue (ex. prix, température).
- Classification : prédire une catégorie (ex. spam ou non spam).
- Clustering : regrouper des données similaires sans étiquettes.
- Séries temporelles, reconnaissance d'image, traitement du langage : chaque domaine a ses modèles spécialisés.

Le choix dépend du type de données, du volume, de la complexité et des objectifs du projet.

### 4. Entraînement du modèle (Model Training)

Apprendre à partir des données d'entraînement.

- Le modèle ajuste ses paramètres internes pour minimiser une fonction de perte.
- Il découvre des patterns ou des relations entre les variables.
- L'entraînement peut être rapide (régression linéaire) ou très long (réseaux de neurones profonds).

C'est ici que le modèle **apprend** à faire des prédictions.

## IV- Les 7 étapes du Machine Learning

17

### 5. Évaluation du modèle (Model Evaluation)

On mesure la performance du modèle sur des données qu'il n'a jamais vues.

- On utilise l'ensemble de **test** pour évaluer la capacité du modèle à généraliser.
- Métriques courantes : Précision (*accuracy*) ; Rappel (*recall*) ; F1-score ; RMSE (erreur quadratique moyenne) ; AUC (aire sous la courbe ROC).

Un bon modèle ne doit pas juste bien fonctionner sur les données d'entraînement, mais aussi sur des cas *réels*.

### 6. Réglage des hyperparamètres (Hyperparameter Tuning)

On optimise les paramètres externes au modèle pour améliorer ses performances.

- Contrairement aux paramètres internes (poids, biais), les hyperparamètres sont définis avant l'entraînement.
- Exemples d'hyperparamètres : taux d'apprentissage, nombre de couches, taille du batch, profondeur d'arbre.
- Méthodes : Grid Search (test systématique) ; Random Search (test aléatoire) ; Bayesian Optimization (approche intelligente)

Un bon réglage peut faire la différence entre un modèle moyen et un modèle excellent.

## IV- Les 7 étapes du Machine Learning

18

### 7. Prédiction (Prediction)

On utilise le modèle entraîné pour faire des prédictions sur de nouvelles données.

- Le modèle est déployé dans une application, un site web, ou un système embarqué.
- Il peut fonctionner en temps réel ou par lots.
- Les prédictions peuvent être utilisées pour prendre des décisions, automatiser des tâches, ou générer des recommandations.

Exemples : prédire si un client va acheter un produit, détecter une anomalie, traduire un texte.

## I- Introduction

19

- La **préparation des données** est une étape fondamentale (**80%**) dans tout projet de Machine Learning.
- Les données doivent être *propres, cohérentes et bien structurées*. Cette phase intervient après la collecte et consiste à transformer les données brutes en un format exploitable par les modèles d'apprentissage.
- Les données réelles sont souvent imparfaites. Elles peuvent contenir des *valeurs manquantes, des doublons, des incohérences, Des formats hétérogènes*.
- Opérations clés de la préparation des données :
  - **Nettoyage** : suppression des *erreurs et des anomalies*.
  - **Transformation** : *normalisation, encodage, mise à l'échelle*.
  - **Sélection** : choix des *variables pertinentes*.
  - **Partition ou Séparation** : *division en ensembles d'entraînement, de validation et de test*.

**NB** : Sans une préparation rigoureuse, même les meilleurs algorithmes risquent de produire des résultats biaisés ou inefficaces.

## II- Nettoyage des données (Data Cleaning)

20

- Dans les systèmes réels, les données brutes sont rarement parfaites. Elles peuvent contenir :
  - des valeurs *manquantes* (ex. champs vides).
  - des *doublons*.
  - des *erreurs de saisie ou de format*.
  - des *incohérences logiques* (ex. âge négatif).
  - des *outliers* (valeurs extrêmes/aberrantes non représentatives).
- Un modèle entraîné sur des données non nettoyées risque de :
  - produire des résultats *biaisés*,
  - *sur-apprendre des erreurs*,
  - *généraliser mal* sur de nouvelles données.

# Chap 2      Préparation des Données

## II- Nettoyage des données

21

Exemple : Nettoyage progressif d'un jeu de données

```
import pandas as pd  
import numpy as np
```

# 1. Exemple de données avec valeurs manquantes

```
df = pd.DataFrame({  
    'nom': ['Ali', 'Sara', 'Youssef', 'Nada'],  
    'age': [25, np.nan, 30, np.nan],  
    'ville': ['Fès', 'Rabat', np.nan, 'Casablanca']  
})  
print("Données initiales :")  
print(df)
```

# Solution 1 : Supprimer les lignes incomplètes

```
df_cleaned = df.dropna()  
print("Données après suppression des lignes incomplètes :")  
print(df_cleaned)
```

Données initiales :

	nom	age	ville
0	Ali	25.0	Fès
1	Sara	NaN	Rabat
2	Youssef	30.0	NaN
3	Nada	NaN	Casablanca

Données après suppression des lignes incomplètes

	nom	age	ville
0	Ali	25.0	Fès

## II- Nettoyage des données

22

Exemple : Nettoyage progressif d'un jeu de données

# **solution 2** : Imputer les valeurs manquantes

```
df['age'].fillna(df['age'].mean(), inplace=True)  
df['ville'].fillna('Inconnue', inplace=True)
```

**NB** : "inplace=True" permet le remplacement des valeurs manquantes directement dans le DataFrame df, sans créer de copie. Sinon, on aurait pu faire :

```
df['age'] = df['age'].fillna(df['age'].mean())
```

## III- Transformation des données (Data Transformation)

23

- Avant de confier les données à un algorithme, il faut les transformer pour qu'elles soient *compréhensibles*, *cohérentes* et *exploitables*. C'est ce qu'on appelle la *transformation* des données.
- Les algorithmes de Machine Learning sont sensibles à la forme et à l'échelle des données. Les transformations permettent de :
  - *Uniformiser* les valeurs.
  - Réduire les *biais*.
  - Accélérer l'apprentissage.
  - Améliorer la *précision*.

### 1- Mise à l'échelle (Scaling)

- La *mise à l'échelle* des variables numériques évite que certaines colonnes dominent les autres à cause de leur amplitude/valeur.  
Exemple : une colonne "revenu" allant de 4000 à 100000 DH et une colonne "âge" de 18 à 90. Sans mise à l'échelle, le modèle va accorder trop d'importance au *revenu*.
- Les techniques de mise à l'échelle permettent de ramener les valeurs numériques dans une *plage cohérente*.

# Chap 2 Préparation des Données

## III- Transformation des données

24

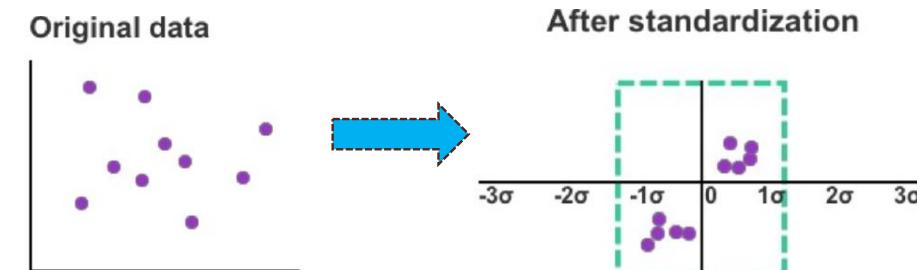
### 1- Mise à l'échelle (Scaling) : 2 Méthodes :

- Méthode 1: **Standardisation** : centrage-réduction (*moyenne = 0*, *écart-type = 1*)

Pour chaque valeur  $x_i$  dans une colonne  $X$ , la transformation est :

$$x_i^{\text{scaled}} = \frac{x_i - \mu}{\sigma} \quad \text{où :}$$

$x_i$  : valeur originale,  
 $\mu$  : moyenne de la colonne,  
 $\sigma$  : écart-type de la colonne et  
 $x_i^{\text{scaled}}$  : valeur transformée.



Source : Oliver Theobald

**NB** : La standardisation est particulièrement utile pour les algorithmes sensibles aux *distances* ou aux *gradients* (ex. *SVM*, *régression logistique*, *réseaux de neurones*).

En Python  
(scikit-learn) :

```
from sklearn.preprocessing import StandardScaler
import numpy as np
X = np.array([[10], [20], [30]])
scaler_std = StandardScaler()
X_scaled = scaler_std.fit_transform(X)
print('X_scaled=\n', X_scaled)
```

```
x_scaled=
[[-1.22474487]
 [ 0.        ]
 [ 1.22474487]]
```

# Chap 2 Préparation des Données

## III- Transformation des données

25

### 1- Mise à l'échelle (Scaling)

- Méthode 2 : **Normalisation (MinMax Scaling)**. C'est une mise à l'échelle linéaire des données. Elle transforme chaque valeur pour qu'elle soit comprise dans une plage  $[Min, Max]$  (Ex.  $[0, 1]$  ou  $[-1, 1]$ ).
- Cette transformation conserve la distribution des données mais réduit leur amplitude.
- Elle est utile pour les algorithmes sensibles aux *distances* ou aux *valeurs absolues*, comme les réseaux de neurones, KNN, ou algorithmes de clustering.
- Pour chaque valeur  $x_i$  dans une colonne  $X$ , la transformation est :

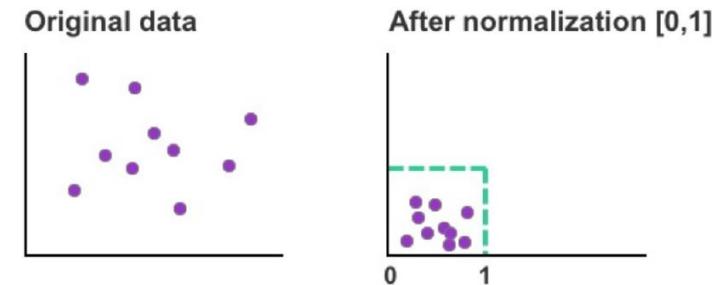
$$x_i^{\text{scaled}} = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}} \times (\text{Max} - \text{Min}) + \text{Min}, \text{ où :}$$

$x_i$  : valeur originale

$x_{\min}$  : valeur minimale de la colonne

$x_{\max}$  : valeur maximale de la colonne

$x_i^{\text{scaled}}$  : valeur transformée entre Min et Max



```
from sklearn.preprocessing import MinMaxScaler
X = [[100], [200], [300]]
# MinMax scaling.
scaler_mm = MinMaxScaler() # par défaut c'est entre 0 et 1
X_mm = # sinon scaler_mm = MinMaxScaler(feature_range=(10, 20))
X_mm = scaler_mm.fit_transform(X)
print('X_mm =\n', X_mm)
```

# Chap 2 Préparation des Données

## III- Transformation des données

26

### 2- Encodage des variables catégorielles

- Les algorithmes ne comprennent pas les mots ou valeurs textuelles. Il faut les transformer en chiffres, mais intelligemment, pour ne pas introduire de ***faux ordres***.
- Généralement 2 techniques d'encodage catégoriel : *Label Encoding* (chaque catégorie devient un entier) et *One-Hot Encoding* (création de colonnes binaires).
- Méthode 1 : ***Label Encoding*** : simple mais peut induire un ordre artificiel.

`labelEncoder()` (de scikit-learn) identifie toutes les catégories uniques dans la colonne, les trie par ordre alphabétique (lexicographique), puis il attribue à chaque catégorie un entier croissant à partir de 0.

```
from sklearn.preprocessing import LabelEncoder
import pandas as pd
df = pd.DataFrame({'couleur': ['rouge', 'bleu', 'vert']})
# Label Encoding
lab_enc = LabelEncoder()
df['couleur_encoded'] = lab_enc.fit_transform(df['couleur'])
print('df[couleur_encoded]=\n', df['couleur_encoded'])
```

```
df[couleur_encoded] =
 0    1
1    0
2    2
Name: couleur_encoded, dtype: int64
```

**NB** : Ce codage introduit un **ordre numérique** entre les catégories, ce qui peut être interprété à tort par certains algorithmes (*régression linéaire, SVM* ). Si les catégories n'ont aucun ordre logique, il vaut mieux utiliser ***OneHotEncoder***.

## III- Transformation des données

### 2- Encodage des variables catégorielles

- Méthode 2 : OneHotEncoder :

- Lorsqu'une variable contient des **catégories textuelles** (ex. : "rouge", "bleu", "vert"), *One-Hot Encoding* crée **une colonne pour chaque catégorie**. Chaque ligne contient un **1** dans la colonne correspondant à sa catégorie, et **0** dans les autres.
- Contrairement à *LabelEncoder*, *One-Hot* n'introduit pas d'ordre artificiel entre les catégories. Il est idéal pour les modèles sensibles aux distances (ex. *KNN*, *régression linéaire*, *réseaux de neurones*).

```
# One-Hot Encoding (en utilisant scikit-learn)
from sklearn.preprocessing import OneHotEncoder
import pandas as pd

# Données brutes
df = pd.DataFrame({'couleur': ['rouge', 'bleu', 'vert', 'rouge']}) # rouge 2 fois

# Initialisation de l'encodeur avec le bon paramètre
encoder = OneHotEncoder(sparse_output=False, dtype=int)

# Transformation
encoded = encoder.fit_transform(df[['couleur']])

# Conversion en DataFrame pour visualiser
encoded_df = pd.DataFrame(encoded, columns=encoder.get_feature_names_out(['couleur']))
print(encoded_df)
```

- Matrice Creuse (**sparse**) : on stocke en mémoire, seulement les 1 et leurs positions.
- Matrice Dense (**dense**) : on stocke les 1 et les 0.

	couleur_bleu	couleur_rouge	couleur_vert
0	0	1	0
1	1	0	0
2	0	0	1
3	0	1	0

# Chap 2 Préparation des Données

## III- Transformation des données

28

### 2- Encodage des variables catégorielles

- Méthode 2 : *OneHotEncoder* :

Inconvénient : Exemple : 10 000 codes postaux → 10 000 colonnes (grande matrice creuse → inefficace)

**Solution 1** : On utilise l'*Ordinal Encoding*, où l'on remplace chaque catégorie par un entier unique, mais cela peut fausser l'apprentissage si le modèle croit qu'il y a une *hiérarchie*.

**Solution 2** : On utilise le *Target Encoding*

- Méthode 3 : *Target Encoding* (appliqué uniquement aux données d'entraînement)

On encode chaque catégorie par une *statistique* de la variable *cible* (*y*) et on réduit alors la *dimensionnalité* par rapport au *One-Hot Encoding*.

```
# Target Encoding
import pandas as pd
df = pd.DataFrame({'ville': ['Fès', 'Rabat', 'Fès', 'Casablanca', 'Rabat', 'Fès'],
                   'achat': [1, 0, 1, 0, 1, 0] # variable cible}) # Données fictives
# Moyenne de la cible par ville : Moyenne=(somme des achats)/nb_occurrences
target_mean = df.groupby('ville')['achat'].mean()
print('target_mean=\n', target_mean)
# Encodage de la variable 'ville' (Remplacement des catég. par leur moyenne)
df['ville_encoded'] = df['ville'].map(target_mean)
print('df=\n', df)
```

	ville	achat	ville_encoded
0	Fès	1	0.666667
1	Rabat	0	0.500000
2	Fès	1	0.666667
3	Casablanca	0	0.000000
4	Rabat	1	0.500000
5	Fès	0	0.666667

NB : *scikit-learn* propose plusieurs méthodes d'encodage des variables catégorielles.

## III- Transformation des données ()

29

### 3- Transformation logarithmique

- Si une valeur est 100 fois plus grande (*outlier/extrême*) que les autres, elle peut fausser tout le modèle. La transformation logarithmique permet d'atténuer cet effet.
- La transformation logarithmique :
  - Réduit la *dispersion* des données.
  - Rend les *distributions* plus normales.
  - Utile pour les *revenus*, les *prix*, les *durées*.

```
import numpy as np
revenus = [3000, 3500, 4000, 1000000]
# Transformation logarithmique
revenus_log = np.log1p(revenus)
print('revenus_log= ', revenus_log)
```

1 000 000 est une valeur **extrême** (outlier) par rapport aux autres.

```
revenus_log= [ 8.00670085  8.16080392  8.29429961 13.81551156]
```

Après la transformation logarithmique, toutes les valeurs sont **proches**.

# Chap 2      Préparation des Données

## III- Transformation des données ()

30

### 4- Binarisation

- Parfois, on veut juste savoir si une valeur dépasse un *seuil*. Exemple : le revenu est-il supérieur à 50 000 ? **Oui ou non**.
- La **binarisation** permet de transformer des valeurs continues en *0* ou *1* selon un *seuil*.

```
# binarisation
from sklearn.preprocessing import Binarizer

X = [[0.2], [0.8], [1.5]]
binarizer = Binarizer(threshold = 1.0)
X_bin = binarizer.fit_transform(X)
print ('X_bin =\n', X_bin)
```

```
X_bin=
[[0.]
 [0.]
 [1.]]
```

- La binarisation est utilisée dans :
  - **Texte/NLP** : transformer les fréquences de mots en présence/absence (ex. Bag-of-Words binaire).
  - **Image Processing** : *noir/blanc* (*pixels > seuil* deviennent *1*, sinon *0*).
  - **Prétraitement ML** : quand on veut simplifier des données numériques en *signaux booléens*.

**NB:** Avec la binarisation, on perd de l'information en réduisant à *0/1*. Utile seulement quand la présence ou absence compte plus que l'intensité.

## IV- Réduction de la dimensionalité (Dimensionality Reduction)

31

- La **réduction de dimensionalité** est une technique qui consiste à prendre un grand nombre de caractéristiques (dimensions) et à en trouver un plus petit nombre qui capture l'essentiel de l'information contenue dans l'ensemble original.
- On passe d'un espace de grande dimension (Ex. 100 features) à un espace de plus faible dimension (Ex. 2 ou 3 features), tout en préservant la structure et les relations présentes dans les données.
- La "*Maldimension*" : phénomène qui avec des données ayant un très grand nombre de dimensions (features). Les problèmes principaux sont :
  - La rareté des données : Dans un espace à 100 dimensions, le volume est si immense que tous les points de données deviennent extrêmement éloignés les uns des autres. Il devient presque impossible de trouver des *patterns* ou des *clusters* denses.
  - La *distance* perd son sens : dans des espaces de très haute dimension, la distance entre tous les points commence à converger vers la même valeur, ce qui rend les algorithmes basés sur les distances (comme les k-plus proches voisins - *KNN*) inefficaces.
  - *Sur-apprentissage* (Overfitting) : beaucoup de features, souvent par rapport au nombre d'observations, mènent presque inévitablement à un modèle qui mémorise le bruit dans les données au lieu d'apprendre le pattern général.
  - Coût *computational* : Plus il y a de features, plus les calculs sont longs et complexes. Réduire la dimension accélère considérablement l'entraînement des modèles.
- La *réduction de dimensionalité* est l'antidote à la *maldimension*.

# Chap 2      Préparation des Données

## IV- Réduction de la dimensionnalité

### 1- Sélection de caractéristiques (Feature Selection)

32

- On garde les variables les plus pertinentes/utiles pour le modèle. On élimine celles qui sont redondantes ou non informatives. Ceci a comme avantages :
- Avantages :
  - **Amélioration des performances du modèle** : Moins de "bruit" (caractéristiques inutiles) permet au modèle de se concentrer sur les signaux importants, ce qui conduit souvent à de meilleures prédictions.
  - **Réduction du surajustement (Overfitting)** : Un modèle avec trop de caractéristiques, surtout si certaines sont non pertinentes, peut "mémoriser" le bruit des données d'entraînement au lieu d'en apprendre les motifs (patterns) généraux. Il performera mal sur de nouvelles données.
  - **Accélération de l'entraînement** : Moins de données à traiter signifie des algorithmes plus rapides. Cela est essentiel pour les très grands jeux de données (Big Data).
  - **Amélioration de l'interprétabilité** : Un modèle avec moins de caractéristiques est plus simple à comprendre et à expliquer (un aspect crucial pour le *Explainable AI - XAI*). On peut facilement identifier quels facteurs influencent le plus la prédiction.

# Chap 2      Préparation des Données

## IV- Réduction de la dimensionnalité

### 1- Sélection de caractéristiques

33

#### Techniques de sélection (à utiliser seulement sur les données d'Entraînement)

3 grandes familles de méthodes (qui peuvent être combinées). Le choix de la méthode dépend d'un compromis entre la *vitesse* de calcul, la *performance* recherchée et la *taille* du Dataset.

##### i. Méthodes Filtres (Filter Methods)

- Ces méthodes évaluent la pertinence des caractéristiques indépendamment du modèle de ML à utiliser par la suite. Elles se basent sur des scores *statistiques* (comme la *corrélation*) pour classer les caractéristiques.
- Principe :
  - Calculer un *score* pour chaque caractéristique (exple : *corrélation* avec la variable *cible*).
  - Sélectionner les *N* caractéristiques avec les meilleurs scores.
  - Ensuite, entraîner le modèle avec cette sélection.
- Avantages : Très rapides et peu coûteuses en calcul.
- Inconvénients : Ignorent l'interaction avec le modèle et les interactions entre les caractéristiques.
- Exemples de méthodes filtres :
  - **Corrélation** : Garder les caractéristiques fortement corrélées avec la variable cible.
  - **Test du Chi<sup>2</sup>** : Pour les données catégorielles.
  - **ANOVA** (Analysis of Variance) : Pour évaluer l'influence d'une caractéristique numérique sur une cible catégorielle.
  - **Information Mutuelle** : Mesure la dépendance entre deux variables.

# Chap 2      Préparation des Données

## IV- Réduction de la dimensionnalité

### 1- Sélection de caractéristiques

34

#### Techniques de sélection

##### ii. Méthodes Embarquées (Embedded Methods)

- La sélection est intégrée directement au processus d'entraînement du modèle. Le modèle apprend quelles caractéristiques sont les meilleures pour améliorer sa propre performance.
- Principe :  
Certains algorithmes de *ML* ont naturellement des mécanismes intégrés pour effectuer la sélection.
- Avantages :  
Plus précises que les méthodes *filtres*, car elles sont spécifiques au modèle. Moins coûteuses que les méthodes *wrappers*.
- Inconvénients : Liées au choix de l'algorithme.
- Exemples de techniques embarquées :
  - **Régression Lasso** (*L1 regularization*) : Pénalise les coefficients des caractéristiques. Certains coefficients deviennent exactement zéro, ce qui équivaut à les supprimer.
  - **Arbres de décision et Forêts Aléatoires** : Fournissent naturellement un score d'importance pour chaque caractéristique (basé sur leur capacité à réduire l'impureté des nœuds). On peut ensuite sélectionner les plus importantes:

# Chap 2      Préparation des Données

## IV- Réduction de la dimensionnalité

### 1- Sélection de caractéristiques

#### Techniques de sélection

35

##### iii. Méthodes Wrappers (Wrapper Methods)

- Elles utilisent les performances d'un modèle spécifique pour évaluer la qualité d'un sous-ensemble de caractéristiques. Elles "emballent" (wrap) le modèle et testent différentes combinaisons.
- Principe :
  - a. Choisir un sous-ensemble de caractéristiques.
  - b. Entraîner le modèle avec ce sous-ensemble.
  - c. Évaluer sa performance (ex: précision, F1-score).
  - d. Répéter les étapes a–b–c avec un nouveau sous-ensemble.
- Avantages : Très performantes, car elles tiennent compte des interactions entre les caractéristiques et sont optimisées pour le modèle choisi.
- Inconvénients : Extrêmement coûteuses en calcul (Complexité exponentielle), car elles nécessitent d'entraîner le modèle de nombreuses fois. Risque de surajustement.
- Exemples :
  - **Recherche arrière (Backward Elimination)** : On part avec toutes les caractéristiques. On retire une à une celle qui améliore le moins les performances jusqu'à ce qu'aucune amélioration ne soit possible.
  - **Recherche avant (Forward Selection)** : On part d'aucune caractéristique. On ajoute une à une celle qui améliore le plus les performances.
  - **Recherche exhaustive** : Teste toutes les combinaisons possibles (très rarement utilisable en pratique à cause de son coût).

# Chap 2 Préparation des Données

## IV- Réduction de la dimensionnalité

### 1- Sélection de caractéristiques

36

#### Exemple : Méthodes Filtres

```
# 1- Chargement et inspection du Dataset
```

```
import pandas as pd
```

```
import seaborn as sns
```

```
import matplotlib.pyplot as plt
```

```
# Télécharger le dataset "Car details v3.csv" dans :
```

```
https://www.kaggle.com/datasets/nehalbirla/vehicle-dataset-from-cardekho
```

```
# Renommer "Car details v3.csv" en "vehicles.csv".
```

```
df = pd.read_csv('D:/Docs/Abdel 2020-2021/Mes Cours/Maching Learning/vehicles.csv')
```

```
# Attention "/" et non pas "\\" dans le chemin de Windows
```

```
df = df.rename(columns={'selling_price': 'price'}) # Renommer la colonne cible
```

```
print('info :')
```

```
print(df.info())
```

```
print('Premières lignes : ')
```

```
Print(df.head())
```

```
info :  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 8128 entries, 0 to 8127  
Data columns (total 13 columns):  
 #   Column      Non-Null Count  Dtype     
---    
 0   name        8128 non-null   object    
 1   year         8128 non-null   int64     
 2   price        8128 non-null   int64    
 3   km_driven    8128 non-null   int64    
 4   fuel          8128 non-null   object    
 5   seller_type  8128 non-null   object    
 6   transmission 8128 non-null   object    
 7   owner         8128 non-null   object    
 8   mileage       7907 non-null   object    
 9   engine        7907 non-null   object    
 10  max_power    7913 non-null   object    
 11  torque        7906 non-null   object    
 12  seats         7907 non-null   float64  
dtypes: float64(1), int64(3), object(9)  
memory usage: 825.6+ KB  
None
```

```
Premières lignes:  
          name  year  price  km_driven  fuel  seller_type  \   
0   Maruti Swift Dzire VDI  2014  450000  145500 Diesel Individual  
1   Skoda Rapid 1.5 TDI Ambition 2014  370000  120000 Diesel Individual  
2   Honda City 2017-2020 EXi  2006  158000  140000 Petrol Individual  
3   Hyundai i20 Sportz Diesel  2010  225000  127000 Diesel Individual  
4   Maruti Swift VXI BSIII 2007  130000  120000 Petrol Individual  
  
          transmission  owner  mileage  engine  max_power  \   
0   Manual First Owner  23.4 kmpl  1248 CC  74 bhp  
1   Manual Second Owner 21.14 kmpl  1498 CC 103.52 bhp  
2   Manual Third Owner 17.7 kmpl  1497 CC  78 bhp  
3   Manual First Owner  23.0 kmpl  1396 CC  90 bhp  
4   Manual First Owner  16.1 kmpl  1298 CC  88.2 bhp  
  
          torque  seats  
0   190Nm@ 2000rpm  5.0  
1   250Nm@ 1500-2500rpm  5.0  
2   12.7@ 2,700(kgm@ rpm)  5.0  
3   22.4 kgm at 1750-2750rpm  5.0  
4   11.5@ 4,500(kgm@ rpm)  5.0
```

# Chap 2 Préparation des Données

## IV- Réduction de la dimensionnalité

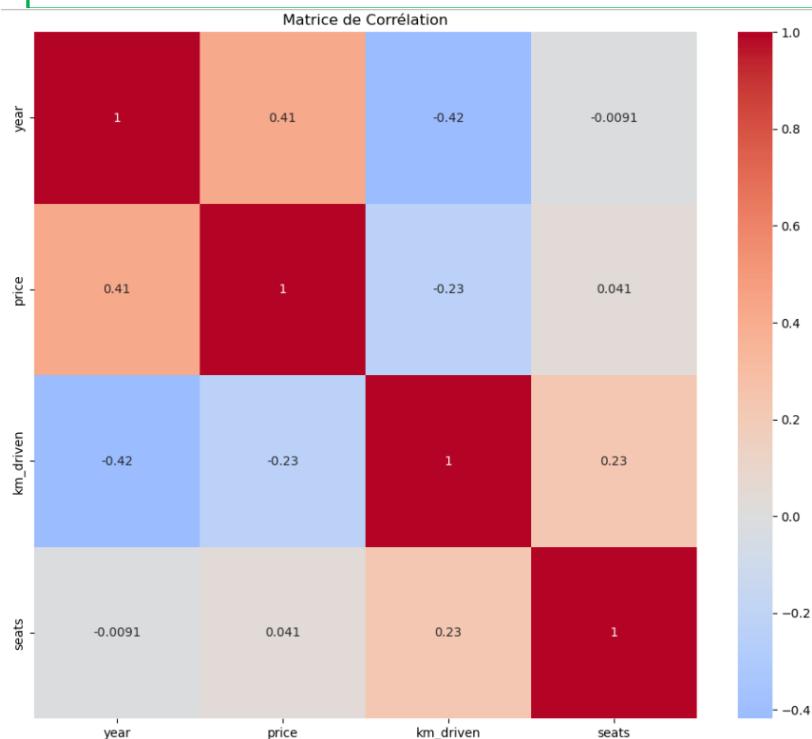
### 1- Sélection de caractéristiques

37

#### Exemple : Méthodes Filtres

##### # 2- Matrice de corrélation

```
correlation_matrix = df.select_dtypes(include=['number']).corr() # on retiendra juste les colonnes numériques (4 colonnes)
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Matrice de Corrélation')
plt.show()
```



Couleur	Interprétation
Rouge foncé	Corrélation fortement <b>positive</b> (+1)
Blanc	<b>Pas de corrélation</b> (0)
Bleu foncé	Corrélation fortement <b>négative</b> (-1)

Observation	Interprétation
Corrélation forte <b>positive</b>	Variables évoluent ensemble (peut-être redondantes)
Corrélation forte <b>négative</b>	Inversement proportionnelles (intéressant pour prédiction)
Corrélation proche de <b>0</b>	Pas de lien linéaire (peu utile en régression linéaire)
Corrélation très forte entre deux features	Risque de <b>multicolinéarité</b> → enlever l'une des deux

#### Limites de la heatmap :

- Elle ne détecte que les relations linéaires.
- Elle ne donne pas d'informations sur la causalité.
- Elle peut être trompeuse si les variables ne sont pas standardisées.

# Chap 2      Préparation des Données

## IV- Réduction de la dimensionnalité

### 1- Sélection de caractéristiques

38

#### Exemple : Méthodes Filtres

```
# 3 Corrélation avec la cible
target_correlation = correlation_matrix['price'].abs().sort_values(ascending=False)
print("Corrélation avec le prix :")
print(target_correlation)
```

```
Corrélation avec le prix :
price      1.000000
year       0.414092
km_driven  0.225534
seats      0.041358
Name: price, dtype: float64
```

```
# 4- sélection des 4 features les plus corrélées avec la cible
top_4_corr = target_correlation[1:4].index.tolist() # On ignore la corrélation de 'price' avec elle-même
print("\nTop 4 des caractéristiques les plus corrélées (linéairement) :", top_4_corr)
```

```
Top 4 des caractéristiques les plus corrélées (linéairement) : ['year', 'km_driven', 'seats']
```

**NB:** On aurait pu prendre seulement "year" et "km\_driven" car "seats" a une corrélation faible (0.0413) avec la cible "price".

**IV- Réduction de la dimensionnalité****2- Extraction de Caractéristiques (Feature Extraction)**

39

- Contrairement à la sélection de features qui choisit parmi les variables existantes, l'extraction crée de nouvelles variables à partir des données originales. Elle permet :
  - L'amélioration des performances en créant des représentations plus informatives.
  - La réduction de la dimensionnalité pour lutter contre le fléau de la dimension.
  - La capture de relations non-linéaires ou complexes que les features originales ne révèlent pas directement.

Exemples : extraction des features mesurables :

- Texte : peut être remplacé par "fréquence des mots".
  - Image : remplacée par des contours et couleurs dominantes.
  - Audio : remplacé par des spectres de fréquence.
- Méthodes d'extraction :

Type de données	Méthodes
Texte	TF-IDF, Word2Vec, BERT
Image	Histogrammes, CNN, SIFT
Audio	MFCC, spectrogrammes
Tabulaires	Encodage, normalisation, interactions

## IV- Réduction de la dimensionnalité

### 2- Extraction de Caractéristiques

40

- On peut créer des variables à la main (feature engineering) ou les extraire automatiquement (feature extraction).
- Le **feature engineering** est basé sur l'expertise métier (Ex: ratios, interaction).
- La **feature extraction** correspond à des transformations automatiques et est basée sur des algorithmes (Ex: PCA, autoencoders):
  - PCA : réduit la dimension tout en conservant l'information.
  - Autoencoders : compressent les données via des réseaux de neurones.
  - CNN : extraient des patterns visuels dans les images.
  - Transformers : capturent le contexte dans les textes.

Exemple : On crée/extraie une nouvelle feature Taille\_Famille (**Family\_Size**) dans le dataset "**Titanic**", à partir des 2 features existantes suivantes :

- sibsp : Nombre de frères/sœurs ou époux(se) à bord (fratrie ou couple) ;
- parch: Nombre de parents (père/mère) ou enfants à bord (génération ascendante ou descendante).

**Family\_Size = sibsp + parch** : `df['family_size'] = df['sibsp'] + df['parch']`

## IV- Réduction de la dimensionnalité

### 2- Extraction de Caractéristiques

41

```
import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier # Ce modèle sera traité ultérieurement
from sklearn.metrics import accuracy_score

# Chargement du dataset Titanic
df = sns.load_dataset('titanic')
# Nettoyage
df = df[['survived', 'sex', 'age', 'class', 'embarked', 'sibsp', 'parch']].dropna()
df1 = df.copy()
# Encodage manuel
df1['sex'] = df1['sex'].map({'male': 0, 'female': 1})
df1['class'] = df1['class'].map({'Third': 3, 'Second': 2, 'First': 1})
df1['embarked'] = df1['embarked'].map({'S': 0, 'C': 1, 'Q': 2})
# Séparation
X1 = df1.drop('survived', axis=1) # axis=1 : pour supprimer une colonne et axis=0 pour supprimer une ligne
y1 = df1['survived']
# Modèle
X_train, X_test, y_train, y_test = train_test_split(X1, y1, test_size=0.2, random_state=42)
modell1 = RandomForestClassifier()
modell1.fit(X_train, y_train)
acc1 = accuracy_score(y_test, modell1.predict(X_test)) # métrique de mesure de performances, étudiée ultérieurement
print("Accuracy avec encodage simple :", acc1)
```

## IV- Réduction de la dimensionnalité

### 2- Extraction de Caractéristiques

42

```
df2 = df.copy()
# One-Hot Encoding."drop_first=True" supprime la première catégorie dans chaque variable encodée
# pour éviter la redondance et colinéarité, car une catégorie peut être déduite des autres.
df2 = pd.get_dummies(df2, columns =['sex', 'class', 'embarked'], drop_first = True)
# Séparation
X2 = df2.drop('survived', axis=1)
y2 = df2['survived']
# Modèle
X_train, X_test, y_train, y_test = train_test_split(X2, y2, test_size=0.2, random_state=42)
model2 = RandomForestClassifier()
model2.fit(X_train, y_train)
acc2 = accuracy_score(y_test, model2.predict(X_test))
print("Accuracy avec One-Hot Encoding :", acc2)
df3 = df.copy()
# Encodage manuel
df3['sex'] = df3['sex'].map({'male': 0, 'female': 1})
df3['class'] = df3['class'].map({'Third': 3, 'Second': 2, 'First': 1})
df3['embarked'] = df3['embarked'].map({'S': 0, 'C': 1, 'Q': 2})
# Feature engineering : taille de la famille
df3['family_size'] = df3['sibsp'] + df3['parch']
# Séparation
X3 = df3.drop(['survived', 'sibsp', 'parch'], axis=1)
y3 = df3['survived']
```

## IV- Réduction de la dimensionnalité

## 2- Extraction de Caractéristiques

43

## # Modèle

```
x_train, x_test, y_train, y_test = train_test_split(x3, y3, test_size=0.2, random_state=42)
model3 = RandomForestClassifier()
model3.fit(x_train, y_train)
acc3 = accuracy_score(y_test, model3.predict(x_test))
print("Accuracy avec extraction de features :", acc3)

print("\nComparaison des transformations :")
print(f"- Encodage simple      : {acc1:.3f}")
print(f"- Encodage One-Hot    : {acc2:.3f}")
print(f"- Extraction de features: {acc3:.3f}")
```

Comparaison des transformations :

- Encodage simple : 0.790
- Encodage One-Hot : 0.790
- Extraction de features: 0.776

## Remarques :

- i. l'encodage One-Hot n'a aucun apport. Explication : Le modèle utilisé (Random Forest) gère très bien les variables catégorielles encodées en entiers contrairement aux modèles linéaires (comme Logistic Regression). Les modèles linéaires interprètent les entiers comme des valeurs ordonnées, ce qui peut même fausser les résultats.
- ii. L'extraction de feature (family\_size) n'a pas amélioré l'Exactitude/accuracy (performances) et l' a même légèrement réduite → Revoir **les features extraites ou le modèle**.
- iii. Dans cet exemple, il n'y pas de **fuite de données** (Data leakage) car toutes les transformations faites sont déterministes sans apprendre de paramètre global (moyenne, corrélation, fréquence, etc.).

# Chap 2      Préparation des Données

## V- Division/Séparation des données (Data Splitting)

44

- La séparation des données est une étape fondamentale pour garantir que le modèle apprend de manière fiable et qu'il est évalué sur des données qu'il n'a jamais vues.
- Surapprentissage (Overfitting) : Le modèle apprend par cœur le jeu d'entraînement, y compris le bruit et les détails, mais échoue à généraliser sur de nouvelles données.
- Analogie :
  - Un étudiant qui mémorise les réponses d'un examen d'entraînement sans comprendre les concepts. Il aura 20/20 à l'entraînement mais échouera le jour de l'examen final.
  - L'étudiant doit apprendre les méthodes, les concepts et le raisonnement nécessaire et le jour de l'examen, avec de nouveaux exercices qu'il n'a jamais vus, il aura une note (<=20) qui mesurera vraiment sa compréhension.
- Solution : entraîner le modèle sur un jeu de données, et l'évaluer sur un autre jeu, totalement nouveau pour lui. C'est le principe de base de la séparation des données.
- Une bonne pratique en ML est de subdiviser/séparer le dataset en 3 sous-ensembles :
  - i. Ensemble d'Entraînement (*Training Set*) ;
  - ii. Ensemble de Test (*Test Set*) ;
  - iii. Ensemble de Validation (*Validation Set*) (Optionnel mais très important)

# Chap 2      Préparation des Données

## V- Division/Séparation des données

45

- **Training Set** : C'est avec ces données que le modèle apprend. Il ajuste ses paramètres internes (comme les poids dans un réseau de neurones) pour minimiser l'erreur sur ces exemples. Taille : **70-80%** du dataset original.
- **Test Set** : doit être conservé loin du modèle pendant son apprentissage. Il évalue la performance finale du modèle sur de nouvelles données inconnues. Taille : **20-30%** du dataset original.
- **Validation Set** : utilisé pour ajuster les hyperparamètres du modèle (ex. : profondeur d'un arbre, taux d'apprentissage). Taille : on prend généralement **80%** du dataset original pour le Training + Validation, puis on sépare ce bloc en **80%** Training et **20%** Validation).

Exemple 1 : ici, on considère juste 2 sous-ensembles : Training Set et Test Set.

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42, stratify=y)
```

Exemple 2 : ici, on considère les 3 sous-ensembles.

```
# On sépare d'abord le Test Set (20%)
x_temp, x_test, y_temp, y_test = train_test_split( x, y,
    test_size=0.2,      # 20% pour le test
    random_state=42,    # Pour la reproductibilité
    stratify=y          # Préserve la proportion des classes
)
```

# Chap 2      Préparation des Données

## V- Division/Séparation des données

46

```
# Ensuite, on sépare le Temp Set en Training et validation
x_train, x_val, y_train, y_val = train_test_split(
    X_temp, y_temp,
    test_size=0.25,           # 0.25 * 0.8 = 0.2 (20% du total original)
    random_state=42,          # Même seed pour la reproductibilité
    stratify=y_temp           # Préserve la proportion ici aussi
)
```

**Remarque 1 :** l'instruction "stratify=y" permet d'éviter le **Déséquilibre Accidentel**

Exemple : un dataset sur 100 patients pour détecter une maladie rare :

- 90 patients sont sains (*Classe 0*)
- 10 patients sont malades (*Classe 1*)

Si on sépare les données au hasard sans *stratify*, on pourrait obtenir par malchance :

- Training set : 80 patients, tous sains (0 malade)
- Test set : 20 patients, dont 10 malades.

**Problème** : Le modèle n'aura jamais vu de cas malade pendant l'entraînement ! Il ne saura pas les reconnaître.

**Solution** : *stratify=y*. Cela signifie : "Préserver la proportion originale des classes dans chaque sous-ensemble que l'on crée."

# Chap 2      Préparation des Données

## V- Division/Séparation des données

47

**Remarque 2 :** l'instruction `random_state=42` permet d'éviter L'Aléatoire Incontrôlé.

Quand on sépare des données ou entraîne un modèle, il y a souvent un élément de hasard :

- Mélange des données avant la séparation
- Initialisation des paramètres du modèle
- Construction des arbres de décision, etc.

**Problème** : sans `random_state`, à chaque exécution du code, on obtiendra un résultat différent !

**Solution** : `random_state`. C'est comme une graine (`seed`) qui initialise le générateur de nombres aléatoires.

**42** est une référence culturelle dans le monde de l'informatique. On aurait pu mettre n'importe entier.

# Chap 2      Préparation des Données

## Etude de cas pratique

48

- On voudrait développer un modèle de Machine Learning capable de prédire si un passager du Titanic a survécu ou non, en fonction de ses caractéristiques personnelles et de voyage.
- On utilisera le jeu de données *Titanic* disponible sur *Kaggle* ou via la bibliothèque *seaborn*. Ce dataset contient des informations telles que :
  - *pclass* : classe du billet (1ère, 2e, 3e)
  - *sex* : sexe du passager
  - *age* : âge
  - *fare* : tarif payé
  - *embarked* : port d'embarquement
  - *survived* : variable cible (0 = non survécu, 1 = survécu).

Travail demandé : mettre en œuvre les 7 étapes du cycle de Machine Learning :

- **Collecte de données** : Charger le dataset *Titanic* et explorer son contenu.
- **Préparation des données** : Nettoyer les données, gérer les valeurs manquantes, encoder les variables catégorielles et normaliser si nécessaire.
- **Choix du modèle** : Sélectionner un algorithme de classification adapté (ex. *Random Forest*, *Logistic Regression*, *SVM*, etc.).

## Etude de cas pratique

49

- **Entrainement du modèle** : Diviser les données en ensembles d'entraînement et de test, puis entraîner le modèle.
- **Evaluation du modèle** : Mesurer la performance du modèle à l'aide de métriques comme la *précision*, le *rappel*, le *F1-score* et la *matrice de confusion*.
- **Réglage des hyperparamètres** : Optimiser les paramètres du modèle à l'aide de techniques comme *GridSearchCV* ou *RandomizedSearchCV*.
- **Prédiction** : Utiliser le modèle final pour prédire la survie d'un ou plusieurs passagers fictifs.

# Chap 3 Régression linéaire et polynomiale

## I- Introduction

50

- L'**apprentissage supervisé** est une méthode où le modèle apprend à partir de données étiquetées.
- Chaque jeu de données (*Dataset*) contient :
  - Des entrées (*features*)
  - Une sortie connue (*label* ou *cible* ou *target*)
- Le but est de prédire la sortie pour de nouvelles données.  
Exemple : prédire si un email est un spam (entrée = texte et sortie = spam ou non).
- **Principe :**
  - Le modèle reçoit un ensemble de données d'entraînement.
  - Il apprend à associer les entrées aux sorties.
  - Il ajuste ses paramètres pour minimiser l'erreur de prédiction.
  - Il est ensuite testé sur de nouvelles données (jamais vues).
- 2 grandes familles de problèmes supervisés :
  - **Régression** : Prédire une valeur continue (ex. *prix* d'une maison, *consommation* d'électricité).
  - **Classification** : Classer selon une *Catégorie/Classe* (ex. diagnostiquer une maladie malade/sain).
- **Modèles les plus utilisés** : *Régression linéaire* ; *Régression logistique* ; *Arbres de décision*; *Random Forest* ; *SVM*(Support Vector Machines) ; *KNN* (K-Nearest Neighbors).

# Chap 3      Régression linéaire et polynomiale

## II- Régression linéaire

51

### 1- Régression linéaire simple

- On traite d'abord la *régression linéaire simple* à une variable indépendante avant d'aborder la *régression multiple* à plusieurs variables indépendantes.
- Exemple de problème : On voudrait prédire le prix de vente moyen de maisons en fonction du nb moyen de leur pièces
  - Jeu de données (Dataset) célèbre : **Boston Housing** (Boston, USA, 1970). Le dataset original a été retiré de la bibliothèque scikit-learn (à partir de la version 1.2) pour des raisons éthiques et raciales.
  - Variable explicative/Entrée (X) : *Nombre moyen de pièces* (RM)
  - Variable à prédire (Y) : *Prix médian* (MEDV) en milliers de \$
  - **Quelle relation existe entre ces deux variables ?**

```
import pandas as pd          # OpenML: plateforme de ML gratuite pr partager Datasets, modèles, etc.
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import fetch_openml

# Chargement des données Boston_Housing. # Ce dataset peut être aussi chargé à parti de Kaggle
# sous forme ".csv" : df = pd.read_csv('boston_housing.csv')
boston = fetch_openml(name='boston', version=1, as_frame=True)
df = boston.frame
print ('Voici le dataset Boston_Housing avec toutes les variables:')
df.info()
```

## II- Régression linéaire

```
Voici le dataset toutes les variables:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
 #   Column   Non-Null Count  Dtype  
--- 
 0   CRIM      506 non-null   float64
 1   ZN        506 non-null   float64
 2   INDUS     506 non-null   float64
 3   CHAS      506 non-null   category
 4   NOX       506 non-null   float64
 5   RM        506 non-null   float64
 6   AGE        506 non-null   float64
 7   DIS        506 non-null   float64
 8   RAD        506 non-null   category
 9   TAX        506 non-null   float64
 10  PTRATIO    506 non-null   float64
 11  B          506 non-null   float64
 12  LSTAT      506 non-null   float64
 13  MEDV       506 non-null   float64
dtypes: category(2), float64(12)
memory usage: 49.0 KB
```

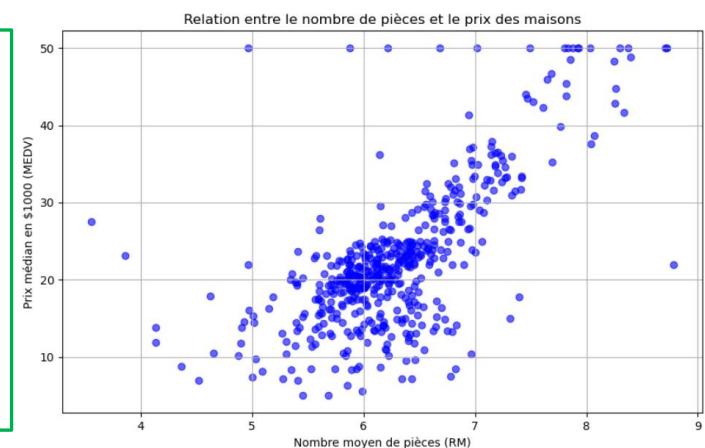
```
# Aperçu des données
# Récupération juste des 2 colonnes : RM et MEDV
print(df[['RM', 'MEDV']].head())

# Statistiques descriptives
print(df[['RM', 'MEDV']].describe())
```

	RM	MEDV
0	6.575	24.0
1	6.421	21.6
2	7.185	34.7
3	6.998	33.4
4	7.147	36.2
count	506.000000	506.000000
mean	6.284634	22.532806
std	0.702617	9.197104
min	3.561000	5.000000
25%	5.885500	17.025000
50%	6.208500	21.200000
75%	6.623500	25.000000
max	8.780000	50.000000

**NB :** Chaque ligne du dataset représente une zone résidentielle (pas un logement individuel), et chaque colonne est une caractéristique de cette zone.

```
# Le nuage de points (relation entre RM et MEDV)
plt.figure(figsize=(10, 6))
plt.scatter(df['RM'], df['MEDV'], alpha=0.6, color='blue')
plt.xlabel('Nombre moyen de pièces (RM)')
plt.ylabel('Prix médian en $1000 (MEDV)')
plt.title('Relation entre le nombre de pièces et le prix des maisons')
plt.grid(True)
plt.show()
```



# Chap 3      Régression linéaire et polynomiale

## II- Régression linéaire

53

- D'après le nuage de points (droite épaisse), on constate que le prix (MEDSV) augmente avec le nombre moyen de pièces (RM) presque linéairement.
- On voudrait approcher cette évolution par une droite qui passe "au plus près" de tous les points :

$$y = ax + b, \text{ Où :}$$

- $y$  = variable à prédire (prix)
- $x$  = variable explicative (nombre de pièces)
- $a$  = pente (de combien le prix augmente par pièce supplémentaire)
- $b$  = ordonnée à l'origine (prix de base quand  $x=0$ ).
- Le terme technique pour la droite de régression est "*hyperplan*" .
- La régression linéaire permet de diviser/séparer les données (en 2 parties) de manière à minimiser la distance entre l'*hyperplan* et les valeurs observées.
- La distance entre la droite d'ajustement optimal et les valeurs observées est appelée *résidu* ou *erreur*.
- Plus ces valeurs sont proches de l'*hyperplan*, plus les prédictions du modèle sont précises.
- On cherchera donc,  $a$  et  $b$  qui minimisent l'*erreur quadratique* (méthode des *Moindres Carrés*) suivante :

$$E(a, b) = \sum_{i=1}^n (y_i - (ax_i + b))^2$$

# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

54

- C'est une fonction à 2 variables. On dérive  $E(a,b)$  par rapport à  $a$  et  $b$ , et on annule les dérivées partielles pour trouver le minimum :

$$\frac{\partial E}{\partial a} = -2 \sum_{i=1}^n x_i (y_i - ax_i - b)$$

$$\frac{\partial E}{\partial b} = -2 \sum_{i=1}^n (y_i - ax_i - b)$$

→ 
$$\begin{cases} \sum y_i = a \sum x_i + bn \\ \sum x_i y_i = a \sum x_i^2 + b \sum x_i \end{cases}$$
 Système à 2 inconnus  $a$  et  $b$

Equations Normales → 
$$\begin{cases} a = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} \\ b = \bar{y} - a\bar{x} \end{cases}$$

avec  $\bar{x} = \frac{1}{n} \sum x_i$ ,  $\bar{y} = \frac{1}{n} \sum y_i$

La droite de régression (hyperplan) est alors déterminée :  $\mathbf{y} = \mathbf{ax} + \mathbf{b}$ .

# Chap 3 Régression linéaire et polynomiale

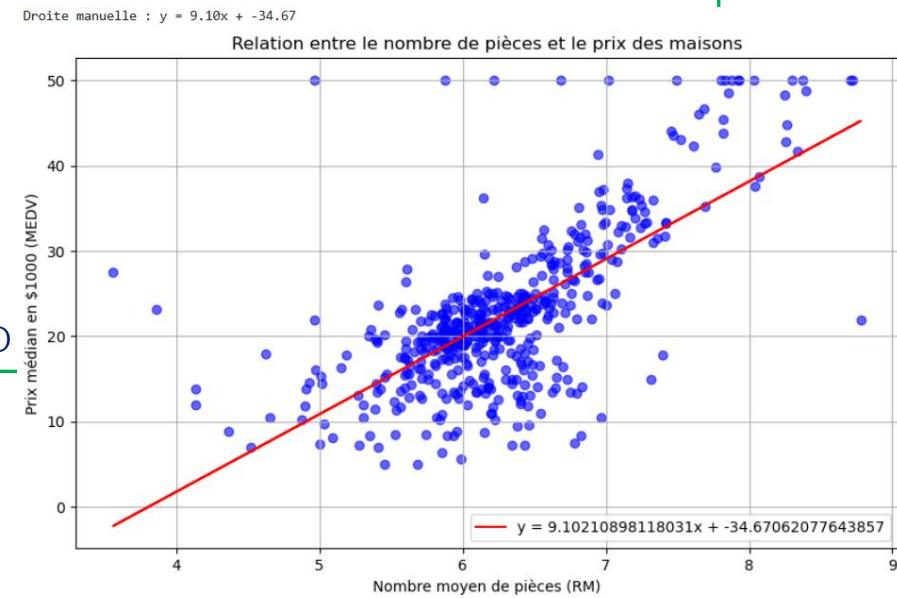
## II- Régression linéaire

55

Calcul manuel de la droite de régression (hyperplan) :  $y = ax + b$ .

```
# Détermination de la droite de la régression linéaire : Méthode des moindres carrés : la
# droite qui minimise la somme des distances au carré entre les points réels et la droite.
# Calcul manuel
x = df['RM']
y = df['MEDV']
# Moyennes
mean_x = np.mean(x)
mean_y = np.mean(y)
# Calcul de la pente (a)
numerator = np.sum((x - mean_x) * (y - mean_y))
denominator = np.sum((x - mean_x) ** 2)
a = numerator / denominator
# Calcul de l'ordonnée à l'origine (b)
b = mean_y - a * mean_x
print(f"Droite manuelle : y = {a:.2f}x + {b:.2f}")
```

Droite manuelle :  $y = 9.10x + -34.67$



# Chap 3      Régression linéaire et polynomiale

## II- Régression linéaire

56

Calcul Professionnel de la droite de régression (hyperplan) :  $y = ax + b$ .

```
# Calcul professionnel : implémentation à l'aide de scikit-learn
# Scikit-learn reproduit exactement le calcul manuel (avec numpy ou pandas). Ici, "Apprendre", c'est
# juste appliquer la méthode avec rigueur et précision.

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Préparation des données
X = df[['RM']]
# Doit être un DataFrame 2D

y = df['MEDV'] # Séparation entraînement/test
X_train, X_test, y_train, y_test = train_test_split(X,
y, test_size=0.2, random_state=42)

# Création et entraînement du modèle
model = LinearRegression()
model.fit(X_train, y_train) # Le modèle apprend

# Coefficients du modèle
a = model.coef_[0]
b = model.intercept_
print(f"Notre modèle : Prix = {a:.2f} × RM + {b:.2f}")
```

Notre modèle : Prix = 9.35 × RM + -36.25

**NB** : Les 2 droites sont proches mais pas identiques !! Raison : dans le calcul manuel, on a utilisé 100% des données, mais en Scikit-learn, on a utilisé juste les données d'entraînement (80%).

# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

57

### Evaluation du modèle : Comment savoir si notre modèle est bon ?

- Résidus :  $e_i = y_i - \hat{y}_i$ .
- SST, SSR, SSE :
  - $SST = \sum(y_i - \bar{y})^2$  (total sum of squares : variabilité totale)
  - $SSE = \sum(y_i - \hat{y}_i)^2$  (error sum of squares : résiduelle/non expliqué)
  - $SSR = \sum(\hat{y}_i - \bar{y})^2$  (regression sum of squares : expliqué)
- $R^2 = 1 - \frac{SSE}{SST}$  (proportion variance expliquée). (Coeff. de Détermination)
- RMSE :  $\sqrt{\frac{SSE}{n}}$ , MAE : moyenne des |résidus|.
- $\bar{R}^2 = 1 - (1 - R^2) \frac{n-1}{n-p}$  (corrigé pour le nombre de paramètres).

### Remarques :

- Plus **SSR** est grand, plus le modèle capture la structure réelle dans les données. Si les prédictions sont trop proches de la moyenne ( $\hat{y}_i \approx \bar{y}$ ) alors le modèle n'explique rien.
- Plus **SSE** est petit, meilleur est l'ajustement.
- **$R^2$**  donne une idée globale mais il reste *naïf* et peut être trompeur (augmente quand on ajoute variables). Utiliser **RMSE/MAE** pour l'erreur moyenne en unités observables et examiner toujours les résidus graphiquement.
- Pour comparaison de modèles, préférer **cross-validation** (CV) ou  $\bar{R}^2$  plutôt que le  **$R^2$**  naïf.

$$SST = SSR + SSE$$

$$MSE = \frac{SSE}{n - p}$$

$$MAE = \frac{1}{n} \sum_{i=1}^n |e_i|$$

$$R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}$$

# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

58

Métriques d'évaluation du modèle :

Coefficient de détermination  $R^2$  :

- Mesure la qualité de l'ajustement (de 0 à 1)
- 1 = parfait, 0 = mauvais

$R^2 : 0.7 - 0.9$  : Bon modèle

$R^2 : 0.3 - 0.6$  : Modèle moyen, améliorable

```
# Evaluation du modèle : Coefficient de détermination R2 (Mesure la qualité de l'ajustement      #
(de 0 à 1) : 1 = parfait, 0 = mauvais):
from sklearn.metrics import r2_score
# Prédictions sur l'ensemble de test
y_pred = model.predict(X_test)
# Calcul du R2
r2 = r2_score(y_test, y_pred)
print(f"Score R2 : {r2:.3f}")
# Variante : directement avec le modèle
print(f"R2 (méthode modèle) : {model.score(X_test, y_test):.3f}")
```

Score  $R^2$  : 0.371

$R^2$  (méthode modèle) : 0.371

# Chap 3      Régression linéaire et polynomiale

## II- Régression linéaire

59

### 2- Régression linéaire multiple

- La régression linéaire multiple est une extension de la régression linéaire simple, mais avec plusieurs variables indépendantes :

$$\mathbf{y} = \mathbf{a}_0 + \mathbf{a}_1 \mathbf{x}_1 + \mathbf{a}_2 \mathbf{x}_2 + \cdots + \mathbf{a}_p \mathbf{x}_p + \boldsymbol{\epsilon} \Rightarrow \text{Notation matricielle : } \mathbf{Y} = \mathbf{X}\mathbf{A} + \boldsymbol{\epsilon}$$

Avec :

- $\mathbf{Y} \in \mathbb{R}^{n \times 1}$  : vecteur des valeurs observées.
  - $\mathbf{X} \in \mathbb{R}^{n \times (p+1)}$  : matrice des variables explicatives (avec une colonne de 1 pour  $a_0$ ).
  - $\mathbf{A} \in \mathbb{R}^{(p+1) \times 1}$  : vecteur des coefficients à estimer.
  - $\boldsymbol{\epsilon}$  : vecteur des erreurs.
  - $n$  : nombre total de points de données (lignes de la matrice  $\mathbf{X}$ )
  - $p$  : nombre de colonnes de variables explicatives, sans compter l'ordonnée à l'origine  $a_0$ .
- Pour trouver la meilleure matrice  $\mathbf{A}$ , on utilisera la méthode des moindres carrés :

$$\epsilon(\mathbf{A}) = \|\mathbf{Y} - \mathbf{X}\mathbf{A}\|^2 = (\mathbf{Y} - \mathbf{X}\mathbf{A})^\top(\mathbf{Y} - \mathbf{X}\mathbf{A})$$

On dérive la fonction erreur  $\epsilon$  par rapport à  $\mathbf{A}$  et on annule le gradient :

$$\nabla_{\mathbf{A}} \epsilon(\mathbf{A}) = -2 \mathbf{X}^\top(\mathbf{Y} - \mathbf{X}\mathbf{A}) = 0 \Leftrightarrow \mathbf{X}^\top\mathbf{Y} = \mathbf{X}^\top\mathbf{X}\mathbf{A} \text{ (Equations Normales).}$$

Si  $\mathbf{X}^\top\mathbf{X}$  est inversible alors  $\mathbf{A} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{Y}$

# Chap 3      Régression linéaire et polynomiale

## II- Régression linéaire



### 2- Régression linéaire multiple

- Le calcul de  $(\mathbf{X}^T \mathbf{X})^{-1}$  a une complexité de  $\mathcal{O}(n^3)$  :  $10^5$  features  $\rightarrow$  matrice  $(10^5 \times 10^5)$  presque impossible à inverser. En plus, si le Dataset (x et y) contient des **Go**, il est impossible de le charger en mémoire d'un seul coup pour pouvoir faire les calculs !
- Pour remédier à ces 2 problèmes, on utilise une autre technique d'optimisation : **Descente de gradient**.

#### Optimisation de la fonction coût par la Descente de gradient :

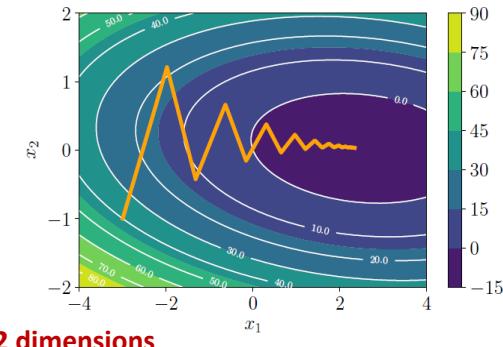
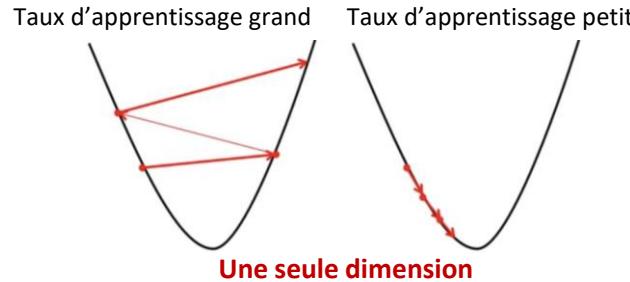
- Analogie : Un alpiniste aveugle au sommet d'une montagne, veut descendre au point le plus bas de la vallée (le minimum global). Il ne peut pas voir le chemin, mais il peut sentir avec ses pieds si le sol monte ou descend.
- La stratégie la plus simple :
  - i. Tâter le sol autour de lui pour trouver la direction de la plus forte pente descendante.
  - ii. Faire un petit pas dans cette direction.
  - iii. Répéter les étapes *i* et *ii* jusqu'à ce qu'il sente que le sol est plat de partout. Il est alors (espérons-le) au fond de la vallée.
- La **descente de gradient** est l'équivalent mathématique de cette stratégie pour une fonction de coût (cost function).
- Pour une fonction à plusieurs variables (comme une fonction coût), le *gradient* est un vecteur qui pointe dans la direction de la *plus forte augmentation* de la fonction.  
→ L'opposé du gradient  $(-\nabla J(\theta))$  pointe donc dans la direction de la *plus forte diminution* de la fonction.

# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

61

- Le **Taux d'Apprentissage** ( $\alpha$ - Alpha) : taille du pas que l'alpiniste va faire à chaque itération.
  - Un  $\alpha$  trop petit : La descente est très lente. L'entraînement prend un temps élevé.
  - Un  $\alpha$  trop grand : Les pas sont trop grands. On risque de "zapper" le minimum, voire de diverger et de voir l'erreur augmenter.



Source : Marc Peter Deisenroth, A. Aldo Faisal Cheng Soon Ong

### Optimisation de la fonction coût en ML : Descente de gradient

- Fonction coût :  $\theta \mapsto J(\theta)$  : mesure à quel point notre modèle Machine Learning se trompe. Elle quantifie l'erreur entre les prédictions du modèle et les vraies valeurs.
- $\theta$  : paramètres du modèle (Exple : le poids  $a$  et le biais  $b$  dans une régression linéaire  $y = a * x + b$ ).
- Objectif : Trouver les valeurs de  $\theta$  qui minimisent cette fonction coût  $J(\theta)$ .
- Un coût *faible* signifie un modèle *précis*.
- Optimisation par plusieurs itérations améliorant à chaque fois la valeur de  $\theta$  :  $\theta = \theta - \alpha * \nabla J(\theta)$ .

# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

62

Algorithme de la descente de gradient :

1.  $\nabla J(\theta)$  : On calcule le gradient (la direction et la pente de la montagne à notre position actuelle).
2.  $\alpha * \nabla J(\theta)$  : On multiplie le gradient par le taux d'apprentissage pour déterminer la **taille du pas**.
3.  $\theta - \dots$  : On se déplace dans la direction **opposée** au gradient (on descend la pente). On met à jour nos paramètres  $\theta$ .
4. On répète ces étapes jusqu'à ce que le gradient soit (presque) nul, ce qui signifie qu'on est (presque) au **minimum**.

Avantages de la descente de gradient :

- i. **Complexité temporelle** : par itération :  $O(n*p)$  pour *gradient batch*. Pour un *mini-batch* (*mini-lot*) de taille  $k$ , on calcule des gradients sur  $k$  exemples  $\rightarrow O(kp)$ . C'est une opération *linéaire* par rapport au nombre de features  $p$  et d'exemples  $k$ . Même avec  $p = 10^5$ , une itération est très rapide.
- ii. **Complexité spatiale (Mémoire)** : Pas besoin de tout charger. On peut utiliser la descente de gradient par **mini-lots**. On charge seulement un petit groupe d'exemples (Exple. 256) à la fois en mémoire, on calcule le gradient sur ce lot, on met à jour les paramètres, et on passe au lot suivant. On peut ainsi entraîner un modèle sur un dataset de  $1To$  avec seulement  $8 Go$  de RAM.

**Résumé** : La descente de gradient est un outil itératif, flexible et extrêmement puissant qui s'adapte à des problèmes plus vastes et complexes (*régression logistique, SVM, réseaux de neurones profonds, etc.*).

# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

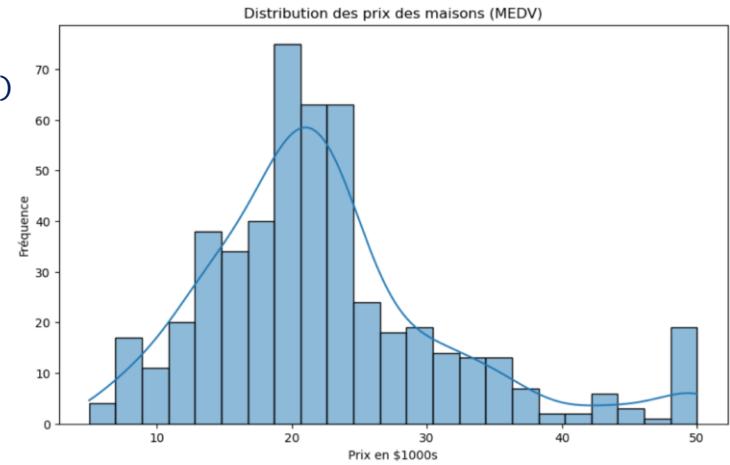
63

### Régression linéaire multiple : Exemple pratique

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler

# Chargement des données
boston = fetch_openml(name='boston', version=1, as_frame=True)
df = boston.frame

# Distribution de la variable cible
plt.figure(figsize=(10, 6))
sns.histplot(df['MEDV'], kde=True)
plt.title('Distribution des prix des maisons (MEDV)')
plt.xlabel('Prix en $1000s')
plt.ylabel('Fréquence')
plt.show()
```



# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

64

### Régression linéaire multiple : Exemple pratique

```
# Matrice de corrélation
plt.figure(figsize=(12, 8))
correlation_matrix = df.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Matrice de Corrélation')
plt.show()

# Corrélations avec la variable cible
correlation_with_target = df.corr()['MEDV'].sort_values(ascending=False)
print("\nCorrélation avec MEDV:")
print(correlation_with_target)
```



# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

65

### Régression linéaire multiple : Exemple pratique

```
# Séparation des features et de la target
x = df.drop('MEDV', axis=1) # axis=1 pour supprimer une colonne, et axis=0 pour supprimer une
                            # ligne de df (matrice)
y = df['MEDV']

# Vérification des valeurs manquantes
print("valeurs manquantes par colonne:")
print(x.isnull().sum())

# Normalisation des features (important pour la régression)
scaler = StandardScaler() # Normalement il faut scaler après la séparation de données (data leakage)
x_scaled = scaler.fit_transform(x)

# Division en train/test (80/20)
x_train, x_test, y_train, y_test = train_test_split(x_scaled, y, test_size=0.2, random_state=42)
```

Valeurs manquantes par colonne:

CRIM	0
ZN	0
INDUS	0
CHAS	0
NOX	0
RM	0
AGE	0
DIS	0
RAD	0
TAX	0
PTRATIO	0
B	0
LSTAT	0

dtype: int64

Dimensions:

X\_train: (404, 13), X\_test: (102, 13)  
y\_train: (404,), y\_test: (102,)

# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

66

### Régression linéaire multiple : Exemple pratique

```
# Création et entraînement du modèle
model = LinearRegression()
model.fit(X_train, y_train)

# Coefficients du modèle
coefficients = pd.DataFrame({'Feature': boston.feature_names,
model.coef_}).sort_values('Coefficient', ascending=False)
print("Coefficients du modèle:")
print(coefficients)
print(f"\nIntercept: {model.intercept_:.2f}") # Ordonnée à l'origine
```

	'Coefficient':	Coefficients du modèle:
	Feature	Coefficient
5	RM	3.115718
8	RAD	2.282785
11	B	1.126499
3	CHAS	0.706532
1	ZN	0.701556
2	INDUS	0.276752
6	AGE	-0.177060
0	CRIM	-0.971494
9	TAX	-1.792605
10	PTRATIO	-1.979954
4	NOX	-1.991430
7	DIS	-3.045771
12	LSTAT	-3.628149

Intercept: 22.49

### Modèle (Hyperplan) :

$$\hat{y} = 22.49 - 0.9715 \cdot CRIM + 0.7016 \cdot ZN + 0.2768 \cdot INDUS + 0.7065 \cdot CHAS - 1.9914 \cdot NOX + 3.1157 \cdot RM - 0.1771 \cdot AGE - 3.0458 \cdot DIS + 2.2828 \cdot RAD - 1.7927 \cdot TAX + 3.1157 \cdot RM - 0.1771 \cdot AGE - 3.0458 \cdot DIS + 2.2828 \cdot RAD - 1.7927 \cdot TAX - 1.9800 \cdot PTRATIO + 1.1265 \cdot B - 3.6281 \cdot LSTAT - 1.9800 \cdot PTRATIO + 1.1265 \cdot B - 3.6281 \cdot LSTAT$$

# Chap 3 Régression linéaire et polynomiale

## II- Régression linéaire

67

### Régression linéaire multiple : Exemple pratique

```
# Prédictions
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# Métriques d'évaluation
def evaluate_model(y_true, y_pred, dataset_name):
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(y_true, y_pred)
    print(f"\nPerformance sur {dataset_name}:")
    print(f"MSE: {mse:.2f}")
    print(f"RMSE: {rmse:.2f}")
    print(f"R²: {r2:.4f}")
    return rmse, r2
rmse_train, r2_train = evaluate_model(y_train, y_pred_train, "Train")
rmse_test, r2_test = evaluate_model(y_test, y_pred_test, "Test")

# Exemple de prédiction
sample_house = X_test[0:1] # Première maison du test set
predicted_price = model.predict(sample_house)[0]
actual_price = y_test.iloc[0]
print("\nExemple de prédiction:")
print(f"Prix prédit: ${predicted_price*1000:.0f}")
print(f"Prix réel: ${actual_price*1000:.0f}")
print(f"Erreur: ${abs(predicted_price - actual_price)*1000:.0f}")
```

Performance sur Train:

MSE: 21.64

RMSE: 4.65

R<sup>2</sup>: 0.7509

Performance sur Test:

MSE: 24.29

RMSE: 4.93

R<sup>2</sup>: 0.6688

Exemple de prédiction:

Prix prédit: \$28997

Prix réel: \$23600

Erreur: \$5397

# Chap 3      Régression linéaire et polynomiale

## III- Régression Polynomiale

68

- En fait, la **régression polynomiale** est une extension de la régression linéaire : créer de nouvelles *features* à partir des features existantes pour capturer des relations *non-linéaires*, tout en continuant à utiliser un modèle linéaire dans ses paramètres.

**Formule de base** (univariée : une seule feature  $x$ ) :

$$y = a_0 + a_1x + a_2x^2 \dots + a_p x^p + \epsilon, \text{ où :}$$

$y$  : La variable que l'on cherche à prédire (la variable cible).

$x$  : Notre unique variable explicative de départ (exple : nb moyen de pièces RM).

$p$  : degré du polynôme.

$x^2, x^3, \dots, x^p$ : Les nouvelles variables que l'on crée. On traite  $x^2$  comme une variable distincte de  $x$ .

$a_0$  : valeur de  $y$  quand toutes les variables ( $x, x^2$ , etc.) sont à zéro.

$a_1, a_2, \dots, a_p$  : coefficients du modèle. Ce sont les poids associés à chaque variable ( $x, x^2$ , etc.) que le modèle va apprendre lors de son entraînement.

$\epsilon$  : erreur (résidu), le pourcentage que le modèle ne peut pas expliquer/prédire

- le modèle est linéaire par rapport aux paramètres  $a \rightarrow$  on peut toujours utiliser la méthode des *Moindres Carrés* pour trouver les meilleurs coefficients  $a$ .

# Chap 3 Régression linéaire et polynomiale

## III- Régression Polynomiale

69

Formule multivarié (plusieurs features) :

$$y = a_0 + \sum_{k=1}^{\deg} \left( \sum_{\beta_1+\beta_2+\dots+\beta_n=k} a_\beta x_1^{\beta_1} x_2^{\beta_2} \dots x_n^{\beta_n} \right) + \varepsilon, \text{ où}$$

$(x_1, \dots, x_n)$  : vecteur d'entrée.

$a_0$  : terme constant (biais).

$a_\beta$  : sont les coefficients du modèle à apprendre.

$\beta_j$  : sont des entiers naturels (exposants).

Exemples 1 :  $\deg = 2, 3$  variables

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + a_4 x_1^2 + a_5 x_2^2 + a_6 x_3^2 + a_7 x_1 x_2 + a_8 x_1 x_3 + a_9 x_2 x_3 + \epsilon$$

Exemple 2 :  $\deg = 3$  et  $n = 3$  variables créent **20** caractéristiques (features) :

Terme constant (Bias) : 1 (associé à  $a_0$ )

Termes de degré 1 (3 termes) :  $x_1, x_2, x_3$

Termes de degré 2 (6 termes) :

- Les Carrés (3 termes) :  $x_1^2, x_2^2, x_3^2$

- Les Interactions 2 à 2 (3 termes) :  $x_1 x_2, x_1 x_3, x_2 x_3$

Termes de degré 3 (10 termes) :

Les Cubes (3 termes) :  $x_1^3, x_2^3, x_3^3$ , les Interactions "2-1" (6 termes) :  $x_1^2 x_2, x_1^2 x_3, x_2^2 x_1, x_2^2 x_3, x_3^2 x_1, x_3^2 x_2$ ,

les Interactions triple (1 terme) :  $x_1 x_2 x_3$

Nombre total de features :  $1 + 3 + 6 + 10 = C_{n+\deg}^{\deg} = C_6^3 = 20$

## III- Régression Polynomiale



- Avantages :
  - Capture les relations non-linéaires
  - Flexible pour différents patterns de données
  - Implémentation simple
- Inconvénients :
  - Risque d'overfitting (surtout degré élevé).
  - Sensible aux outliers.
  - Interprétation plus complexe.
- Pour choisir le degré optimal :
  - Validation croisée.
  - AIC/BIC pour comparaison de modèles.
  - Regularization (Ridge/Lasso).

# Chap 3      Régression linéaire et polynomiale

## III- Régression Polynomiale

71

Exemple pratique : Comparaison de la régression linéaire et la régression polynomiale de degré 2

```
import pandas as pd
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.feature_selection import SelectKBest, f_regression
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
import seaborn as sns
# Chargement du dataset Boston Housing
boston = fetch_openml(name="boston", version=1, as_frame=True)
df = boston.frame

# Sélection des 2 meilleures features
X = df.drop(columns="MEDV")
y = df["MEDV"]
selecteur = SelectKBest(score_func=f_regression, k=2) # sélectionner les k meilleures variables
# Les plus corrélées à la variable cible y
X_selected = selecteur.fit_transform(X, y)
selected_features = X.columns[selecteur.get_support()] # masque booléen indiquant les colonnes sélectionnées
print("Meilleures features sélectionnées :", selected_features.tolist())
```

# Chap 3 Régression linéaire et polynomiale

## III- Régression Polynomiale

72

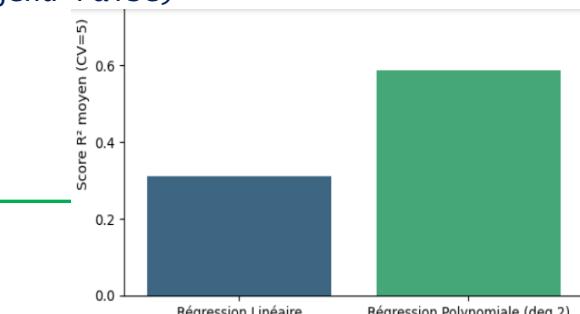
Exemple pratique : Comparaison de la régression linéaire et la régression polynomiale de degré 2

```
# Régression linéaire
lineaire_model = LinearRegression()
lineaire_scores = cross_val_score(lineaire_model, x_selected, y, cv=5, scoring='r2')
print("R² régression linéaire :", round(lineaire_scores.mean(), 4))

# Régression polynomiale (degré 2) : c'est en fait une transformation des features, suivie d'une régression linéaire
polynomial_model = make_pipeline(PolynomialFeatures(degree=2), LinearRegression())
polynomial_scores = cross_val_score(polynomial_model, x_selected, y, cv=5, scoring='r2')
print("R² régression polynomiale (degré 2) :", round(polynomial_scores.mean(), 4))

# Comparaison visuelle
models = ['Régression Linéaire', 'Régression Polynomiale (deg 2)']
scores = [lineaire_scores.mean(), polynomial_scores.mean()]
sns.barplot(x=models, y=scores, hue=models, palette="viridis", legend=False)
plt.ylabel("Score R² moyen (CV=5)")
plt.title("Comparaison des performances")
plt.ylim(0, 1)
plt.show()
```

Meilleures features sélectionnées : ['RM', 'LSTAT']  
R<sup>2</sup> régression linéaire : 0.3096  
R<sup>2</sup> régression polynomiale (degré 2) : 0.5876



Cela montre que le modèle polynomial capture mieux la complexité de la relation entre les 2 variables ['RM', 'LSTAT'] et le prix des maisons.

# Chap 4 Classification et Régression logistique

## I- Définitions

73

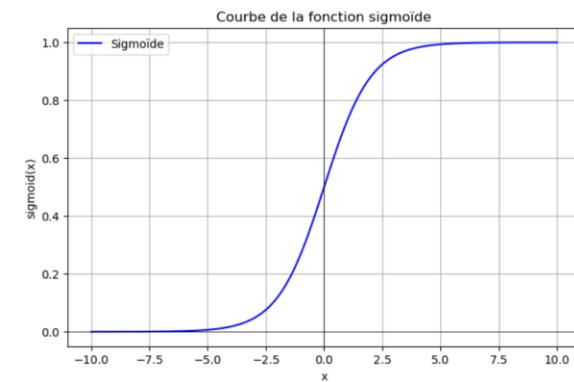
- Le *Machine Learning* supervisé consiste à entraîner un modèle à partir de données étiquetées pour qu'il puisse prédire une sortie (*label*) à partir d'une entrée (*features*).
- La **Classification** est similaire au problème de régression, sauf que les valeurs de  $y$  à prédire ne prennent qu'un petit ensemble de valeurs **discrètes**.
- La classification s'intéresse à la prédiction de catégories discrètes (classe, étiquette). La sortie est **qualitative** plutôt que **quantitative**.
- Il existe 3 types de classification :
  - **Binaire** : deux classes (exple : diagnostic de maladie oui/non, un mail est spam ou non, etc.).
  - **Multiclasse** : plus de deux classes (par exple : type de fleurs, type de maladie, reconnaissance de chiffres, prédire le type d'animaux à partir une image, etc.).
  - **Multilabel** : plusieurs classes peuvent être vraies en même temps (Exple : classification d'images avec plusieurs objets)
- En *multi-classes* standard, un exemple appartient à une et une seule classe. En *multi-étiquettes*, il peut en appartenir à plusieurs.
- Le modèle renvoie une probabilité pour chaque classe et on tranche via un **seuil**.
- La **Régression logistique** est une méthode de classification binaire. Elle permet de prédire la probabilité qu'une observation appartienne à une classe.

# Chap 4 Classification et Régression logistique

## II- Modèle-Fonction Sigmoïde

74

- La **Régrssion logistique** modélise les probabilités d'appartenance aux classes. Elle utilise une fonction logistique (*sigmoïde*) pour transformer une prédiction linéaire en **probabilité**.
- Comme en régression linéaire :
  - le modèle calcule d'abord une combinaison linéaire des variables d'entrée (c'est la partie "régression" :  $z = \beta_0 + \beta_1X_1 + \dots + \beta_kX_k$ ),
  - mais au lieu d'utiliser ce score  $z$  directement comme prédiction (ce qui donnerait un nombre quelconque), il le passe à travers la fonction logistique (la partie "logistique").
  - La fonction logistique transforme le score  $z$  en une probabilité  $P(Y = 1) = f(z)$ .
- Fonction logistique courante : **Sigmoïde** :  $\sigma(z) = \frac{1}{1+e^{-z}}$  transforme toute valeur *réelle* en une probabilité entre 0 et 1.
- **Modèle** :  $P(Y = 1 | X) = \sigma(z) = \sigma(\beta_0 + \beta_1X_1 + \dots + \beta_kX_k)$
- Autres fonctions logistiques (cela dépend du problème):  
**Probit** (*Normit*), **Cloglog**, **softmax**, **tanh**, etc.



# Chap 4 Classification et Régression logistique

## II- Modèle-Fonction softmax

75

- La Régression logistique est un classificateur probabiliste. Au lieu de dire juste "c'est un spam", elle dit "il y a 97% de chances que ce soit un spam". La décision finale (0 ou 1) est prise en appliquant un seuil (généralement 0.5) à cette probabilité.
- En régression logistique **binaire**, on utilise la **sigmoïde** pour transformer la sortie du modèle en probabilité entre **0** et **1**.
- En **régression logistique multinomiale** (plusieurs classes), on utilise la fonction **softmax** (une généralisation de *sigmoïde*) pour transformer un vecteur de scores (**logits**) en **probabilités** réparties entre plusieurs classes :

Soit un vecteur de scores  $z = [z_1, z_2, \dots, z_K]$  pour  $K$  classes.

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \in [0,1] \text{ et la somme de toutes les valeurs vaut 1.}$$

- *Softmax* choisira la classe avec la probabilité maximale comme prédiction.

# Chap 4 Classification et Régression logistique

## III- Fonction Coût

76

### Fonction coût de classification Binaire

- **MSE** n'est pas adaptée ici : 
$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - p^{(i)})^2 \quad ???$$
 avec  $p^{(i)}$  probabilité prédite et  $y^{(i)}$  la classe réelle  $\in \{0,1\}$ . MSE n'est pas cohérent avec la nature probabiliste du problème.
- L'**'Entropie croisée'** (une vraie fonction de perte probabiliste) :

$$L = - \sum_{i=1}^n [y^{(i)} \log p^{(i)} + (1 - y^{(i)}) \log(1 - p^{(i)})]$$

- Elle mesure l'écart entre la distribution réelle (les vraies classes) et la distribution prédite (les probabilités).
- Elle est issue de la théorie de l'information : plus le modèle est incertain ou se trompe, plus l'entropie croisée est élevée.
- Convexe  $\Rightarrow$  facilite l'optimisation.

### Fonction coût de la classification Multiclasse

$$L = - \sum_{i=1}^n \sum_{k=1}^K y_k^{(i)} \log(\hat{y}_k^{(i)}) \quad (\text{Convexe})$$

$K$  le nombre de classes

$y^{(i)} \in \{0, 1\}^K$  le vecteur one-hot de la vraie classe pour l'exemple  $i$

$\hat{y}^{(i)} \in [0, 1]^K$  le vecteur de probabilités prédites par le modèle

Scikit-learn : `LogisticRegression(multi_class='multinomial', solver='lbfgs')` : softmax

TensorFlow / Keras : `loss='categorical_crossentropy'` (si one-hot) ou `loss='sparse_categorical_crossentropy'` (si labels entiers).

# Chap 4 Classification et Régression logistique

## IV- Evaluation du modèle

77

- Un modèle entraîné n'est pas utile si l'on ne peut pas mesurer et interpréter ses performances de manière fiable.
- Evaluation :
  - Mesurer l'utilité : Le modèle est-il meilleur qu'une prédiction aléatoire ou qu'une règle simple ?
  - Diagnostiquer les problèmes : Le modèle fait-il certaines erreurs plus que d'autres ?
  - Comparer des modèles : Choisir entre différentes variables, algorithmes ou hyperparamètres.
  - Communiquer la confiance : Présenter les résultats de manière claire et juste aux responsables/décideurs.
- Outils et métriques d'évaluation :
  - Validation croisée (*K-fold, leave-one-out, etc.*)
  - *Precision, rappel*
  - *F1-score*
  - *Matrice de confusion*
  - Courbe *ROC, AUC*
- La métrique est choisie en fonction de l'importance du déséquilibre des classes.

**NB:** Le chapitre suivant s'intéressera à l'évaluation des modèles et aux différentes métriques.

## V- Exemples

78

Exemple 1 (Classification binaire : Tumeur bénigne ou maligne ?)

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
# Chargement et Exploration des Données
data1 = load_breast_cancer()
# Créer un DataFrame pandas pour les features
df = pd.DataFrame(data1.data, columns=data1.feature_names)
# Ajouter la colonne 'target' (la variable à prédire)
df['target'] = data1.target
# Ajouter les noms des classes pour plus de clareté
df['target_name'] = data1.target_names[data1.target]
# Afficher les premières lignes des données
print("Aperçu des données :")
print(df.head())

```

Aperçu des données :						
	mean radius	mean texture	mean perimeter	mean area	mean smoothness	\
0	17.99	10.38	122.80	1001.0	0.11840	
1	20.57	17.77	132.90	1326.0	0.08474	
2	19.69	21.25	130.00	1203.0	0.10960	
3	11.42	20.38	77.58	386.1	0.14250	
4	20.29	14.34	135.10	1297.0	0.10030	
	mean compactness	mean concavity	mean concave points	mean symmetry	\	
0	0.27760	0.3001	0.14710	0.2419		
1	0.07864	0.0869	0.07017	0.1812		
2	0.15990	0.1974	0.12790	0.2069		
3	0.28390	0.2414	0.10520	0.2597		
4	0.13280	0.1980	0.10430	0.1809		
	mean fractal dimension	...	worst perimeter	worst area	worst smoothness	\
0	0.07871	...	184.60	2019.0	0.1622	
1	0.05667	...	158.80	1956.0	0.1238	
2	0.05999	...	152.50	1709.0	0.1444	
3	0.09744	...	98.87	567.7	0.2098	
4	0.05883	...	152.20	1575.0	0.1374	
	worst compactness	worst concavity	worst concave points	worst symmetry	\	
0	0.6656	0.7119	0.2654	0.4601		
1	0.1866	0.2416	0.1860	0.2750		
2	0.4245	0.4504	0.2430	0.3613		
3	0.8663	0.6869	0.2575	0.6638		
4	0.2050	0.4000	0.1625	0.2364		
	worst fractal dimension	target	target_name			
0	0.11890	0	malignant			
1	0.08902	0	malignant			
2	0.08758	0	malignant			
3	0.17300	0	malignant			
4	0.07678	0	malignant			

[5 rows x 32 columns]

## V- Exemples

79

Exemple 1 (Classification binaire : Tumeur bénigne ou maligne ?)

```
# Informations sur le dataset
print(f"\nForme du dataset : {df.shape}")
print(f"Classes : {df['target_name'].value_counts()}")
# 0 : malignant (maligne), 1 : benign (bénigne)
# Dans la version originale, 'malignant' est 0, ce qui est la classe positive logique pour un diagnostic.

# Préparation des Données
# Séparation des features (X) et de la target (y)
X = df.drop(['target', 'target_name'], axis=1)
y = df['target']

# Division du dataset en jeu d'entraînement (80%) et jeu de test (20%)
# 'stratify=y' assure que la proportion des classes est la même dans "train" et "test"
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Normalisation des features (Crucial pour de nombreux algorithmes)
# On ajuste le scaler sur le "train" et on l'applique sur le "train" et le "test" : (Eviter data leakage)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test) # Notez que c'est ".transform" et non pas ".fit_transform"

# Entraînement du Modèle. On choisit une régression logistique, un classique pour la classification
# binaire. L'algorithme utilise une descente de gradient pour minimiser la fonction de coût.
model = LogisticRegression(random_state=42, max_iter=1000) # max_iter assure la convergence
model.fit(X_train_scaled, y_train)
```

```
Forme du dataset : (569, 32)
Classes : target_name
benign      357
malignant   212
Name: count, dtype: int64
```

## V- Exemples

80

Exemple 1 (Classification binaire : Tumeur bénigne ou maligne ?)

```

# Évaluation du Modèle
# Prédictions sur le jeu de test
y_pred = model.predict(x_test_scaled)

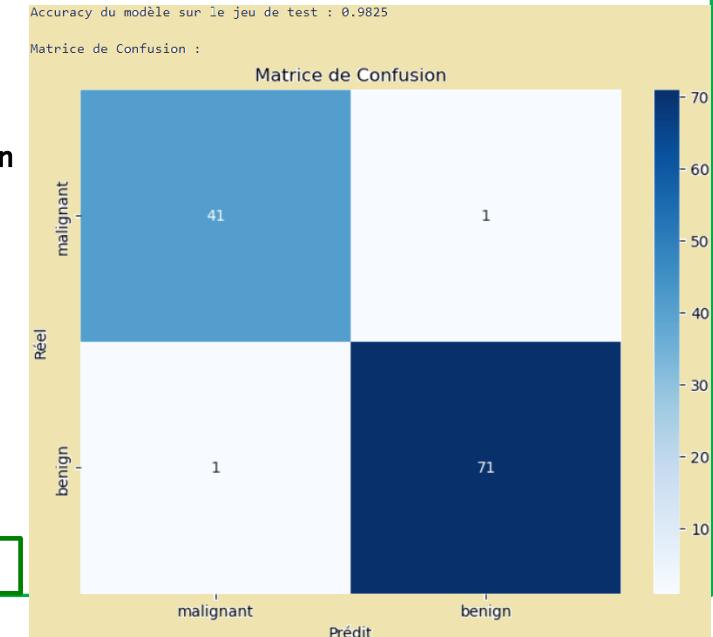
# Calcul de l'Exactitude (Accuracy)
accuracy = accuracy_score(y_test, y_pred)
print(f"\nAccuracy du modèle sur le jeu de test : {accuracy:.4f}")

# Matrice de Confusion (Beaucoup plus informative !)
print("\nMatrice de Confusion :")
cm = confusion_matrix(y_test, y_pred)

# Création d'un heatmap pour visualiser la matrice de confusion
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=data1.target_names,
            yticklabels=data1.target_names)
plt.xlabel('Prédit')
plt.ylabel('Réel')
plt.title('Matrice de Confusion')
plt.show()

```

**Accuracy=(TP+TN)/(FP+FN+TP+TN)=(41+71)/(1+14+1+71)=114/112≈0.9825**



## V- Exemples

81

Exemple 1 (Classification binaire : Tumeur bénigne ou maligne ?)

```
# Rapport de Classification (Précision, Rappel, F1-score)
print("\nRapport de Classification :")
print(classification_report(y_test, y_pred, target_names=data.target_names))

# Interprétation : Importance des Features
# Les coefficients de la régression logistique nous indiquent l'importance des features.
# On les trie par valeur absolue.

importance = pd.DataFrame({ 'feature': data.feature_names, 'coefficient': model.coef_[0] })
importance['abs_coef'] = np.abs(importance['coefficient'])
importance = importance.sort_values('abs_coef', ascending=False)

print("\nTop 5 des caractéristiques les plus importantes pour la prédiction :")
print(importance[['feature', 'coefficient']].head())
```

- **support** correspond au nombre réel d'occurrences de chaque classe dans les données de test.
- Les autres métriques (**precision**, **recall**, ...) seront traitées dans le prochain chapitre.

**Interprétation globale :** Le modèle est très performant avec une accuracy globale de 98.25%. Il commet très peu d'erreurs : seulement 2 erreurs sur 114 échantillons. Erreur la plus critique : 1 cas malin classé comme bénin (FN), car cela pourrait retarder un diagnostic important.

	Rapport de Classification :	precision	recall	f1-score	support
malignant	0.98	0.98	0.98	42	
benign	0.99	0.99	0.99	72	
accuracy				0.98	114
macro avg	0.98	0.98	0.98	114	
weighted avg	0.98	0.98	0.98	114	

	Top 5 des caractéristiques les plus importantes pour la prédiction :	feature	coefficient
21	worst texture	-1.255088	
10	radius error	-1.082965	
27	worst concave points	-0.953686	
23	worst area	-0.947756	
20	worst radius	-0.947616	

## V- Exemples

82

Exemple 2 (Multiclasse) : Modèle de classification automatique identifiant l'espèce d'une fleur d'Iris (السو森) à partir de ses mesures physiques (Dataset : Iris).

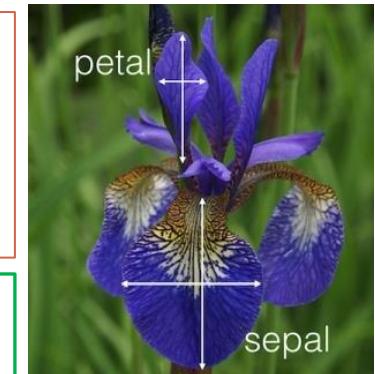
Features :

- **sepal length (cm)** : Longueur du sépale (en cm)
- **sepal width (cm)** : Largeur du sépale (en cm)
- **petal length (cm)** : Longueur du pétales (en cm)
- **petal width (cm)** : Largeur du pétales (en cm)

Target :

- species** : Espèces de la fleur d'Iris
- 0 : Iris setosa
  - 1 : Iris versicolor
  - 2 : Iris virginica

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
# Chargement des données
iris = load_iris()
X = iris.data    # Caractéristiques
y = iris.target  # Classes (0, 1, 2)
print("Exemple pédagogique, Régression Logistique de 3 classes :")
print("Dimensions des données:")
print(f"X: {X.shape}, y: {y.shape}")
print(f"Classes: {np.unique(y)}")
print(f"Noms des classes: {iris.target_names}")
```



Setosa



Versicolor



Virginica

## V- Exemples

83

Exemple 2 (Multiclasse) : Classification des fleurs d'Iris

# Division en train/test

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42, stratify=y)
model = LogisticRegression(solver='lbfgs', max_iter=1000, random_state=42)
```

# 'multi\_class' est détecté automatiquement

# le solveur utilise L-BFGS : une "descente de gradient intelligente" qui apprend de l'historique  
# pour converger plus vite)

print(f"\nParamètres du modèle:")

print(f"Solver: {model.get\_params()['solver']}")

print(f"Max iterations: {model.get\_params()['max\_iter']}")

# Entraînement du modèle

model.fit(x\_train, y\_train)

# Évaluation

y\_pred = model.predict(x\_test)

accuracy = model.score(x\_test, y\_test)

print(f"\nPerformance du modèle:")

print(f"Accuracy: {accuracy:.3f}")

# Rapport de classification détaillé

print("\n" + "="\*50)

print("Rapport de la classification détaillé :")

print("=". \* 50)

print(classification\_report(y\_test, y\_pred, target\_names=iris.target\_names))

$score_k = a_k \cdot x + b_k$  pour  $k = 0, 1, 2$ . Probabilité (softmax) :  $P(\text{classe} = k) = e^{score_k} / (e^{score_0} + e^{score_1} + e^{score_2})$   $\Rightarrow$  Prédiction : probabilité la plus élevée

Exemple pédagogique : REGRESSION LOGISTIQUE 3 CLASSES

Dimensions des données:

X: (150, 4), y: (150,)

Classes: [0 1 2]

Noms des classes: ['setosa' 'versicolor' 'virginica']

Paramètres du modèle:

Solver: lbfgs

Max iterations: 1000

Performance du modèle:

Accuracy: 0.933

Rapport de la classification détaillé :

	precision	recall	f1-score	support
setosa	1.00	1.00	1.00	15
versicolor	0.88	0.93	0.90	15
virginica	0.93	0.87	0.90	15
accuracy			0.93	45
macro avg	0.93	0.93	0.93	45
weighted avg	0.93	0.93	0.93	45

# Chap 5 Techniques d' Evaluation et Validation Croisée

## I- Introduction

84

- L'*Evaluation* permet de :
  - Tester les **Performances** : (s'assurer si le modèle atteint la précision requise).
  - Vérifier si le modèle peut **Généraliser** sur de nouvelles données non déjà vues.
  - **Comparer** les modèles et de choisir parmi plusieurs alternatives.
  - Tester la **Robustesse** d'un modèle (i.e., résistance au bruit et aux variations dans les données).
- Pour une bonne évaluation, on utilise la *validation croisée* :
  - i. Diviser les données en  **$k$  partitions** (généralement  $k=5$  ou  $10$ )
  - ii. Entraîner le modèle  **$k$**  fois, chaque fois en utilisant  **$k-1$**  partitions pour l'entraînement et **1** partition pour le test.
  - iii. Calculer la **moyenne** des performances.

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier()
scores = cross_val_score(model, X, y, cv=5) # Cross Valid. (k=5)
print(f"Score moyen : {scores.mean():.3f} (± {scores.std() * 2:.3f})")
```

Environ 95 % des scores se trouvent dans l'intervalle [mean -  $2\sigma$ , mean +  $2\sigma$ ].

**NB :** Un modèle qui prédit "toujours **non**" dans un dataset avec **95%** de "non" aura **95%** de précision, mais il est totalement inutile. Il faut donc des métriques plus significatives et bien adaptées au modèle.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## II- Validation croisée

85

### 1- Définition

- La **validation croisée** (cross-validation) est une technique d'évaluation de modèles statistiques qui consiste à partitionner un Dataset en sous-ensembles (**Plis/folds**), à entraîner le modèle sur certains de ces sous-ensembles et à le tester sur les autres.
- Utilité de la VC :
  - Evaluer la performance réelle d'un modèle.
  - Comparer différents modèles ou algorithmes.
  - Optimiser les hyperparamètres.
  - Déetecter le surapprentissage.
  - Utilisation optimale des données limitées.
  - Réduction de la variance de l'estimation.
- Principe :
  - i. Division des données en  $k$  partitions ( $5 \leq k \leq 10$  est un bon compromis **Complexité/Performance**)
  - ii. Pour chaque partition :
    - Utiliser  $k - 1$  partitions pour l'entraînement.
    - Utiliser 1 partition pour le test.
  - iii. Moyenner les performances sur tous les tests.

**NB :** Les données du dataset peuvent être mélangées (**`shuffle=True`**) avant le partitionnement.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## II- Validation croisée

86

### 2- Principales méthodes de validation croisée

#### i. Validation croisée simple (Hold-out)

- On divise les données en deux parties : train et test (par exemple 80% et 20%).
- Simple mais sensible à la manière dont les données sont séparées.

#### ii. K-Fold Cross Validation

- Les données sont divisées en  $K$  sous-ensembles égaux (et forment une partition du dataset).
- Le modèle est entraîné sur  $K - 1$  folds et testé sur le fold restant.
- Ce processus est répété  $K$  fois, chaque fold servant une fois pour le test et  $K - 1$  fois pour l'entraînement.
- Avantage : meilleure estimation de la performance globale.

#### iii. Leave-One-Out (LOOCV)

- Cas particulier de K-Fold où  $K = n$  (nombre total d'échantillons).
- Chaque observation (une ligne de dataset) est utilisée une fois comme test ( $n - 1$  lignes restantes pr train)
- Très précise, mais coûteuse en calcul.

#### iv. Stratified K-Fold

- Variante du K-Fold qui conserve la proportion des classes dans chaque fold.
- Utile pour les problèmes de classification avec classes déséquilibrées.

#### v. Time Series Split

- Pour les données temporelles, on respecte l'ordre chronologique.
- On entraîne sur les données passées et teste sur les données futures.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## II- Validation croisée

87

### Utilisation de la validation croisée dans le réglage des hyperparamètres :

La VC sert surtout à estimer la performance d'un modèle pour choisir les meilleurs hyperparamètres, sans toucher au jeu de test final.

1. On coupe le Dataset en 2 sous-ensembles :
  - Train set : utilisé pour l'entraînement + la validation croisée.
  - Test set : réservé pour l'évaluation finale.
2. Sur le **train set**, on fait une **validation croisée** :
  - i. Pour chaque combinaison d'hyperparamètres (ex. n\_neighbors, max\_depth, etc.).
  - ii. On fait une validation croisée à k plis (k-fold CV).
  - iii. On calcule la moyenne des scores de validation.
  - iv. Et on choisit les hyperparamètres avec les meilleurs résultats moyens.
3. Enfin, on **réentraîne** le modèle avec ces hyperparamètres optimaux sur *tout* le train set, puis on **évalue une seule fois** sur le test set.

**NB** : il ne faut jamais normaliser/standardiser avant de diviser les données (**Pb de fuite de données/data leakage**). On doit :

- Diviser les données en folds (validation croisée) ou en train/test.
- Calculer les statistiques de normalisation uniquement sur le jeu d'entraînement.
- Appliquer la transformation au jeu d'entraînement et au jeu de test (en utilisant les statistiques du train uniquement).

# Chap 5 Techniques d' Evaluation et Validation Croisée

## III- Grid search (Recherche de grille)

88

- La **Grid Search** (recherche sur grille) est une technique systématique pour trouver la meilleure combinaison d'Hyperparamètres d'un modèle d'apprentissage automatique.

Différence entre Hyperparamètres et Paramètres :

- **Paramètres** : Variables apprises pendant l'entraînement (ex: poids dans une régression).
- **Hyperparamètres** : Variables définies avant l'entraînement (ex: taux d'apprentissage, profondeur d'arbre, nombre de voisins dans KNN, etc.). Ils doivent être choisis manuellement ou via une recherche automatisée.

Principe :

1. Définir une grille d'hyperparamètres à tester.
2. Pour chaque combinaison :
  - Entraîner le modèle.
  - Evaluer les performances (avec validation croisée).
3. Sélectionner la combinaison avec les meilleures performances.

Avantages :

- Bonne exploration de l'espace des hyperparamètres.
- Facile à implémenter et à comprendre.
- Garantit de trouver le meilleur point dans la grille définie.

Limitations :

- Coût computationnel élevé.
- Nécessite de définir des plages pertinentes.
- Peut manquer des optimums entre les points de la grille.

**NB** : Il existe des techniques alternatives : **Random Search** ou **Bayesian Optimization** (plus efficaces pour les grands espaces d'hyperparamètres).

# Chap 5 Techniques d' Evaluation et Validation Croisée

## III- Grid search

Exemple :

89

```
from sklearn import datasets
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split

# Charger le dataset
digits = datasets.load_digits()
X = digits.data
y = digits.target
# Séparer en train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Définir les hyperparamètres à tester
param_grid = {'C': [0.1, 1, 10], 'gamma': [0.001, 0.01, 0.1], 'kernel': ['rbf'] }

# Initialiser le modèle SVM (sera traité ultérieurement)
svc = SVC() # la métrique utilisée par défaut par SVC est
             # l'accuracy (le pourcentage de bonnes prédictions).

# Appliquer Grid Search avec validation croisée
grid_search = GridSearchCV(svc, param_grid, cv=5) # k=5
grid_search.fit(X_train, y_train)

# Afficher les meilleurs paramètres
print("Meilleurs paramètres : ", grid_search.best_params_)
print("Meilleure accuracy : ", grid_search.best_score_)
```

GridSearchCV procède comme suit :

1. Il découpe **le jeu d'entraînement** en **5 sous-ensembles (folds)**.
2. Pour chaque combinaison d'hyperparamètres :
  - a. Il entraîne le modèle sur **4 folds**.
  - b. Il le valide sur **le fold restant**.
  - c. Il répète cela 5 fois (donc chaque fold sert une fois de validation).
3. Il moyenne les scores pour estimer les performances pour cette combinaison d'hyperparamètres.
4. Il choisit la **meilleure combinaison** (celle qui maximise le score moyen de validation).
5. Enfin, il **réentraîne le modèle** sur **tout X\_train** avec ces meilleurs paramètres.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## IV- Bias-variance Tradeoff

90

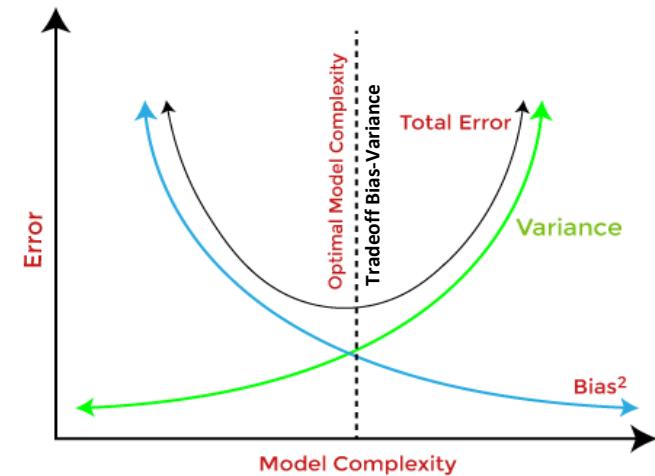
- Le **biais** (bias) est l'erreur provenant d'hypothèses erronées dans l'algorithme d'apprentissage. Un biais élevé peut être lié à un algorithme qui manque de relations pertinentes entre les *features* et la *target* (**sous-apprentissage**).
- La **variance** est l'erreur due à la sensibilité aux petites fluctuations de l'échantillon d'apprentissage. Une variance élevée peut entraîner un **surapprentissage**, i.e., modéliser le bruit aléatoire des données d'apprentissage plutôt que la target prévue.
- La **décomposition Biais-Variance** :

Pour mesurer l'erreur moyenne entre les prédictions d'un modèle et la valeur réelle d'une variable, on utilise MSE (Erreur quadratique moyenne) :  $E[(\hat{y} - y)^2] = (E[\hat{y}] - y)^2 + E[(\hat{y} - E[\hat{y}])^2] + \sigma^2$

- $(E[\hat{y}] - y)^2$  : **Biais<sup>2</sup>**
- $E[(\hat{y} - E[\hat{y}])^2]$  : **Variance**
- $\sigma^2$  : **Bruit irréductible** (erreur due à l'aléa dans les données).

$$\Rightarrow \text{Erreur totale} = \text{Biais}^2 + \text{Variance} + \text{Bruit irréductible}$$

- L'erreur totale a souvent une forme en U :
  - Modèle trop simple  $\Rightarrow$  biais élevé
  - Modèle trop complexe  $\Rightarrow$  variance élevée
  - point optimal  $\Rightarrow$  compromis entre les deux.



# Chap 5 Techniques d' Evaluation et Validation Croisée

## IV- Biais-variance Tradeoff

91

Exemple 1 :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
from sklearn.metrics import mean_squared_error

# Données simulées
np.random.seed(0)
n_samples = 200
X = np.linspace(-1, 1, n_samples).reshape(-1, 1) # transforme un vecteur 1D en une matrice colonne
true_function = lambda x: np.sin(np.pi * x)
y_true = true_function(X).ravel()      # ravel aplatis un tableau multidimensionnel en un vecteur 1D
noise = 0.2 * np.random.randn(n_samples) # Génération d'un bruit aléatoire gaussien moy=0, écart type=0.2
y = y_true + noise

# Degrés à tester
degs = [1, 2, 3, 4, 6, 8, 12]
k= 5 # K-fold
kf = KFold(n_splits=k, shuffle=True, random_state=12)

# Stockage des estimations par degré
biais2_list = []
var_list = []
mse_list = []
deg_list = []
```

Calcul manuel du tradeoff : degré optimal

# Chap 5 Techniques d' Evaluation et Validation Croisée

## IV- Biais-variance Tradeoff

92

```
for d in degs:  
    # Prédictions sur l'ensemble via CV  
    pred_matrix = []    # chaque ligne: prédictions sur tout le jeu X pour un fold  
    for train_index, test_index in kf.split(X):  
        X_train, X_test = X[train_index], X[test_index]  
        y_train, y_test = y[train_index], y[test_index]  
        model = Pipeline([  
            ('poly', PolynomialFeatures(degree=d, include_bias=True)),  
            ('lr', LinearRegression())  
        ])  
        model.fit(X_train, y_train)  
        y_pred = model.predict(X)    # Ce n'est pas "X_test". On voudrait comparer les prédictions de chaque  
                                     # modèle (entraîné sur un fold différent) sur les mêmes points X.  
        pred_matrix.append(y_pred)  
    pred_matrix = np.array(pred_matrix)    # forme: (n_splits, n_samples)  
  
    # Biais2 moyen: moyenne sur les x des ( $\hat{E}_f(x)$ ) -  $f(x)$ )2 :  $E(\hat{f}(x)) \equiv E(\hat{f}(x))$   
    f_hat_mean = pred_matrix.mean(axis=0)  
    biais2 = np.mean((f_hat_mean - y_true) ** 2)  
  
    # Variance moyenne: moyenne sur x de Var[ $\hat{f}(x)$ ]  
    var_per_x = pred_matrix.var(axis=0)  
    var_mean = var_per_x.mean()  
  
    # Erreur moyenne (MSE) sur l'ensemble avec f_hat_mean  
    mse = mean_squared_error(y_true, f_hat_mean)  
  
    biais2_list.append(biais2)  
    var_list.append(var_mean)  
    mse_list.append(mse)  
    deg_list.append(d)
```

# Chap 5 Techniques d' Evaluation et Validation Croisée

## IV- Biais-variance Tradeoff

93

```
# Degré optimal basé sur le risque biais2 + variance
risk = np.array(biais2_list) + np.array(var_list)
opt_index = np.argmin(risk)
optimal_degree = deg_list[opt_index]

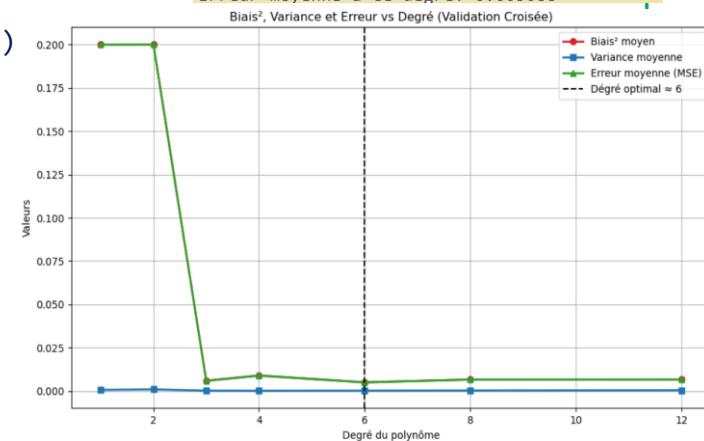
# Affichage des résultats pour chaque degré
print("Résultats par degré :")
print(f'{ "Degré":>6} | { "Biais2":>10} | { "Variance":>10} | { "MSE":>10}')
print("-" * 42)
for d, b2, var, mse in zip(deg_list, biais2_list, var_list, mse_list):
    print(f'{d:>6} | {b2:.6f} | {var:.6f} | {mse:.6f}')

# Affichage du degré optimal
print("\nDegré optimal estimé:", optimal_degree)
print(f"Biais2 moyen à ce degré: {biais2_list[opt_index]:.6f}")
print(f"Variance moyenne à ce degré: {var_list[opt_index]:.6f}")
print(f"Erreur moyenne à ce degré: {mse_list[opt_index]:.6f}")

# Tracé des trois courbes sur un seul graphique
plt.figure(figsize=(10,6))
plt.plot(deg_list, biais2_list, label='Biais2 moyen', color='tab:red', marker='o', linewidth=2)
plt.plot(deg_list, var_list, label='Variance moyenne', color='tab:blue', marker='s', linewidth=2)
plt.plot(deg_list, mse_list, label='Erreur moyenne (MSE)', color='tab:green', marker='^', linewidth=2)
plt.axvline(x=optimal_degree, color='k', linestyle='--', label=f'Degré optimal ≈ {optimal_degree}')
plt.xlabel('Degré du polynôme')
plt.ylabel('Valeurs')
plt.title('Biais2, Variance et Erreur vs Degré (Validation Croisée)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Résultats par degré :			
Degré	Biais <sup>2</sup>	Variance	MSE
1	0.199934	0.000745	0.199934
2	0.199980	0.001046	0.199980
3	0.006006	0.000323	0.006006
4	0.009017	0.000247	0.009017
6	0.005088	0.000314	0.005088
8	0.006709	0.000404	0.006709
12	0.006736	0.000566	0.006736

Degré optimal estimé: 6  
Biais<sup>2</sup> moyen à ce degré: 0.005088  
Variance moyenne à ce degré: 0.000314  
Erreur moyenne à ce degré: 0.005088



# Chap 5 Techniques d' Evaluation et Validation Croisée

## IV- Biais-variance Tradeoff

Exemple 2 :

94

```
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
np.random.seed(0)
n_samples = 100
X = np.linspace(-1, 1, n_samples).reshape(-1, 1)
true_function = lambda x: np.sin(np.pi * x)
y_true = true_function(X).ravel()
noise = 0.2 * np.random.randn(n_samples)
y = y_true + noise
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=42 )
# Pipeline : transformation polynomiale + régression
pipeline = Pipeline([ ('poly', PolynomialFeatures(include_bias=False)), ('lr', LinearRegression()) ])
# Paramètres à tester : degrés de polynôme
param_grid = {'poly_degree': [1, 2, 3, 4, 6, 8, 12]}
# Recherche avec validation croisée
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='neg_mean_squared_error')
grid_search.fit(X_train, y_train)
# Meilleur degré trouvé
best_degree = grid_search.best_params_['poly_degree']
print("Degré optimal trouvé :", best_degree)
```

Calcul professionnel du tradeoff : degré optimal

**NB** : En Scikit-learn, toutes les fonctions de scoring doivent être maximisées. Or, le MSE est une erreur, donc plus petit = meilleur. Pour respecter cette logique, scikit-learn inverse le signe : 'neg\_mean\_squared\_error'

# Chap 5 Techniques d' Evaluation et Validation Croisée

## V- Régularisation (L1, L2)

95

- La **Régularisation** est une technique utilisée pour réduire la variance d'un modèle en limitant sa complexité (réduire l'Overfitting). Elle agit directement sur les **coefficients/poids** du modèle pour les contraindre à rester petits ou à s'annuler.
- On ajoute à la fonction de coût une pénalité sur les poids du modèle :  
$$J(\theta) = \text{Erreur d'entraînement} + \lambda \times \text{Pénalité sur } \theta \Rightarrow J(\theta) = \text{Loss}(\theta) + \lambda R(\theta),$$
  - $R(\theta)$  : Terme de régularisation.
  - $\theta$  : Paramètres du modèle.
  - $\lambda$  : Hyperparamètre de régularisation.
    - si  $\lambda=0 \Rightarrow$  aucune régularisation (risque de surapprentissage),
    - si  $\lambda$  est trop grand  $\Rightarrow$  le modèle devient trop simple (sous-apprentissage).

Justification : L'objectif du modèle est de minimiser la fonction  $J(\theta)$ . Quand  $\lambda$  augmente, le modèle cherche de plus en plus à minimiser  $R(\theta)$  plutôt que l'erreur sur les données. Donc, pour réduire la pénalité, il réduit la valeur des coefficients/poids.

- Si  $\lambda$  est petit  $\Rightarrow$  les coefficients  $\theta$  peuvent s'ajuster librement aux données  $\Rightarrow$  modèle flexible.
- Si  $\lambda$  est grand  $\Rightarrow$  les coefficients sont fortement contraints vers **0**  $\Rightarrow$  le modèle devient presque **constant** (il ne "bouge" plus pour suivre les variations des données).

Résultat :

- Il n'exprime plus la relation réelle entre les variables d'entrée et la sortie.
- Il fait de grosses erreurs même sur les données d'entraînement  $\Rightarrow$  sous-apprentissage (underfitting).

**Résumé** : La **régularisation** agit comme un levier de contrôle sur la **complexité** du modèle. Et comme la complexité influence directement le **biais** et la **variance**, régulariser permet **d'ajuster ce compromis**.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## V- Régularisation (L1, L2)

96

- 2 Principaux types de régularisation : **L1** et **L2**
- Régularisation **L2** – *Ridge* est basée sur la somme des carrés des coefficients :

$$J(\theta) = \text{Erreur} + \lambda \sum_{j=1}^p \theta_j^2$$

- Encourage les petits poids, mais rarement nuls.
- Tendance à répartir la pénalisation entre toutes les variables.
- Utilisée dans :
  - Ridge Regression : Regression linéaire + une pénalisation L2
  - Ridge Classifier
  - Linear SVM avec L2
  - Réseaux de neurones (via weight decay).
- Régularisation **L1** - *Lasso Regression* repose sur la somme des valeurs absolues des coefficients :

$$J(\theta) = \text{Erreur} + \lambda \sum_{j=1}^p |\theta_j|$$

- Encourage des poids nuls  $\Rightarrow$  sélection de variables automatique.
- Donne des modèles plus parcimonieux (simples/légers).
- Utilisée dans :
  - Lasso Regression : Regression linéaire + une pénalisation L1
  - Lasso Logistic Regression

# Chap 5 Techniques d' Evaluation et Validation Croisée

## V- Régularisation (L1, L2)

97

- Régularisation **Mixte - Elastic Net** : Combine L1 et L2 :

$$J(\theta) = \text{Erreur} + \lambda_1 \sum |\theta_j| + \lambda_2 \sum \theta_j^2$$

- Compromis entre **sélection de variables (L1)** et **stabilité numérique (L2)**.
- Régression linéaire : coefficients peuvent être instables et grands.
- Ridge (L2) : coefficients réduits, mais tous non nuls.
- Lasso (L1) : certains coefficients sont exactement nuls → sélection de variables.
- $\lambda$  (souvent noté **alpha** dans scikit-learn) contrôle la force de régularisation.
- On le choisit souvent par validation croisée (ex : RidgeCV, LassoCV, ou GridSearchCV).

**NB** : La régularisation peut ne pas améliorer les performances . Elle est utile lorsque :

- Dataset avec beaucoup de features (high-dimensional)
- Forte colinéarité entre les features
- Présence de bruit dans les données
- Besoin de sélection de features (Lasso)

# Chap 5 Techniques d' Evaluation et Validation Croisée

## V- Régularisation (L1, L2)

98

### Utilisation Professionnelle de la Régularisation

```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge, Lasso, LogisticRegression
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# Préparation des données
data = load_breast_cancer()
X, y = data.data, data.target
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.3, random_state=42, stratify=y )

# Régression linéaire avec régularisation : alpha=lambda
ridge = Ridge(alpha=1.0)
ridge.fit(X_train, y_train)
lasso = Lasso(alpha=0.1)
lasso.fit(X_train, y_train)

print("Ridge train:", ridge.score(X_train, y_train), "test:", ridge.score(X_test, y_test)) # métrique : R2
print("Lasso train:", lasso.score(X_train, y_train), "test:", lasso.score(X_test, y_test)) # métrique : R2

# Régression logistique
log_l2 = LogisticRegression(penalty='l2', C=1.0, solver='liblinear', max_iter=1000)
log_l1 = LogisticRegression(penalty='l1', C=1.0, solver='liblinear', max_iter=1000)
log_none = LogisticRegression(penalty='l2', C=1e6, solver='liblinear', max_iter=1000)
```

**NB :** Dans Scikit-learn, pour Ridge et Lasso on a  $\alpha = \lambda$ . Pour LogisticRegression on a  $C = 1/\lambda$   
⇒ si  $C$  est grand alors régularisation faible. Si  $C$  est petit alors régularisation forte.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## V- Régularisation (L1, L2)

99

### Utilisation Professionnelle de la Régularisation

```
log_l2.fit(X_train, y_train)
log_l1.fit(X_train, y_train)
log_none.fit(X_train, y_train)

print("Log L2 acc:", accuracy_score(y_test, log_l2.predict(X_test))) # métrique : Accuracy
print("Log L1 acc:", accuracy_score(y_test, log_l1.predict(X_test))) # métrique : Accuracy
print("Log sans régularisation acc:", accuracy_score(y_test, log_none.predict(X_test)))

# SVM linéaire :  $C = 1/\lambda$ 
svm = SVC(C=1.0, kernel='linear')
svm.fit(X_train, y_train)
print("SVM acc:", accuracy_score(y_test, svm.predict(X_test)))

# MLPClassifier (réseau de neurones) : alpha=  $\lambda$ 
mlp = MLPClassifier(hidden_layer_sizes=(50,), alpha=0.01, max_iter=1000, random_state=42)
mlp.fit(X_train, y_train)
print("MLP acc:", mlp.score(X_test, y_test))
```

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VI- Métriques d'évaluation pour la Classification



1. **La matrice de confusion** est un outil fondamental qui présente les prédictions correctes et incorrectes :

	Prédit : Négatif (0)	Prédit : Positif (1)
Réel : Négatif (0)	Vrais Négatifs (TN)	Faux Positifs (FP)
Réel : Positif (1)	Faux Négatifs (FN)	Vrais Positifs (TP)

Le choix des métriques à optimiser dépend du coût relatif de ces erreurs. Exemple :

- FP : Un email important (ham) est mis dans les spams. Coût élevé !
- FN : Un spam passe dans la boîte de réception. Coût faible.

2. **Exactitude (Accuracy)** : Pourcentage de prédictions correctes

$$Accuracy = (VP + VN) / (VP + FN + FP + VN) \in [0,1]$$

Utilisation : Uniquement lorsque les **classes** sont parfaitement **équilibrées** et que le coût des **FP** et **FN** est **similaire**.

3. **Précision (Precision)** :  $Precision = VP / (VP + FP) \in [0,1]$  : Proportion de vrais positifs parmi les prédictions positives. Minimiser les **Faux Positifs**. "Quand le modèle décide oui, à quel point est-il sûr?" .

Utilisation : Détection de spam (éviter les FP), systèmes de recommandation (montrer seulement les résultats pertinents).

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VI- Métriques d'évaluation pour la Classification

101

**4. Rappel/Sensibilité (Recall/ Sensitivity)** : capacité du modèle à trouver tous les cas positifs réels :

$$\text{Recall} = VP / (VP + FN) \in [0,1]$$

Signification : Parmi tous les véritables positifs, combien le modèle en a retrouvés ?

Objectif : Minimiser les **Faux Négatifs**.

Utilisation : Diagnostic médical (ne manquer aucun cas de maladie), rappel de produits défectueux, etc.

Exemple : détecter une maladie rare chez **100** patients. **10** patients sont réellement malades (positifs). Le modèle en détecte **8** correctement (**VP=8**),mais il en manque 2 (**FN=2**) . **Recall=**  $8 / (8+2) = 8/10 = 0.8 = 80\% \rightarrow$  modèle a donc identifié **80 % des cas malades**.

**5. F1-Score** : Moyenne harmonique de la précision et du rappel (elle doit être élevée) :

$$F1 = \frac{2 \times (\text{Precision} \times \text{Recall})}{(\text{Precision} + \text{Recall})} \leq 2\min(\text{Precision}, \text{Recall})$$

Signification : Une métrique unique qui cherche un équilibre entre les deux :

- Si la précision est élevée mais le rappel très faible  $\Rightarrow$  F1 sera faible.
- Si le rappel est élevé mais la précision très faible  $\Rightarrow$  F1 sera faible.

Utilisation : Parfait lorsqu'on a un déséquilibre des classes et que l'on veut une seule mesure pour comparer des modèles. Il pénalise les modèles qui sacrifient une métrique pour l'autre.

# Chap 5 Techniques d'Evaluation et Validation Croisée

## VI- Métriques d'évaluation pour la Classification

102

- La fonction `classification_report` dans `scikit-learn` est pratique. Elle résume plusieurs métriques d'évaluation de classification en un tableau lisible.

Exemple :

```
from sklearn.metrics import classification_report
# Vérités et prédictions (exemple binaire)
y_true = [0, 1, 0, 1, 0, 1, 1, 0]
y_pred = [0, 1, 0, 0, 0, 1, 1, 1]
# Rapport de Classification
print(classification_report(y_true, y_pred, target_names=["Classe 0", "Classe 1"]))
```

	precision	recall	f1-score	support
Classe 0	0.75	0.75	0.75	4
Classe 1	0.75	0.75	0.75	4
accuracy			0.75	8
macro avg	0.75	0.75	0.75	8
weighted avg	0.75	0.75	0.75	8

- Support** : Nombre d'échantillons réels dans cette classe.
- macro avg** : Moyenne simple (non pondérée) des métriques de toutes les classes.
- weighted avg** : Moyenne pondérée selon le nombre d'exemples (support) dans chaque classe.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VI- Métriques d'évaluation pour la Classification

103

### 6. ROC (Receiver Operating Characteristic)

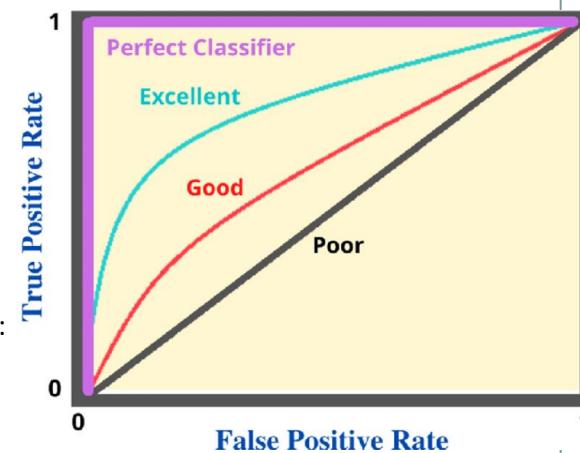
- La régression logistique retourne une probabilité. Pour en faire une classe (0 ou 1), on doit choisir un seuil (par défaut 0.5). Si probabilité  $\geq 0.5$  alors Classe 1, sinon Classe 0. Mais le seuil **0.5** est arbitraire ! Le seuil de **0.5** vient du cas **symétrique** où : les classes ont la même importance (même coût d'erreur), et les classes sont équilibrées.
- La Courbe **ROC** trace le taux de vrais positifs **TPR** (True Positive Rate) =  $VP / (VP+FN)$  = **Recall** en fonction du taux de faux positifs **FPR** (False Positive Rate) =  $FP / (FP+TN)$ , pour différents seuils de classification.
- Un bon modèle aura un **TPR** proche de **1** et un **FPR** proche de **0**. La courbe sera donc **proche du coin supérieur gauche**. Une diagonale  $\Rightarrow$  modèle aléatoire. En dessous de la diagonale  $\Rightarrow$  pire qu'un hasard.
- Comment tracer la courbe ROC ?

Supp. que `y_test =[1, 0, 1, 0]` et `y_scores =[0.9, 0.8, 0.4, 0.1]`.

Les seuils testés seront (en général) les valeurs uniques triées

Seuil (threshold)	Prédictions binaires	$TPR = TP / (TP+FN)$	$FPR = FP / (FP+TN)$
$\geq 0.9$	[1, 0, 0, 0]	0.5	0.0
$\geq 0.8$	[1, 1, 0, 0]	0.5	0.5
$\geq 0.4$	[1, 1, 1, 0]	1.0	0.5
$\geq 0.1$	[1, 1, 1, 1]	1.0	1.0

Donc les points (FPR, TPR) de la courbe sont :  
(0.0, 0.5), (0.5, 0.5), (0.5, 1.0), (1.0, 1.0)



# Chap 5 Techniques d' Evaluation et Validation Croisée

## VI- Métriques d'évaluation pour la Classification

104

- Utilité de ROC :
  - i. Evaluer la performance globale d'un modèle
  - ii. Comparer plusieurs modèles en traçant plusieurs courbes **ROC** sur le même graphique.
  - iii. Choisir un seuil de classification optimal : *ROC* permet de tester différents seuils, et de choisir celui qui équilibre sensibilité et spécificité selon le besoin.  $\text{Spécificité} = VN / (VN + FP)$
  - iv. Adapter le modèle aux enjeux métier : *ROC* est utile pour arbitrer selon : le coût d'une fausse alerte (faux positif) et le coût d'un faux négatif (ne pas détecter une fraude, une maladie, etc.).
  - v. Déetecter un modèle aléatoire ou biaisé : *ROC* proche de la **diagonale** indique un modèle inefficace. Une courbe en **dessous** de la diagonale  $\Rightarrow$  modèle **inverse** ou **biaisé**.

### 7. AUC (Area Under the Curve)

- **AUC** : Aire sous la courbe *ROC*. C'est est un nombre entre **0** et **1** .
- L'**AUC** est la probabilité qu'un classifieur donne une probabilité plus élevée à un exemple positif qu'à un exemple négatif choisi au hasard, quel que soit le seuil de décision :
  - Plus il est proche de **1**, meilleur est le modèle (i.e. ROC s'approche du coin sup. gauche)
  - Un modèle **aléatoire** a un **AUC  $\approx 0.5$**
  - Un modèle **parfait** a un **AUC = 1.0**
  - Modèle **inverse** **AUC < 0.5** (classe les exemples à l'envers).

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VI- Métriques d'évaluation pour la Classification

105

Exemple : un modèle qui prédit la probabilité de **spam**

Email	Vraie classe	Score prédition (probabilité spam)
e1	1 (spam)	0.95
e2	1 (spam)	0.90
e3	0 (ham)	0.40
e4	0 (ham)	0.20
e5	0 (ham)	0.10

- $TPR = TP / (TP + FN)$  et  $FPR = FP / (FP + TN)$
- Les **vrais exemples de la classe 1 (positive)** : e1 et e2  $\Rightarrow$  scores positifs = [0.95, 0.90].
- Les **vrais exemples de la classe 0** : e3, e4, e5  $\Rightarrow$  scores négatifs = [0.40, 0.20, 0.10].

- On remarque que tous les *scores positifs > scores négatifs*  $\Rightarrow$  modèle **parfait**. En effet :

On voit qu'il existe une "zone vide" entre **0.90** et **0.40**. Si l'on prend un seuil dans cet intervalle (ex. **0.7**) :

- Tous les positifs ( $\geq 0.90$ ) sont au-dessus du seuil  $\Rightarrow$  prédits comme positifs  $\Rightarrow TPR = 1$ .
- Tous les négatifs ( $\leq 0.40$ ) sont en dessous du seuil  $\Rightarrow$  prédits comme négatifs  $\Rightarrow FPR = 0$ .  
 $\Rightarrow$  on est exactement au point **(0, 1)** dans le plan **ROC**.

Résumé :

- Seuil très haut (plus grand que 0.95)  $\Rightarrow$  personne n'est classé positif  $\Rightarrow (FPR=0, TPR=0)$ .
- Seuil entre max(négatifs) et min(positifs) (ex. 0.7)  $\Rightarrow$  séparation parfaite  $\Rightarrow (FPR=0, TPR=1)$ .
- Seuil très bas (plus petit que 0.1)  $\Rightarrow$  tout le monde est classé positif  $\Rightarrow (FPR=1, TPR=1)$ .

La courbe **ROC** : points **(0,0)  $\rightarrow$  (0,1)  $\rightarrow$  (1,1)** (Coïncide avec angle sup. gauche)  $\Rightarrow$  **AUC = 1**

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VI- Métriques d'évaluation pour la Classification

106

### Remarques :

- Toujours vérifier l'équilibre des classes dans les données.
- Ne jamais utiliser l'**Accuracy** seule pour évaluer un modèle de classification.
- Toujours examiner la **matrice de confusion** pour comprendre la nature des erreurs.
- Choisir la métrique en fonction du problème métier : Vaut-il mieux éviter les **Faux Positifs** (optimiser la **Precision**) ou éviter les **Faux Négatifs** (optimiser le **Recall**) ?

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VII- Métriques d'évaluation pour la Régression

107

- Contrairement à la classification où l'on mesure l'Exactitude/Accuracy, en régression on évalue à quel point les prédictions sont proches des vraies valeurs. Ici, on prédit des valeurs continues (prix, précipitations, etc.).

### 1. Mean Absolute Error (MAE) - Erreur Absolue Moyenne

- MAE** mesure la moyenne des écarts absolus entre les prédictions et les vraies valeurs.

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$$

- Avantage : Facile à interpréter, dans la **même unité** que la variable cible.
- Inconvénient : Ne pénalise pas les grandes erreurs de manière disproportionnée.

### 2. Mean Squared Error (MSE) - Erreur Quadratique Moyenne

- MSE** mesure la moyenne des carrés des écarts. En élevant au carré, on donne plus de poids aux grandes erreurs.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

- Avantage : Pénalise fortement les grandes erreurs (utile quand les grosses erreurs sont inacceptables)
- Inconvénients :
  - Plus difficile à interpréter (**unité au carré**).
  - Sensible aux outliers : Une seule grande erreur peut dominer la métrique.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VII- Métriques d'évaluation pour la Régression

108

### 3. Root Mean Squared Error (RMSE) - Racine de l'Erreur Quadratique Moyenne

- Le **RMSE** est la racine carrée du MSE, ce qui ramène la métrique dans l'unité originale.

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$$

Avantage : Même unité que la variable cible, plus interprétable que le MSE.

Inconvénient : Toujours sensible aux outliers

**Remarque** :  $MAE \leq RMSE$  (Inégalité de Cauchy-Schwarz).

### 4. R<sup>2</sup> Score (Coefficient de Détermination)

- Le **R<sup>2</sup>** mesure la proportion de variance expliquée par le modèle. R<sup>2</sup> compare l'erreur de prédiction du modèle à une référence très simple qui est : "toujours prédire la moyenne des y".

$$R^2 = \frac{\text{Erreur résiduelle (non expliquée)}}{\text{Variabilité totale}} = \frac{SSR}{SST} = 1 - \frac{SSE}{SST} = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

- Si  $R^2 = 0$ , le modèle ne fait pas mieux que la prédiction naïve (la moyenne). Si  $R^2 = 1$ , le modèle prédit parfaitement. Si  $R^2 < 0$ , le modèle fait pire que de simplement prédire la moyenne.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VII- Métriques d'évaluation pour la Régression

109

- $R^2$  dépend de la dispersion des données : un même modèle peut avoir un  $R^2$  élevé sur un jeu très variable (car il capture beaucoup de variance) mais faible sur un jeu plus homogène, même si ses erreurs absolues sont identiques :

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

Exemple : Supposons deux jeux de données avec le même modèle, erreur moyenne = 10.

- Dataset A : les valeurs de  $y$  varient beaucoup (entre 0 et 1000) :

Variance énorme  $\Rightarrow$  l'erreur de 10 est faible par rapport à la dispersion  $\Rightarrow R^2$  proche de 1.

- Dataset B : les valeurs de  $y$  varient peu (entre 0 et 20) :

Variance faible  $\Rightarrow$  l'erreur de 10 est énorme par rapport à la dispersion  $\Rightarrow R^2$  peut être négatif.

Dans les deux cas, le **MAE** = 10, mais le  $R^2$  change radicalement. D'où :

- **MAE/RMSE** sont absous (donnent une erreur mesurable en unités de la cible).
- $R^2$  est relatif (dépend du contexte de la variance de la cible).

### Cas où $R^2$ est utile :

- i. Comparer le modèle à une baseline simple (prédirer la moyenne).  $R^2$  dit directement si le modèle apporte de l'information ou non. Si  $R^2 \leq 0$ , alors le modèle est pire que la baseline.
- ii. Mesurer la proportion de variabilité expliquée. Exemple : un modèle de régression linéaire avec  $R^2 = 0.75 \Rightarrow$  on peut dire que "75 % de la variabilité de Y est expliquée par les variables X".

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VII- Métriques d'évaluation pour la Régression

110

- iii. Quand les unités de la cible ne sont pas parlantes. Si l'on prédit une grandeur abstraite (ex. un score, un indice), dire "l'erreur moyenne est **2.65**" n'a pas beaucoup de sens. Par contre, "le modèle explique **85 %** de la variance" est plus facile à comprendre.
- iv. Pour comparer plusieurs modèles sur le même jeu de données. Ex. "Régression linéaire" et "Arbre de décision". Le modèle avec le  $R^2$  le plus élevé explique le plus de variabilité  $\Rightarrow$  il "suit" mieux les données.

### Cas où $R^2$ n'est pas suffisant :

- i. Si on a besoin d'une évaluation pratique des erreurs. Ex. : prévision de température  $\rightarrow$  "MAE = 1,5°C" est beaucoup plus informatif que dire " $R^2 = 0.85$ ".
- ii. Si l'on compare des jeux de données différents. Un  $R^2=0.8$  sur des données très dispersées  $\neq$  un  $R^2 =0.8$  sur des données homogènes.
- iii. En régression non linéaire complexe : le sens de "variance expliquée" devient moins clair.

### 4. Mean Absolute Percentage Error (MAPE) - Erreur Absolue Pourcentage Moyenne

- **MAPE** exprime l'erreur en pourcentage de la valeur réelle. Il indique en moyenne, de combien de % les prédictions s'éloignent de la réalité :  
$$MAPE = \frac{100}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$
- Avantage : MPAE est un pourcentage indépendant de toute unité. En plus, il est compréhensible même par les non-experts. Ex. MAPE = 5 % signifie que le modèle se trompe en moyenne de 5 % par rapport à la vraie valeur.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VIII- Synthèse de Métriques d'évaluation

### 1. Métriques pour la Classification Binaire

111

Métrique	Formule	Interprétation	Avantages	Limites
Exactitude (Accuracy)	$(VP + VN) / (VP + FN + FP + VN)$	Proportion totale de prédictions correctes.	Intuitive et facile à comprendre. Bonne métrique lorsque les classes sont équilibrées.	Très trompeuse si les classes sont déséquilibrées (ex: 99% de classe A, 1% de classe B).
Précision (Precision)	$VP / (VP + FP)$	Parmi les prédictions positives, combien sont réellement positives.	Cruciale lorsque le coût des Faux Positifs est élevé (ex: spam marqué comme important).	N'évalue pas la capacité à trouver tous les positifs. Peut être élevée même si on rate beaucoup de VP.
Rappel (Recall) ou Sensibilité	$VP / (VP + FN)$	Parmi les vrais positifs, combien ont été correctement identifiés.	Cruciale lorsque le coût des Faux Négatifs est élevé (ex: diagnostic d'une maladie).	Peut être élevé au prix de nombreux Faux Positifs. Ne tient pas compte des FP.
Score F1 (F1-Score)	$2 * (Precision * Recall) / (Precision + Recall)$	Moyenne harmonique de la Précision et du Rappel.	Bon compromis entre Précision et Rappel. Adapté aux jeux de données déséquilibrés.	Donne le même poids à la Précision et au Rappel, ce qui n'est pas toujours souhaitable.
Spécificité	$VN / (VN + FP)$	Capacité à identifier correctement les négatifs.	Importante pour évaluer la performance sur la classe négative.	Souvent utilisée avec le Rappel pour une vision complète.
Courbe ROC & AUC	Surface sous la courbe ROC (trace le Taux Vrais Positifs (Recall) vs Taux Faux Positifs (1 - Spécificité)).	Capacité du modèle à distinguer les classes, quel que soit le seuil de classification.	Indépendante du seuil de classification. Excellente pour comparer des modèles.	Peut être trop optimiste pour les jeux de données très déséquilibrés.
Coefficient Kappa de Cohen	$(Précision observée - Précision aléatoire) / (1 - Précision aléatoire)$	Mesure l'accord entre les prédictions et les vérités terrain, en corrigeant le "hasard".	Tient compte du déséquilibre des classes. Plus robuste que l'accuracy.	Moins intuitif que l'accuracy. Interprétation contextuelle.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VIII- Synthèse de Métriques d'évaluation

### 2. Métriques pour la Classification Multi-Classe

112

Métrique	Formule / Concept	Interprétation	Avantages	Limites
Précision Globale	Identique au cas binaire.	Proportion totale de prédictions correctes.	Simple et directe.	Masque les performances sur des classes spécifiques, surtout si elles sont rares.
Précision/Rappel Moyens (Macro-Average)	Moyenne arithmétique des métriques (Précision, Rappel, F1) calculées pour chaque classe indépendamment.	Donne le même poids à chaque classe, indépendamment de sa taille.	Bonne lorsque toutes les classes sont également importantes.	Peut être pénalisée artificiellement par de mauvaises performances sur une petite classe.
Précision/Rappel Moyens (Micro-Average)	Calcule les métriques globales en agrégant tous les VP, FP, FN de toutes les classes.	Donne plus de poids aux classes plus fréquentes.	Reflète mieux la performance globale sur l'ensemble des échantillons.	Les performances sur les petites classes peuvent être "noyées" par celles des grandes classes.
Perte Logistique (Log Loss)	$-\frac{1}{n} \sum_{i=1}^n [y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)]$	Pénalise les prédictions "confiantes mais erronées".	Mesure fine de la qualité des probabilités prédites.	Difficile à interpréter directement. Très sensible aux mauvaises prédictions confiantes.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VIII- Synthèse de Métriques d'évaluation

### 3. Métriques pour la Régression

113

Métrique	Formule	Interprétation	Avantages	Limites
Erreur Quadratique Moyenne (MSE)	$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$	Moyenne des carrés des erreurs. Mesure l'ampleur moyenne des erreurs au carré.	- Différentiable, idéale pour l'optimisation - Pénalise fortement les grandes erreurs - Sensible aux outliers (peut être un avantage)	- Non interprétable dans l'unité originale - Très sensible aux valeurs aberrantes - Donne plus de poids aux grandes erreurs
Racine de l'Erreur Quadratique Moyenne (RMSE)	$RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$	Racine carrée du MSE. Écart type des erreurs de prédiction.	- Interprétable dans l'unité de la variable cible - Même échelle que les données - Pénalise les grandes erreurs	- Toujours sensible aux outliers - Donne un poids disproportionné aux grandes erreurs - Non différentiable en zéro
Erreur Absolute Moyenne (MAE)	$MAE = \frac{1}{n} \sum_{i=1}^n  y_i - \hat{y}_i $	Moyenne de la magnitude absolue des erreurs.	- Robuste aux outliers - Facile à interpréter - Même échelle que les données - Différentiable presque partout	- Non différentiable en zéro - Ne pénalise pas suffisamment les grandes erreurs dans certains cas
Pourcentage d'Erreur Absolue Moyenne (MAPE)	$MAPE = \frac{100}{n} \sum_{i=1}^n \left  \frac{y_i - \hat{y}_i}{y_i} \right $	Pourcentage moyen d'erreur absolue par rapport aux vraies valeurs.	- Interprétable en pourcentage - Sans dimension (comparaison entre jeux de données) - Facile à expliquer aux non-experts	- Non défini quand $y_i = 0$ - Pénalise asymétriquement les surestimations/sous-estimations - Instable pour les petites valeurs de $y_i$
Coefficient de Détermination ( $R^2$ )	$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$	Proportion de la variance de la variable dépendante expliquée par le modèle.	- Sans échelle (comparaison entre modèles) - Interprétation intuitive (0% à 100%) - Seuil de référence : modèle de la moyenne	- Peut être trompeur avec peu d'échantillons - Augmente avec l'ajout de variables, même non pertinentes - Peut être négatif si le modèle est très mauvais
Erreur Absolue Médiane (MedAE)	$MedAE = median( y_1 - \hat{y}_1 ,  y_2 - \hat{y}_2 , \dots,  y_n - \hat{y}_n )$	Médiane des erreurs absolues.	- Extrêmement robuste aux outliers - Représente l'erreur "typique" - Bonne pour les distributions asymétriques	- Moins sensible aux variations des données - Non différentiable - Moins utilisée, donc moins de références
Erreur Quadratique Moyenne Logarithmique (MSLE)	$MSL = \frac{1}{n} \sum_{i=1}^n (\log(1 + \hat{y}_i) - \log(1 + y_i))^2$	MSE appliquée aux logarithmes des valeurs.	- Pénalise relativement les erreurs - Bonne pour les données à large plage - Moins sensible aux valeurs extrêmes	- Complexé à interpréter - Non symétrique

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VIII- Synthèse de Métriques d'évaluation

### 4. Choix des Métriques

114

- i. Problème déséquilibré ?  $\Rightarrow$  Eviter l'**Accuracy**. Préférer le **F1-Score**, la **Précision**, le **Rappel** ou l'**AUC**.
- ii. Coût des Faux Positifs élevé ? (ex: spam)  $\Rightarrow$  Optimiser la **Précision**.
- iii. Coût des Faux Négatifs élevé ? (ex: fraude, maladie)  $\Rightarrow$  Optimiser le **Rappel**.
- iv. Besoin d'un compromis ?  $\Rightarrow$  Utiliser le **Score F1**.
- v. Besoin de comparer des modèles de façon générale ?  $\Rightarrow$  **AUC-ROC** est un excellent choix pour la classification binaire.
- vi. Régression et sensibilité aux outliers ?  $\Rightarrow$  Préférer la **MAE** ou la **MedAE** à la **MSE/RMSE**.
- vii. Évaluer la qualité des probabilités ?  $\Rightarrow$  Utiliser la **Log Loss** (classification) ou le **R<sup>2</sup>** (régression pour la variance expliquée).

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VIII- Synthèse de Métriques d'évaluation

### 4. Choix des Métriques

115

Type de problème	Exemples de modèles	Objectif du modèle	Métriques principales	Autres métriques utiles / Remarques
Classification binaire	Logistic Regression, SVM, Random Forest, XGBoost, Réseaux de neurones	Prédire deux classes (ex. : spam / non-spam)	Accuracy, Precision, Recall, F1-score, AUC-ROC	- Accuracy si classes équilibrées.- F1-score si déséquilibrées.- ROC-AUC pour la qualité globale de séparation.- Precision-Recall AUC si déséquilibre fort.
Classification multiclasse	Softmax Regression, Random Forest, CNN (images)	Prédire parmi plusieurs classes (ex. : type d'animal)	Accuracy, Macro / Weighted F1-score	- Confusion matrix pour interprétation.- Top-k accuracy (vision).
Classification multilabel	Modèles multi-sorties, BERT (NLP)	Plusieurs étiquettes possibles par instance	Hamming Loss, Subset Accuracy, Micro/Macro F1-score	- Jaccard Index utile pour évaluer la similarité entre ensembles de labels.
Régression	Linear Regression, Random Forest Regressor, XGBoost Regressor	Prédire une valeur continue	RMSE (Root Mean Squared Error), MAE (Mean Absolute Error), R <sup>2</sup> (Coefficient de détermination)	- RMSE pénalise les grandes erreurs.- MAE plus robuste aux outliers.- R <sup>2</sup> pour interprétabilité.
Régression robuste / avec outliers	RANSAC, quantile regression	Résistance aux valeurs aberrantes	MAE, Huber loss, Quantile loss	- Éviter MSE qui amplifie les outliers.
Clustering (non supervisé)	K-Means, DBSCAN, GMM	Regrouper les données sans labels	Silhouette Score, Davies-Bouldin Index, Calinski-Harabasz Index	- Si labels connus : ARI, NMI, Homogeneity.

# Chap 5 Techniques d' Evaluation et Validation Croisée

## VIII- Synthèse de Métriques d'évaluation

### 4. Choix des Métriques

116

Type de problème	Exemples de modèles	Objectif du modèle	Métriques principales	Autres métriques utiles / Remarques
Réduction de dimension	PCA, t-SNE, UMAP	Représentation compacte des données	Explained Variance Ratio, Reconstruction Error	- Évaluer la séparabilité visuelle (2D/3D).
Détection d'anomalies	Isolation Forest, One-Class SVM, Autoencoder	Identifier les points atypiques	Precision, Recall, F1-score, AUC-ROC, PR-AUC	- Souvent fort déséquilibre → privilégier Recall / PR-AUC.
Recommandation	Collaborative filtering, Matrix factorization	Prédire des notes ou classements	RMSE, MAE, Precision@k, Recall@k, MAP, NDCG	- NDCG et MAP mieux adaptées aux systèmes de ranking.
Segmentation d'image	U-Net, Mask R-CNN	Prédire un masque pixel par pixel	IoU (Intersection over Union), Dice Coefficient (F1)	- Pixel accuracy utile pour vision globale.
Détection d'objet	YOLO, Faster R-CNN	Localiser et classifier des objets	mAP (mean Average Precision)	- Parfois mesuré à différents seuils d'IoU (ex. mAP@0.5).
Séries temporelles (prévision)	ARIMA, LSTM, Prophet	Prédire des valeurs futures	RMSE, MAE, MAPE (Mean Absolute Percentage Error)	- MAPE utile pour interprétation en %.
Traitement du langage naturel (NLP)	BERT, GPT, Seq2Seq	Traduction, résumé, QA	BLEU, ROUGE, METEOR, Perplexity	- BLEU/ROUGE : comparaison de texte.- Perplexity : qualité du modèle de langage.

# Chap 6 Arbres de décision et Forêts aléatoires

## I- Arbres de Décision (Decision Tree)

117

- Un *Arbre de décision* est une structure arborescente qui permet de prendre une décision ou de prédire une étiquette en séparant les données selon des tests successifs sur les features :
  - Les feuilles représentent les décisions finales (labels) ou les valeurs prédites.
  - Les nœuds internes représentent les tests conditionnels.
- Contrairement aux réseaux de neurones qui sont des boîtes noires, les arbres de décision, sont transparents et faciles à interpréter. Ils fonctionnent avec moins de données et consomment moins de ressources de calcul.
- Les *Arbres de classification* prédisent des résultats *catégoriels* à partir de variables numériques et catégorielles.
- Les *Arbres de régression* prédisent des résultats *numériques* à partir de variables numériques et catégorielles
- Les arbres de décision sont souvent utilisés pour résoudre des problèmes de classification, mais peuvent également être utilisés comme modèle de régression pour prédire des résultats numériques.
- Avantages: interprétabilité, entraînement rapide, peu ou pas de prétraitement nécessaire, capable de gérer des types de données variés.
- Limites: tendance au surapprentissage si l'arbre est profond, sensibilité aux petites fluctuations des données, performances parfois inférieures à des méthodes en ensemble (par ex. forêts aléatoires).

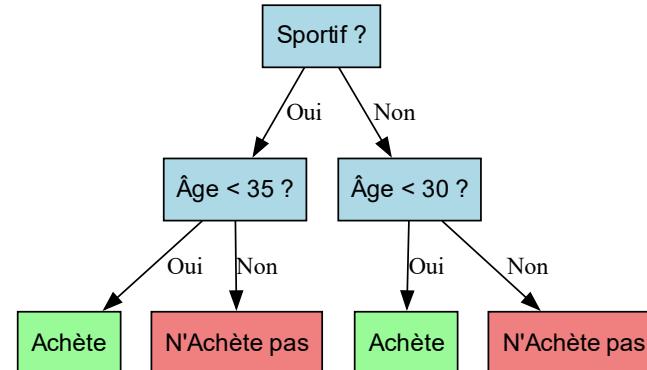
# Chap 6 Arbres de décision et Forêts aléatoires

## I- Arbre de Décision

118

Exemple : On veut prédire si un client achètera un **jus bio** à partir de 3 caractéristiques simples :

- Age du client (en années)
- Revenu mensuel (en DH)
- Sportif ? (Oui / Non)



- **Critères de division** : L'algorithme essaie plusieurs divisions possibles et, pour chacune il calcule l'impureté du nœud avant et après la division, puis il choisit celle qui réduit le plus l'**impureté** :
  - **Gini impureté**:  $Gini = 1 - \sum_{i=1}^C p_i^2$ , où  $p_i$  est la proportion des exemples appartenant à la classe  $i \in [1, C]$ . Si  $Gini = 0 \Rightarrow$  nœud **pur** (toutes les données appartiennent à une seule classe). Plus le  $Gini$  est grand plus le nœud est impur.  $Gain = Gini_{avant} - Gini_{après}$ . On choisit la séparation (variable + seuil) qui maximise ce gain (réduction d'impureté).
  - **Gain d'information** (ou entropie): mesure de la pureté obtenue en séparant les données par une caractéristique.  $Entropie = -\sum_{i=1}^C p_i \log_2(p_i)$ .  $Gain = Entropie_{avant} - Entropie_{après}$ . On choisit la variable et le seuil (la séparation) qui donne le gain d'information le plus élevé.
  - **Gain calculé sur des critères spécifiques** (par exemple, pour les régressions, réduction de l'erreur quadratique moyenne).

# Chap 6 Arbres de décision et Forêts aléatoires

## I- Arbre de Décision

119

**NB :** Plus un arbre est profond, plus il peut apprendre des détails (et du bruit) du jeu de données. S'il est trop profond il va mémoriser les données d'entraînement au lieu de généraliser (Overfitting).

- **Variantes avancées :**
  - **Random Forest** : Ensemble d'arbres aléatoires  $\Rightarrow$  vote majoritaire
  - **Gradient Boosting/XGBoost /LightGBM** : Arbres ajoutés successivement pour corriger les erreurs.
- **Algorithme de construction :**
  - i. Départ : collecte des données étiquetées (features X et label y).
  - ii. Sélection du meilleur test : à chaque nœud, évaluer toutes les features et choisir celle qui maximise le critère de division (par exemple le gain d'information).
  - iii. Application du test et répartition : les données sont séparées selon le seuil ou la condition choisie.
  - iv. Répétition récursive : appliquer le même processus aux sous-ensembles jusqu'à atteindre une condition d'arrêt (Exple : pureté des feuilles, profondeur maximale, ou nombre minimum d'exemples par feuille).
  - v. Arbre complet ou arrêté : on obtient un arbre qui peut être directement utilisé pour prédire les nouvelles observations.

## II- Arbre de Décision pour la classification

120

Exemple (construction manuelle) : Prédire si une personne achètera un ordinateur en fonction des features : Age, Revenu, Etudiant, et Crédit.

Dataset :

ID	Age	Revenu	Etudiant	Crédit	Achat
1	Jeune	Elevé	Non	Mauvais	Non
2	Jeune	Elevé	Non	Bon	Non
3	Jeune	Moyen	Non	Mauvais	Oui
4	Jeune	Faible	Oui	Bon	Oui
5	Moyen	Faible	Oui	Bon	Oui
6	Agé	Moyen	Oui	Bon	Oui
7	Agé	Elevé	Non	Bon	Oui
8	Agé	Faible	Oui	Mauvais	Non
9	Moyen	Faible	Oui	Mauvais	Oui
10	Jeune	Moyen	Oui	Bon	Oui

Etape 1 : Calcul de l'entropie initiale :

On a 10 exemples : 7 "Oui" et 3 "Non" :

$$\text{Entropie}(S) = -p_{\text{Oui}} \log_2(p_{\text{Oui}}) - p_{\text{Non}} \log_2(p_{\text{Non}})$$

$$= -\frac{7}{10} \log_2 \left( \frac{7}{10} \right) - \frac{3}{10} \log_2 \left( \frac{3}{10} \right) \approx 0.881$$

## II- Arbre de Décision pour la classification

121

**Etape 2 :** Calcul du gain d'information pour chaque feature :

Feature : **Age**

- **Jeune** : 5 exemples : 3 Oui, 2 Non  $\Rightarrow$  Entropie = 0.971
- **Moyen** : 2 exemples : 2 Oui  $\Rightarrow$  Entropie = 0
- **Agé** : 3 exemples : 2 Oui, 1 Non  $\Rightarrow$  Entropie = 0.918

$$\text{Gain}(Age) = 0.881 - \left( \frac{5}{10} \cdot 0.971 + \frac{2}{10} \cdot 0 + \frac{3}{10} \cdot 0.918 \right) = \textcolor{red}{0.1201}$$

Feature : **Revenu**

- **Elevé** : 3 exemples : 2 Non, 1 Oui  $\Rightarrow$  Entropie = 0.918
- **Moyen** : 3 exemples : 3 Oui  $\Rightarrow$  Entropie = 0
- **Faible** : 4 exemples : 3 Oui, 1 Non  $\Rightarrow$  Entropie = 0.811

$$\text{Gain}(Revenu) = 0.881 - \left( \frac{3}{10} \cdot 0.918 + \frac{3}{10} \cdot 0 + \frac{4}{10} \cdot 0.811 \right) = \textcolor{blue}{0.2812}$$

Feature : **Etudiant**

- **Oui** : 6 exemples : 5 Oui, 1 Non  $\Rightarrow$  Entropie = 0.650
- **Non** : 4 exemples : 2 Oui, 2 Non  $\Rightarrow$  Entropie = 1

$$\text{Gain}(Etudiant) = 0.881 - \left( \frac{6}{10} \cdot 0.650 + \frac{4}{10} \cdot 1 \right) = \textcolor{red}{0.091}$$

Feature : **Crédit**

- **Bon** : 6 exemples : 5 Oui, 1 Non  $\Rightarrow$  Entropie = 0.650
- **Mauvais** : 4 exemples : 2 Oui, 2 Non  $\Rightarrow$  Entropie = 1

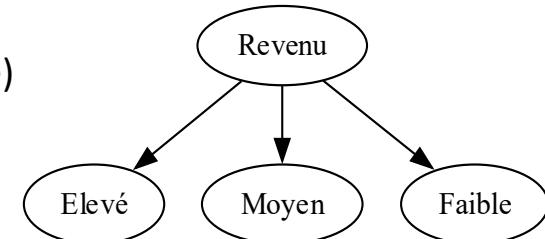
$$\text{Gain}(Crédit) = 0.881 - \left( \frac{6}{10} \cdot 0.650 + \frac{4}{10} \cdot 1 \right) = \textcolor{red}{0.091}$$

## II- Arbre de Décision pour la classification

122

**Etape 3 :** Choix du 1<sup>er</sup> critère (Construction de la **racine**/1<sup>er</sup> niveau de l'arbre)

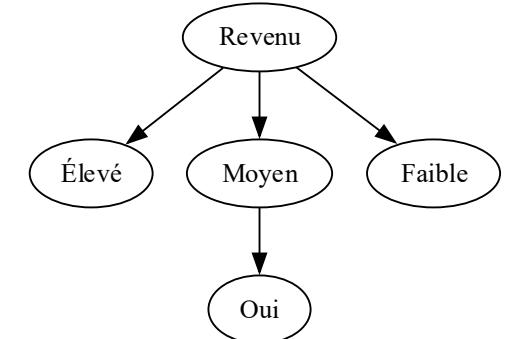
Le **gain** le plus élevé est pour **Revenu** ⇒ c'est le **nœud racine**.



**Etape 4 :** Branche "**Revenu = Moyen**"

ID	Achat
3	Oui
6	Oui
10	Oui

Tous les exemples sont "Oui" ⇒ **Feuille (pure) : Oui**



**Etape 5 :** Branche "**Revenu = Elevé**"

3 instances (1 oui et 2 non) :

ID	Achat
1	Non
2	Non
7	Oui

$$\text{Entropie initiale pour cette branche} = -p_{Oui} \cdot \log_2(p_{Oui}) - p_{Non} \cdot \log_2(p_{Non}) = -\frac{1}{3} \cdot \log_2\left(\frac{1}{3}\right) - \frac{2}{3} \cdot \log_2\left(\frac{2}{3}\right) = 0,918.$$

Calculons les gains pour les attributs restants (**Age**, **Etudiant**, **Crédit**) pour identifier la meilleure feature qui séparera ces 3 instances :

## II- Arbre de Décision pour la classification

123

- Feature **Age** : On a 3 instances de "revenu élevé"

- **Jeune** (2 instances) : Oui=0 et Non=2  $\Rightarrow$  Entropie = 0  $\Rightarrow$  feuille pure(Achat=Non)
- **Agé** (1 instance) : Oui=1 et Non=0  $\Rightarrow$  Entropie = 0  $\Rightarrow$  feuille pure (Achat=Oui)

$$\text{Entropie pondérée}(Age) = (2/3)*0 + (1/3)*0 = 0$$

$$\text{Gain}(Age) = 0.918 - 0 = 0.918$$

- Feature **Etudiant** : On a 3 instances de "revenu élevé"

- **Oui** (0 instances) : aucun étudiant n'a un "revenu élevé"
- **Non** (3 instances) : Oui=1, Non=2  $\Rightarrow$  Entropie = 0.918.

$$\text{Gain}(Etudiant) = 0.918 - 0.918 = 0$$

- Feature **Crédit** : On a 3 instances de "revenu élevé"

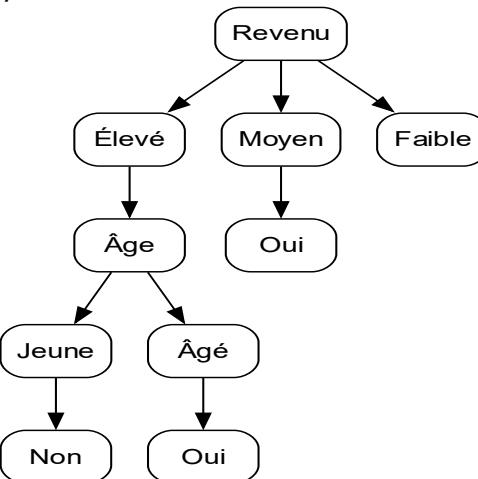
- **Mauvais** (1 instance) : (0 Oui, 1 Non)  $\Rightarrow$  entropie =0  $\Rightarrow$  feuille pure : Achat=Non)
- **Bon** (2 instances) : (1 Oui, 1 Non)  $\Rightarrow$  entropie = $-0.5\log0.5-0.5\log0.5=1$ .

$$\text{Entropie pondérée après split par Crédit} : 1/3*0+2/3*1=0.666.$$

$$\text{Gain (Crédit)}=0.918-0.666=0.252.$$

$\Rightarrow$  Meilleure feature : **Age** (gain = 0.918)  $\Rightarrow$  On ajoute alors le nœud "**Age**" en dessous du nœud "**Elevé**".

Les 2 branches du nœud "Age" ont des entropies nulles  $\Rightarrow$  feuilles pures (**Jeune** : Non et **Agé** : Oui). Donc l'arbre se développe comme indiquée dans la figure.



## II- Arbre de Décision pour la classification

124

## Etape 6 : Branche "Revenu = Faible"

4 instances de "revenu faible" : Oui=3, Non=1

Entropie locale = 0,811.

Calculons les gains des features restantes : **Age**, **Etudiant** et **Crédit** (un attribut peut figurer plusieurs fois dans 1 arbre) :

- Feature **Age** : 4 instances de "revenu faible"

- **Jeune** (1 instance) : 1 Oui et 0 Non  $\Rightarrow$  Feuille pure (Achat=Oui). Entropie=0
  - **Moyen** (2 instances) : 2 Oui et 0 Non  $\Rightarrow$  Feuille pure (Achat=Oui). Entropie=0
  - **Agé** (1 instance) : 0 Oui et 1 Non  $\Rightarrow$  Feuille pure (Achat=Non). Entropie=0
- $\Rightarrow$  chacun des sous-groupes est pur  $\Rightarrow$  Entropie après split = 0  $\Rightarrow$  Gain = 0,811 (split parfait).

- Feature **Etudiant** : 4 instances de "revenu faible"

- **Oui** (4 instances): 3 (Achat=Oui) et 1 (Achat Non)  $\Rightarrow$  Entropie= 811.
  - **Non** (0 instance) : 0 Oui et 0 Non  $\Rightarrow$  Entropie=0
- $\Rightarrow$  Entropie= 811  $\Rightarrow$  Gain=0 (pas séparable).

- Feature **Crédit** : 4 instances de "revenu faible"

- **Bon** (2 ex., 2 Oui et 0 Non)  $\Rightarrow$  Entropie=0
- **Mauvais** (2 ex., 1 Oui,1 Non)  $\Rightarrow$  Entropie= 1

$$\text{Entropie} = 2/4 * 0 + 2/4 * 1 = 0,5 \quad \text{Gain} = 0,811 - 0,5 = 0,311.$$

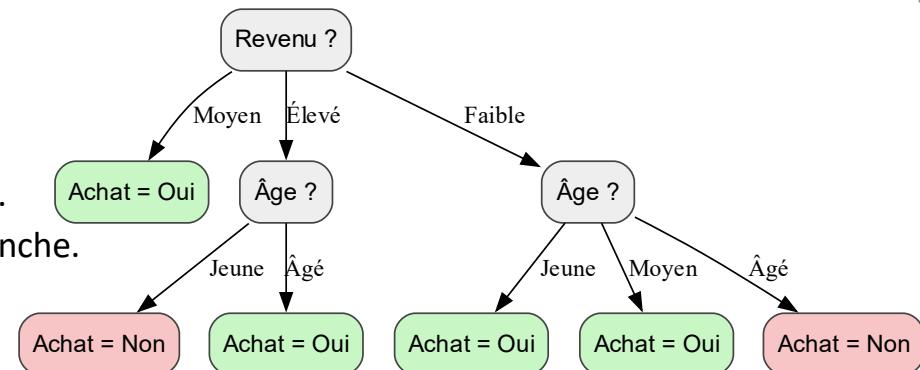
Donc on choisit **Age** (le gain le plus élevé) pour cette branche.

Feuilles pour "Revenu=Faible" :

Age = Jeune  $\Rightarrow$  Achat = Oui

Age = Moyen  $\Rightarrow$  Achat = Oui

Age = Agé  $\Rightarrow$  Achat = Non



## II- Arbre de Décision pour la classification

125

**NB:** Quand aucune feature ne permet de réduire l'entropie (ou que toutes les features restantes ont la même valeur), on crée une feuille avec la classe majoritaire.

⇒ **Arbre final (profondeur = 3) :**

**Exemples de Règles décisionnelles extraites :**

- Si Revenu = Moyen → prédire Oui.
- Si Revenu = Elevé ET Age = Jeune → prédire Non.
- Si Revenu = Faible ET Age = Agé → prédire Non.
- Sinon (beaucoup de combinaisons restantes) → prédire Oui.

## II- Arbre de Décision pour la classification

126

## Exemple 1 (Classification):

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt

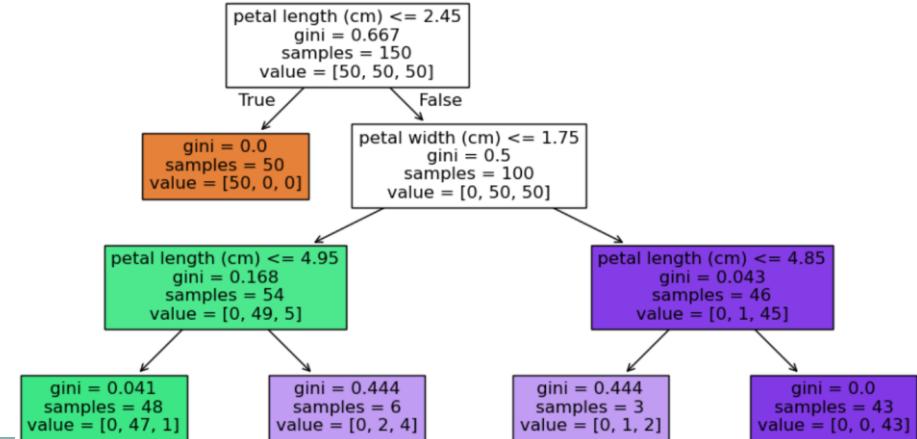
# Charger le dataset des fleurs Iris
X, y = load_iris(return_X_y=True) # charge le dataset Iris et sépare directement les données (X) et les étiquettes (y)

# Créer un arbre de décision
classif = DecisionTreeClassifier(max_depth=3)
classif.fit(X, y)

# Visualiser l'arbre
plt.figure(figsize=(12, 6))
plot_tree(classif, filled=True, feature_names=load_iris().feature_names)
plt.show()
```

Classe dominante	Couleur approximative	Interprétation
Setosa	Orange / brun	Ech. majoritairement Setosa
Versicolor	Vert / bleu-vert	Ech. Majoritairement Versicolor
Virginica	Violet	Ech. majoritairement virginica

- Chaque **nœud** de l'arbre contient : Un test sur une caractéristique (ex. : petal length (cm) <= 2.45).
- L'indice gini, qui mesure l'impureté (0 = pur, 0.5 = mélange parfait de 2 classes).
- Le nombre d'échantillons (samples) arrivant à ce nœud.
- La distribution des classes (value = [setosa, versicolor, virginica]).
- Les couleurs sont générées automatiquement par **sklearn.tree.plot\_tree**, qui colore chaque nœud en fonction de la classe majoritaire et de la pureté (plus la couleur est saturée, plus le nœud est pur).



## II- Arbre de Décision pour la classification

127

## Interprétation de l'arbre de décision:

## 1. Racine de l'arbre

**Condition :** petal length (cm)  $\leq 2.45$

Gini = 0.667, samples = 150, value = [50, 50, 50]  $\Rightarrow$  Les 150 fleurs sont parfaitement équilibrées entre les 3 espèces.

## Branche gauche (True) :

- Toutes les fleurs ont un pétales  $\leq 2.45$  cm
- Gini = 0.0, samples = 50, value = [50, 0, 0]

Cela correspond à **Iris setosa** (Orange) : toutes les fleurs de cette espèce ont de petits pétales  $\Rightarrow$  classification parfaite.

## Branche droite (False) :

- Pétales  $> 2.45$  cm  $\Rightarrow$  correspondent aux deux autres espèces (versicolor et virginica).
- Gini = 0.5, samples = 100, value = [0, 50, 50].

## 2. Deuxième niveau (branche droite)

Test suivant : petal width (cm)  $\leq 1.75$

## Branche gauche (True)

Gini = 0.168, samples = 54, value = [0, 49, 5]  $\Rightarrow$  Majoritairement **versicolor** (vert), quelques **virginica** mal classées.

Sous-branches :

- Si petal length  $\leq 4.95 \Rightarrow$  gini = 0.041, presque toutes versicolor (vert clair)  
2 feuilles : 48 Versicolor  $\rightarrow$  vert saturé ET 6 samples (2 versicolor, 4 virginica)  
 $\rightarrow$  violet clair car Virginica est majoritaire ici

## Branche droite (False)

Gini = 0.043, samples = 46, value = [0, 1, 45]  $\Rightarrow$  Majoritairement **Iris virginica**.

Sous-branches : Si petal length  $\leq 4.85 \Rightarrow$  petit mélange : 2 feuilles :

- 3 samples (1 versicolor, 2 virginica)  $\rightarrow$  violet clair ET
- 43 virginica  $\rightarrow$  violet saturé (pureté maximale)

Spèce prédictée	Règle de décision principale	Pureté
Setosa	petal length $\leq 2.45$	100 %
Versicolor	petal length $> 2.45$ et petal width $\leq 1.75$	~90–95 %
virginica	petal length $> 2.45$ et petal width $> 1.75$	~95–100 %

## III- Arbre de Décision pour la Regression (DecisionTreeRegressor)

128

- L'arbre de décision pour la régression prédit une valeur continue au lieu d'une classe.
- Principe :
  1. A chaque nœud, l'algorithme cherche la meilleure coupure sur une caractéristique.
  2. Toutes les coupures sont testées (entre valeurs triées). Si une feature contient [2.0, 2.5, 3.7, 4.1] → seuils testés seront : **2.25, 3.1, 3.9**
  3. Le meilleur seuil est celui qui réduit le plus l'impureté dans les sous-groupes.
- Mesure d'impureté utilisée : Contrairement à la classification (Gini / Entropie), la régression utilise une erreur continue : **MSE(Variance)** ou **MAE**. L'objectif est de minimiser l'erreur dans les nœuds enfants.
- La valeur prédite dans une feuille est la moyenne des valeurs cibles dans cette feuille :

$$\hat{y} = \frac{1}{N} \sum y_i , \quad (N : \text{nombre d'échantillons dans la feuille}).$$

- Hyperparamètres importants :
  - `max_depth` : profondeur maximale
  - `min_samples_split` : taille minimale d'un nœud à diviser
  - `min_samples_leaf` : taille minimale d'une feuille

Avantages	Limites	Utilisations courantes
Simple à comprendre	Sensible au bruit	Prédiction de prix (immobilier, produits...)
Interprétable	Modèle peu stable	Prévision de quantités (demande, ventes)
Pas besoin de normalisation	Fonction par morceaux irrégulière	Modélisation de tendances non linéaires
Gère les non-linéarités	Risque élevé de surapprentissage	Analyse de séries non linéaires

## III- Arbre de Décision pour la Regression (DecisionTreeRegressor)

129

## Exemple 2 (Regression):

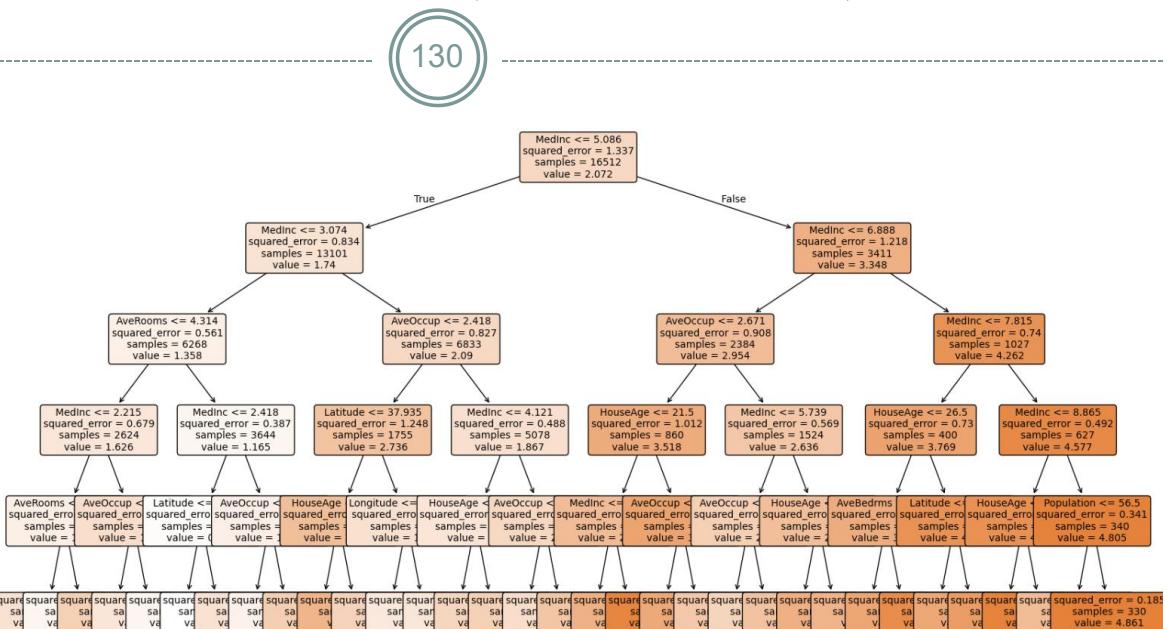
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error
from sklearn.datasets import fetch_california_housing
import matplotlib.pyplot as plt
from sklearn import tree
from sklearn.tree import export_text
# On utilise California Housing qui est le remplaçant moderne du Boston dataset
data = fetch_california_housing(as_frame=True)
df = data.frame
X = df.drop("MedHouseVal", axis=1)
y = df["MedHouseVal"]
# Séparation train/test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Modèle arbre de décision pour la régression
model = DecisionTreeRegressor(max_depth=5, random_state=42)
model.fit(X_train, y_train)
# Prédictions
y_pred = model.predict(X_test)
# Erreur quadratique moyenne
mse = mean_squared_error(y_test, y_pred)
print("MSE :", mse)
```

# Chap 6 Arbres de décision et Forêts aléatoires

## III- Arbre de Décision pour la Régression (DecisionTreeRegressor)

### Exemple 2 (Regression):

```
print("MSE : ", mse)
plt.figure(figsize=(20, 10))
tree.plot_tree(
    model,
    feature_names=X.columns,
    filled=True,
    rounded=True,
    fontsize=10
)
plt.show()
```



### Interprétation :

L'arbre de décision va automatiquement :

- Découper l'espace des caractéristiques (ex. LSTAT, RM, CRIM...)
- Créer des règles du style :  
*Si LSTAT < 12.5 et RM > 6.2  $\Rightarrow$  prix  $\approx$  32.1*
- Approcher la valeur du prix par moyennes dans les feuilles.

Avec une profondeur limitée (`max_depth=5`), on évite le surapprentissage.

**NB** : Avec "California Housing", un MSE autour  $\in [0.3, 0.5]$  est courant selon la profondeur.

# Chap 6 Arbres de décision et Forêts aléatoires

## II- Forêts aléatoires (Random Forests)

131

- Les **forêts aléatoires** sont une **méthode d'ensemble** qui combine plusieurs arbres de décision pour améliorer la performance prédictive et réduire le risque de surapprentissage.
  - Plutôt que de construire un seul arbre, on en construit des centaines, chacun sur un échantillon aléatoire des données et avec une sélection aléatoire des variables à chaque nœud.
  - Résultat final : la prédiction est obtenue par vote majoritaire (classification) ou moyenne (régression) des prédictions de tous les arbres.
- **Principe :**
  1. **Bootstrap Aggregating (Bagging)** : on génère plusieurs sous-échantillons des données d'entraînement (avec remplacement). Chaque arbre est entraîné sur un échantillon différent  $\Rightarrow$  diversité des modèles.
  2. **Sélection aléatoire des variables** : à chaque nœud, on ne considère qu'un sous-ensemble aléatoire de variables pour déterminer la meilleure séparation. Cela évite que tous les arbres soient identiques et favorise la diversité.
  3. **Agrégation des prédictions :**
    - **Classification** : chaque arbre vote pour une classe, la classe majoritaire est choisie.
    - **Régression** : on fait la moyenne des prédictions des arbres.

# Chap 6 Arbres de décision et Forêts aléatoires

## II- Forêts aléatoires (Random Forests )

132

### Algorithme

1. Pour  $b = 1$  à  $B$  (nombre d'arbres) :
  - Échantillonner  $n$  observations avec remise → jeu bootstrap  $D_b$ .
  - Construire un arbre  $T_b$  sur  $D_b$  :
    - À chaque nœud, sélectionner au hasard  $m$  variables parmi  $p$  (mtry).
    - Trouver la meilleure coupe parmi ces  $m$  variables et créer les sous-nœuds.
    - Poursuivre jusqu'au critère d'arrêt (profondeur max, n\_min, pureté).
2. Pour une observation nouvelle  $x$  :
  - Régression : prédiction = moyenne des prédictions  $T_b(x)$ .
  - Classification : prédiction = vote majoritaire des  $T_b(x)$ .

Complexité temporelle : Pour  $B$  arbres,  $n$  observations,  $p$  variables, profondeur moyenne  $D$ , nombre de variables testées par nœud (max\_features)  $m$ , la complexité est :

$$O(BmnD) = O(Bn \log nm)$$

Paramètre de Forêt aléatoire	Description et effet
<b>n_estimators (B)</b>	Nombre d'arbres dans la forêt. - Plus grand $\Rightarrow$ meilleure stabilité et performance plus robuste. - Inconvénient : augmente le temps et le coût de calcul. - Régler jusqu'à stabilisation de la performance.
<b>max_features (mtry)</b>	Nombre de variables (features) testées à chaque nœud. Règles courantes : - Classification : $\sqrt{p}$ ( $p$ : nb total des features) - Régression : $p/3$ Plus petit $\Rightarrow$ plus de diversité entre arbres $\Rightarrow$ variance plus faible, mais biais plus élevé.
<b>max_depth, min_samples_split, min_samples_leaf</b>	Paramètres de complexité des arbres individuels. - Limiter la profondeur ou imposer des tailles minimales permet de réduire le surapprentissage (overfitting).
<b>bootstrap</b>	Indique si chaque arbre est entraîné sur un échantillon bootstrap (tirage avec remise). - <b>True</b> (par défaut) : active le bagging (diversité entre arbres). - <b>False</b> : bagging désactivé, moins courant.
<b>oob_score</b>	Si <b>True</b> , calcule l'erreur sur les échantillons non utilisés (out-of-bag). - Permet une estimation interne de la performance sans validation croisée.
<b>n_jobs</b>	Nombre de processeurs utilisés pour le calcul parallèle. - <b>Utile pour accélérer</b> l'entraînement sur de grands jeux de données. - <b>Valeur recommandée</b> : -1 pour utiliser tous les coeurs disponibles.

# Chap 6 Arbres de décision et Forêts aléatoires

## II- Forêts aléatoires (Random Forests )

133

- Avantages des Forêts Aléatoires :
  - Robuste au surapprentissage : grâce à la diversité des arbres.
  - Bonne performance : souvent meilleure qu'un seul arbre de décision.
  - Gère les données manquantes et les variables non linéaires.
  - Estimation de l'importance des variables : permet d'identifier les variables les plus influentes.
- Inconvénients :
  - Moins interprétable qu'un arbre unique.
  - Temps de calcul plus élevé (surtout avec beaucoup d'arbres) et Taille mémoire importante.

### Exemple 1 : Classification

```
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

X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)
rf = RandomForestClassifier(n_estimators=200, max_features='sqrt', oob_score=True, random_state=42, n_jobs=-1)
rf.fit(X_train, y_train)
print("OOB accuracy:", rf.oob_score_)
y_pred = rf.predict(X_test)
print("Test accuracy:", accuracy_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

# Chap 6 Arbres de décision et Forêts aléatoires

## II- Forêts aléatoires (Random Forests )

134

### Exemple 2 : Regression

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error
import numpy as np

data = fetch_california_housing()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target, test_size=0.2,
random_state=0)

rf_reg = RandomForestRegressor(
    n_estimators=300,
    max_features=0.33,      # environ p/3
    n_jobs=-1,    # pour utiliser tous les coeurs disponibles de l'ordinateur (parallelisme)
    random_state=0,
    oob_score=True     # calculer l'erreur sur les échantillons non utilisés
)
rf_reg.fit(X_train, y_train)

y_pred = rf_reg.predict(X_test)
print("RMSE:", np.sqrt(mean_squared_error(y_test, y_pred)))
print("OOB R^2:", rf_reg.oob_score_) #oob_score_: coefficient de détermination R2 calculé sur les prédictions OOB.
```

# Chap 6 Arbres de décision et Forêts aléatoires

## II- Forêts aléatoires (Random Forests )

135

### Conseils pratiques :

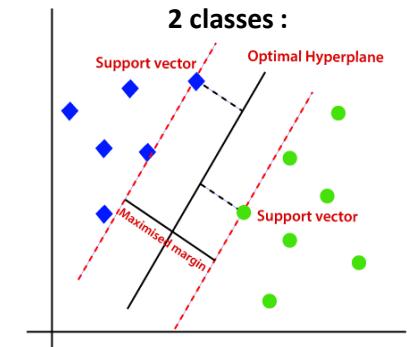
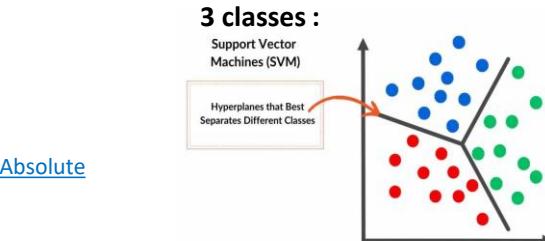
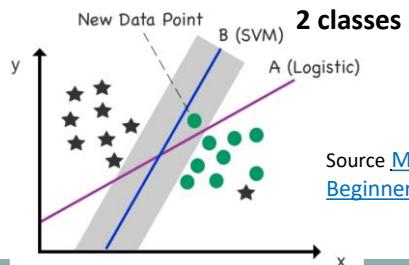
- **Normalisation** : pas nécessaire pour Random Forests.
- **Features catégoriques** : encodage *one – hot* possible ; pour nombreuses catégories, préférer encodage ordinal avec précautions.
- **Jeu déséquilibré** : ajuster *class\_weight = 'balanced'* ou rééchantillonner.
- **Fixer random\_state** pour reproductibilité.
- **Utiliser OOB** pour estimer la performance sans CV si dataset grand.
- **Hyperparam tuning** : privilégier *n\_estimators*, *max\_features*, *max\_depth*, *min\_samples\_leaf*. Faire *GridSearch*/*RandomSearch* sur un sous-ensemble si coûts élevés :

```
from sklearn.model_selection import GridSearchCV
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [10, 20, None],
    'min_samples_split': [2, 5, 10]
}
grid_search = GridSearchCV(RandomForestClassifier(), param_grid, cv=5)
grid_search.fit(X_train, y_train)
print(f"Meilleurs paramètres : {grid_search.best_params_}")
```

## I- Introduction

136

- Les **SVM** (Support Vector Machines) sont des algorithmes d'apprentissage **supervisé** utilisés pour la classification et la régression. Ils sont particulièrement efficaces pour les problèmes de classification non linéairement séparables.
- Les SVM cherchent à déterminer la frontière optimale, appelée **hyperplan**, qui sépare les différentes classes de données.
- Pour une classification binaire, un hyperplan est un sous-espace qui divise l'espace de caractéristiques multidimensionnel en deux moitiés, chacune contenant les points de données d'une seule classe.
- Les **Vecteurs de support** sont les points de données les plus proches de l'hyperplan. Ils "supportent" l'hyperplan et déterminent sa *position* et la *marge* dans le SVM.
- La **Marge** est la distance entre l'hyperplan et les *vecteurs de support*. L'objectif principal des SVM est de maximiser la **marge** entre les deux classes. Plus cette marge est grande, meilleures sont les performances du modèle sur des données nouvelles et inconnues.



Source : <https://medium.com/@ambika199820/support-vector-machine-svm>

## II- Classification SVM Linéaire

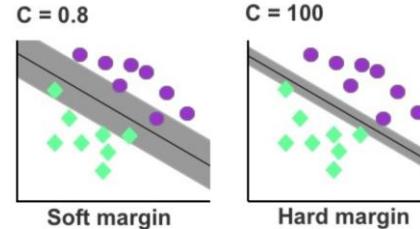
137

- L'équation d'un hyperplan dans un espace de caractéristiques peut être représentée comme suit :

$$\omega^T x + b = 0, \text{ où :}$$

- $\omega$  : vecteur normal à l'hyperplan (direction perpendiculaire à celui-ci).
- $x$  : un point de données.
- $b$  : le terme de biais (distance de l'hyperplan par rapport à l'origine le long du vecteur normal  $\omega$ ).

- La **distance** d'un point de données  $x$  à l'hyperplan peut être calculée comme la projection de  $x$  sur le vecteur normal  $\omega$ , divisée par la norme de  $\omega$  :  $d = \frac{|\omega^T x + b|}{\|\omega\|}$
- La **marge** correspond à la distance entre les deux hyperplans parallèles passant par les vecteurs de support des deux classes.
- L'objectif est de trouver les valeurs optimales de  $\omega$  et  $b$  définissant l'hyperplan tout en respectant certaines contraintes. La marge doit être maximisée  $\Rightarrow$  minimiser la valeur absolue de  $\omega$ .
- Marge dure** : Un hyperplan à marge maximale qui sépare parfaitement les données sans erreurs de classification.



Source [Machine Learning For Absolute Beginners \(O. Theobald\)](#)

- Marge souple** : Permet certaines erreurs de classification en introduisant des variables d'écart, équilibrant la maximisation de la marge et les pénalités de mauvaise classification lorsque les données ne sont pas parfaitement séparables.

## II- Classification SVM Linéaire

138

- **C** : Terme de régularisation équilibrant la maximisation de la marge et les pénalités de mauvaise classification. Une valeur de C plus élevée impose une pénalité plus sévère pour les erreurs de classification.
- **Perte charnière** : Une fonction de perte pénalisant les points mal classés ou les violations de marge et est combinée avec la régularisation dans SVM : *Fonction objectif* =  $(1/\text{marge}) + \lambda \sum \text{peine}$
- Le problème d'optimisation du SVM impose que, pour chaque donnée  $(x_i, y_i)$  avec  $y_i \in \{+1, -1\}$  :

$$y_i(\omega^T x + b) \geq 1$$

On aurait pu prendre n'importe quelle constante  $c > 0$  à la place de "1", mais en SVM, on normalise les contraintes pour que la marge soit exactement  $= 2 / \|\omega\|$ .

- Si  $y_i = +1$  (classe positive) :  $(\omega^T x + b) \geq 1$ .      **Classificateur SVM linéaire**
- Si  $y_i = -1$  (classe négative) :  $(\omega^T x + b) \leq -1$ .

Ces inégalités garantissent que les points ne sont pas seulement du bon côté de l'hyperplan central, mais qu'ils se trouvent au-delà d'une bande de largeur 2 autour de lui.

Les points qui "touchent" cette bande sont les **vecteurs de support**.

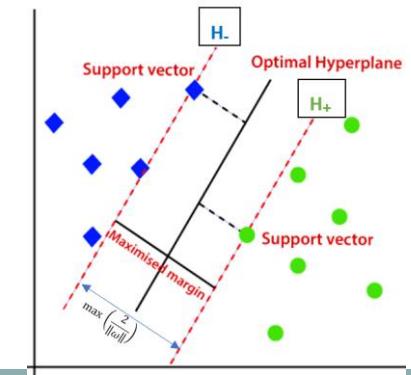
Ceux de la classe positive satisfont exactement :  $w^T x + b = +1 : H_+$

Ceux de la classe négative satisfont exactement :  $w^T x + b = -1 : H_-$

Prenons  $x_+ \in H_+ \Leftrightarrow w^T x_+ + b = 1$ .

La distance orthogonale à un hyperplan le long de la normale  $w$  vaut, pour un point

$$x_0 : d(x_0, H_-) = \frac{|w^T x_0 + b - (-1)|}{\|w\|} \text{ et } d(H_+, H_-) = \frac{|w^T x_+ + b + 1|}{\|w\|} = \frac{|1+1|}{\|w\|} = \frac{2}{\|w\|}.$$



# Chap 7 Machines à vecteurs de support (SVM)

## II- Classification SVM Linéaire

139

- L'idée du SVM est de maximiser cette marge, ce qui correspond à minimiser  $\|\omega\|$ . Puisque pour  $\omega \geq 0$  on a :  $\text{argmin } \|\omega\| = \text{argmin } \|\omega\|^2 = \text{argmin } \frac{1}{2} \|\omega\|^2$ .
- on préfère la version avec le carré de la norme car  $\|\omega\|$  n'est pas différentiable partout. En plus,  $\frac{1}{2} \|\omega\|^2 = \frac{1}{2} \omega^T \omega$  est **quadratique**, **lisse**, strictement **convexe**, parfaitement **differentiable partout**.  $\Rightarrow$  C'est beaucoup plus **simple** et **stable** à optimiser.
- Formulation du problème d'optimisation (cas linéaire séparé) :

problème du SVM linéaire dur (hard-margin SVM) :

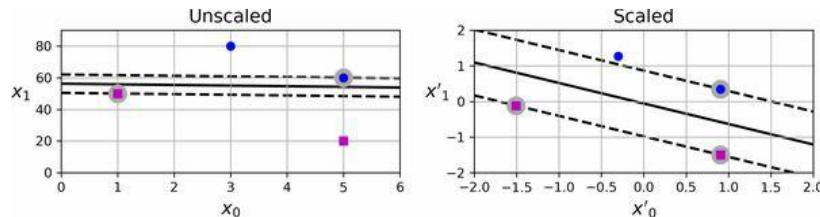
$$\min_{\omega, b} \frac{1}{2} \|\omega\|^2 \text{ sous contraintes } y_i(\omega^T x_i + b) \geq 1, \text{ pour } i = 1, 2, \dots, m$$

$y_i$  : étiquette de classe (+1 ou -1) pour chaque instance d'entraînement.

$x_i$  : vecteur de caractéristiques pour la  $i$ -ème instance d'entraînement.

$m$  : nombre total d'instances d'entraînement.

- Les SVM sont sensibles à l'échelle des caractéristiques :



Source [Aurélien Géron - Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow](#)

l'échelle verticale est beaucoup plus grande que l'échelle horizontale, de sorte que la marge le plus large possible est proche de l'horizontale. Après une mise à l'échelle des caractéristiques, la frontière de décision sur le graphique de droite est bien meilleure

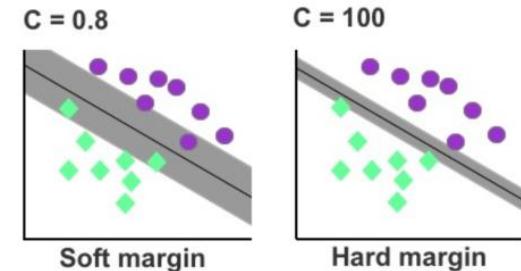
## II- Classification SVM Linéaire

140

## Classification à marge souple :

- La Classification à **marge stricte** exige que toutes les instances soient hors de la marge et du bon côté, mais cela présente des inconvénients :

- Ne fonctionne que si les données sont linéairement séparables.
- Très sensible aux valeurs aberrantes. Une seule valeur aberrante peut :
  - rendre impossible la définition d'une marge stricte,
  - ou modifier fortement la frontière de décision,
  - entraînant un risque de mauvaise généralisation.



[Source Machine Learning For Absolute Beginners \(O. Theobald\)](#)

- Besoin d'un modèle plus flexible : Trouver un compromis : **maximiser la largeur de la marge tout en limitant les violations** (points dans la marge ou du mauvais côté).  $\Rightarrow$  **classification à marge souple** :

On minimise :  $(1/2) \|w\|^2 + C \times \text{somme}(\text{des violations})$ .

- Marge souple et hyperparamètre C** : C contrôle le compromis entre largeur de marge et violations. Il correspond à une forme de **régularisation**. Ici  $C \equiv 1/\lambda$  :

- Régularisation classique (Ridge/Lasso), on minimise :  $(1/2) \|w\|^2 + \lambda \times \text{somme}(\text{des erreurs})$ .

- Régularisation SVM** :  $\min_{\omega, b} \frac{1}{2} \|\omega\|^2 + C \sum_{i=1}^m \xi_i$ , Sous contraintes :

$$\begin{cases} y_i(\omega^T x_i + b) \geq 1 - \xi_i \\ \xi_i \geq 0, \quad \forall i \in [1, m]. \end{cases}$$

$\xi_i$ : variables de relâchement (slack variables)

## II- Classification SVM Linéaire

141

**Faible C :**

- Marge plus large.
- Plus grand nombre de violations.
- Moindre risque de surapprentissage.
- Trop faible  $\Rightarrow$  sous-apprentissage.

**C élevé :**

- Marge plus étroite.
- Moins de violations.
- Risque accru de surapprentissage.
- Exemple :  $C = 100$  généralise mieux que  $C = 1$ .

**Exemple 1 :** classificateur SVM linéaire pour détecter les fleurs d'Iris Virginica

```
from sklearn.datasets import load_iris
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 2) # Iris virginica

svm_classif = make_pipeline(StandardScaler(), LinearSVC(C=1, random_state=42))
svm_classif.fit(X, y)

X1 = [[5.2, 1.5], [5.4, 1.2]]
svm_classif.predict(X1)
svm_classif.decision_function(X1) # Scores mesurant la distance signée (positive/négative)
# entre chaque instance et la frontière de décision
```

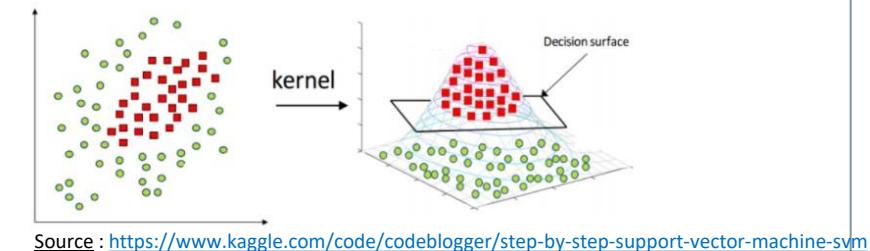
**NB:** `LinearSVC` ne fournit pas de méthode `predict_proba()` pour estimer des probabilités. Mais avec la classe `SVC` avec `probability=True`.

## III- Classification SVM Non linéaire

142

- La classification SVM non linéaire est utilisée lorsque les données ne sont pas séparables par une frontière linéaire.
- Principe :
  - Projeter (mapper) les données dans un espace de dimensions plus élevées où elles deviennent séparables.
  - Ou utiliser une technique appelée l' **astuce du noyau** (kernel trick) qui calcule implicitement les produits scalaires dans cet espace élevé sans construire explicitement les coordonnées de cet espace.

Cette projection est faite **implicitement** grâce à une **fonction noyau (kernel)** :  $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$ , où  $\phi(x)$  est la transformation vers l'espace supérieur.

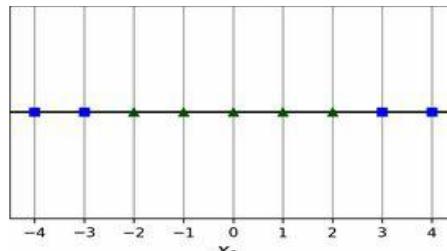


Nom du noyau	Formule	Quand l'utiliser
Linéaire	$K(x, x') = x \cdot x'$	Données déjà séparables linéairement
Polynomiale	$K(x, x') = (x \cdot x' + c)^d$	Données avec interactions entre variables
RBF (Gaussian / Radial Basis Function)	$K(x, x') = e^{-\gamma \ x-x'\ ^2}$	Cas général, très populaire
Sigmoïde	$K(x, x') = \tanh(ax \cdot x' + c)$	Similaire aux réseaux de neurones

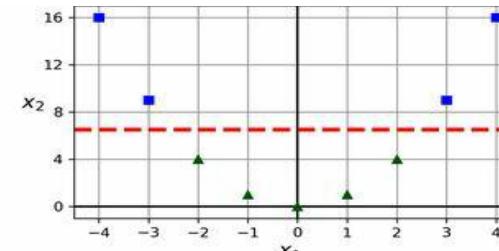
## III- Classification SVM Non linéaire

143

- Une des solutions pour un dataset non linéairement séparable est d'ajouter des **caractéristiques supplémentaires** (par ex. des **polynômes**) pour le rendre linéairement séparable.



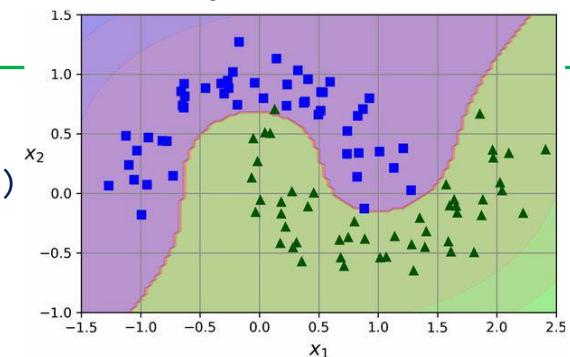
Une seule Feature  $x_1$  (1D): Données Non linéairement séparables.



Ajout d'une autre feature  $x_2 = x_1^2$  (2D) : Les données deviennent linéairement séparables.

Exemple: On considère le dataset moons (deux croissants entrelacés), généré avec la fonction `make_moons` : créer un **pipeline** contenant un transformateur **PolynomialFeatures**, suivi d'un **StandardScaler** et d'un classificateur **LinearSVC**.

```
from sklearn.datasets import make_moons
from sklearn.preprocessing import PolynomialFeatures
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)
polynomial_svm_class = make_pipeline(
    PolynomialFeatures(degree=3),
    StandardScaler(),
    LinearSVC(C=10, max_iter=10_000, random_state=42) )
polynomial_svm_class.fit(X, y)
```



Source Aurélien Géron - Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

# Chap 7      Machines à vecteurs de support (SVM)

## III- Classification SVM Non linéaire

144

### Polynomial Kernel

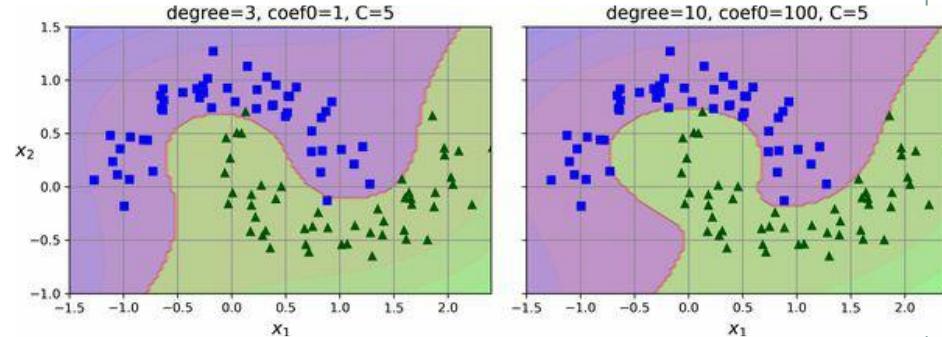
- L'ajout de caractéristiques polynomiales est simple et utile pour de nombreux algorithmes, mais :
  - A faible degré  $\Rightarrow$  ne gère pas les datasets complexes.
  - A haut degré  $\Rightarrow$  génère trop de caractéristiques et ralentit le modèle.
- Les SVM disposent de l'astuce du noyau qui permet d'obtenir les mêmes résultats qu'avec des polynômes de haut degré, sans créer physiquement toutes les caractéristiques.

$$K(x_i, x_j) = (x_i \cdot x_j + r)^d$$

Cette astuce est implémentée dans **Scikit-Learn** via la classe **SVC** avec `kernel="poly"`.

```
poly_kernel_svm_class = make_pipeline(StandardScaler(), SVC(kernel="poly", degree=3, coef0=1, C=5))
poly_kernel_svm_class.fit(X, y)
```

- Le **degré** du polynôme influence la complexité du modèle :
  - Trop élevé  $\Rightarrow$  risque de surapprentissage.
  - Trop faible  $\Rightarrow$  risque de sous-apprentissage.
- L'hyperparamètre **coef0** règle l'importance des termes de degré élevé par rapport aux termes de degré faible.



## III- Classification SVM Non linéaire

145

## Radial Basis Function (RBF) Kernel

$$K(x_i, x_j) = \exp(-\gamma ||x_i - x_j||^2)$$

```
from sklearn import svm
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
from sklearn.datasets import load_iris
import numpy as np
data = load_iris()
X = data.data      # caractéristiques (4 colonnes)
y = data.target    # classes (0, 1, 2)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
model = svm.SVC(kernel='rbf', C=1.0, gamma='scale') # Création et entraînement du modèle SVM
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred, target_names=data.target_names))
```

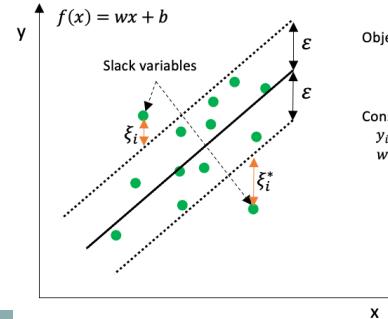
## IV- Régression SVM (SVR)

146

- L'objectif d'un **SVR** est de prédire une valeur continue en construisant une fonction la plus simple possible qui suit les données.
- On essaie de tracer une courbe qui suit des points, mais sans se laisser influencer par chaque petite fluctuation ou bruit dans les données :
  - On construit un "**tube**" autour de la courbe : Tant que les prédictions sont à moins de  $\varepsilon$  (un petit seuil) de la vraie valeur, on considère que c'est une bonne prédiction, et on n'en tient pas compte comme erreur.
  - Points importants : seuls les points en dehors du tube influencent réellement le modèle. Ce sont les Support Vectors.
  - Régularisation ( $C$ ) : On décide de combien on punit les points qui sortent du tube.
    - $C$  grand  $\rightarrow$  on veut que le tube contienne presque tout  $\rightarrow$  modèle plus flexible, risque d'overfit.
    - $C$  petit  $\rightarrow$  on accepte plus de sorties du tube  $\rightarrow$  modèle plus simple.
- Noyaux** : permettent de capturer des relations non linéaires (RBF, polynomial, etc.).

- Points clés :**

- modèle robuste au bruit.
- solution parcimonieuse (peu de points influencent la courbe).
- nécessite un bon réglage des hyperparamètres ( $C$ ,  $\varepsilon$ , gamma).



Objective:  

$$\text{Minimize: } \frac{1}{2} \|w\|^2 + C \sum_{i=1}^l (\xi_i + \xi_i^*)$$
  
 Constraints:  

$$\begin{aligned} y_i - wx_i - b &\leq \varepsilon + \xi_i \\ wx_i + b - y_i &\leq \varepsilon + \xi_i^* \\ \xi_i, \xi_i^* &\geq 0 \end{aligned}$$

Source: <https://medium.com/@niousha.rf/support-vector-regressor-theory-and-coding-exercise-in-python-ca6a7dfda927>

## IV- Régression SVM (SVR)

147

Exemple : (SVR)

```
import numpy as np
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error, r2_score

data = fetch_california_housing()
X = data.data
y = data.target
scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y.reshape(-1, 1)).ravel()
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled, test_size=0.2, random_state=42)
svr = SVR(kernel='rbf', C=10, epsilon=0.1, gamma='scale')
svr.fit(X_train, y_train)
y_pred_scaled = svr.predict(X_test)
y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-1, 1)) # Le SVR prédit des valeurs standardisées (c'est-à-
# dire centrées réduites). Pour pouvoir les comparer aux vraies valeurs (qui sont en prix réels dans le dataset), il faut les dénormaliser.

rmse = np.sqrt(mean_squared_error(y_test, y_pred))
r2 = r2_score(y_test, y_pred)

print("RMSE :", rmse)
print("R2 score :", r2)
```

# Chap 7 Machines à vecteurs de support (SVM)

## V- Domaines d'application de SVM

148

- i. **Classification d'images** : reconnaissance d'objets, d'animaux et de scènes.
- ii. **Reconnaissance de l'écriture manuscrite**.
- iii. **Détection d'anomalies/valeurs aberrantes** : détection des fraudes, détection des intrusions réseau, contrôle qualité, etc.
- iv. **Détection et reconnaissance faciale et gestuelle**.
- v. **Classification de texte et analyse des sentiments** : détection de spam, catégorisation des sujets, analyse des sentiments (positif ou négatif), etc.
- vi. **Bioinformatique** : prédiction de la structure des protéines, classification des gènes et le diagnostic des maladies à partir de données d'expression génique.
- vii. **Télédétection et analyse d'images satellites** : classifier les types de couverture terrestre.

### Avantages :

- Efficace dans les espaces de grande dimension.
- Mémoire efficace (seuls les vecteurs de support sont importants).
- Polyvalent (différents noyaux disponibles).
- Bonne performance théorique garantie.

### Inconvénients :

- Non adapté aux grands jeux de données (seulement pour datasets de quelques centaines à quelques milliers de données).
- Sensible au bruit.