**Module 4 APPRENTISAGE AUTOMATIQUE**

**Chapitre 1 Introduction au Machine Learning**

À la fin de ce chapitre, vous serez capable de :

* Comprendre ce qu’est le Machine Learning (ML)
* Différencier les types d’apprentissage (supervisé, non supervisé, par renforcement)
* Identifier les cas d’usage typiques du ML
* Installer et configurer votre environnement de travail

**1. Qu’est-ce que le Machine Learning ?**

Le **Machine Learning** est un sous-domaine de l’intelligence artificielle (IA) qui consiste à apprendre automatiquement à partir de données pour **faire des prédictions ou des décisions** sans être explicitement programmé.

**Exemple :**

* **Tâche** : prédire le prix d’une maison
* **Expérience** : base de données de ventes précédentes
* **Performance** : évaluer avec l’erreur quadratique moyenne (MSE)

**2. Les types d’apprentissage automatique**

**2.1 Apprentissage supervisé (Supervised Learning)**

* L’algorithme apprend à partir d’un **jeu de données étiqueté** (avec réponses).
* But : prédire une **valeur** ou une **classe**.

**Exemples :**

| **Application** | **Entrée (X)** | **Sortie (Y)** |
| --- | --- | --- |
| Prédire un salaire | Âge, diplôme | Salaire |
| Filtrer les spams | Email | Spam / Non spam |

Algorithmes : régression linéaire, kNN, SVM, Random Forest

**2.2 Apprentissage non supervisé (Unsupervised Learning)**

* L’algorithme **n’a pas de labels** : il découvre des structures cachées.

**Exemples :**

| **Application** | **Données** | **Résultat** |
| --- | --- | --- |
| Regrouper des clients | Achats | Groupes (clusters) |
| Réduction de dimensions | Images | Projection simplifiée |

Algorithmes : K-Means, PCA, DBSCAN

**2.3 Apprentissage par renforcement (Reinforcement Learning)**

* L’algorithme apprend **par essais/erreurs** via **récompense** ou **punition**.
* Utilisé pour des systèmes de décision séquentielle (jeu, robot, trading).

**Exemples :**

| **Agent** | **Environnement** | **Objectif** |
| --- | --- | --- |
| Voiture autonome | Route | Éviter les obstacles |
| Joueur d’échecs | Plateau | Gagner la partie |

**3. Cas d’usage du Machine Learning**

| **Domaine** | **Application** |
| --- | --- |
| Finance | Détection de fraude bancaire |
| Santé | Diagnostic médical automatisé |
| Marketing | Recommandation de produits |
| Industrie | Maintenance prédictive |
| Informatique | Reconnaissance vocale, traduction automatique |

**4. Mise en place de l’environnement**

**Outils nécessaires :**

* **Langage** : Python (le plus utilisé en ML)
* **IDE** : Jupyter Notebook / VSCode / Google Colab
* **Librairies** :
  + NumPy (calculs numériques)
  + Pandas (manipulation de données)
  + Matplotlib / Seaborn (visualisation)
  + Scikit-learn (algorithmes ML)

**Mini-activité pratique**

Voici un exemple simple pour illustrer la logique de prédiction :

# Exemple de régression linéaire avec scikit-learn

import numpy as np # Importation de la bibliothèque NumPy pour la gestion des tableaux numériques

import matplotlib.pyplot as plt # Importation de matplotlib pour tracer les graphiques

from sklearn.linear\_model import LinearRegression # Importation du modèle de régression linéaire depuis scikit-learn

# Données simples (ex : nombre d'heures étudiées vs. score obtenu)

X = np.array([[1], [2], [3], [4], [5]]) # Variable indépendante (entrée), sous forme de matrice (5 lignes, 1 colonne)

y = np.array([2, 4, 5, 4, 5]) # Variable dépendante (sortie), vecteur des scores correspondants

# Création du modèle de régression linéaire

model = LinearRegression() # Instanciation du modèle

model.fit(X, y) # Entraînement du modèle sur les données : calcul de la droite qui approxime au mieux les points

# Prédiction

y\_pred = model.predict(X) # Prédiction des sorties (scores) à partir des valeurs de X après entraînement

# Affichage des résultats

plt.scatter(X, y, color='blue', label="Données") # Affiche les données d'origine (points bleus)

plt.plot(X, y\_pred, color='red', label="Prédiction") # Affiche la droite de régression (prédictions) en rouge

plt.xlabel("Heures d'étude") # Étiquette de l'axe X

plt.ylabel("Score") # Étiquette de l'axe Y

plt.legend() # Affiche la légende pour identifier les courbes

plt.title("Régression Linéaire") # Titre du graphique

plt.show() # Affiche le graphique

* Le ML apprend à partir des données pour automatiser des décisions ou prédictions.
* Il existe 3 grandes familles : supervisé, non supervisé, par renforcement.
* Python avec Scikit-learn est l’outil de base le plus accessible pour débuter.

**Chapitre 2 Régression Supervisée**

Ce chapitre vous permet de :

* Comprendre la régression linéaire et ses variantes
* Appliquer un modèle de régression sur des données réelles
* Évaluer la performance d’un modèle de régression
* Éviter le surapprentissage grâce à la régularisation

**1. Qu’est-ce que la régression ?**

La **régression** est une technique de Machine Learning supervisé permettant de prédire une **valeur continue** à partir de variables explicatives.

**Exemples concrets :**

| **Problème** | **Variables d'entrée (X)** | **Variable cible (Y)** |
| --- | --- | --- |
| Prédire le prix d'une maison | Surface, nb de chambres | Prix |
| Estimer la consommation d'énergie | Température, jour, heure | kWh |

**2. Régression Linéaire Simple**

**Formule :**

* = variable d'entrée
* = coefficient (pente)
* = biais (ordonnée à l'origine)

**Étapes :**

1. Entraîner le modèle (fit)
2. Prédire des valeurs (predict)
3. Évaluer la performance (score, mean\_squared\_error, etc.)

**Exemple en Python :**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error

# Données

X = np.array([[1], [2], [3], [4], [5]]) # Variable indépendante (entrée), sous forme de tableau 2D

y = np.array([1.8, 3.9, 5.8, 8.2, 10.1]) # Variable dépendante (valeurs observées)

# Modèle

model = LinearRegression() # Création d'un modèle de régression linéaire

model.fit(X, y) # Entraînement du modèle sur les données X (entrée) et y (sortie réelle)

# Prédiction

y\_pred = model.predict(X) # Prédiction des valeurs de y à partir de X après entraînement

# Évaluation

mse = mean\_squared\_error(y, y\_pred) # Calcul de l'erreur quadratique moyenne (MSE) entre les vraies valeurs et les prédictions

print(f"Erreur quadratique moyenne : {mse:.2f}") # Affichage de l'erreur quadratique moyenne, arrondie à deux décimales

# Affichage

plt.scatter(X, y, color='blue', label="Données") # Affiche les points réels en bleu

plt.plot(X, y\_pred, color='red', label="Régression") # Affiche la droite de régression en rouge

plt.xlabel("X")

plt.ylabel("y")

plt.title("Régression linéaire simple")

plt.legend() # Affiche la légende

plt.show() # Affiche le graphique

**3. Régression Linéaire Multiple**

Lorsque vous avez **plusieurs variables explicatives** :

**Exemple :**

import pandas as pd

from sklearn.linear\_model import LinearRegression

# Données

data = {

'surface': [50, 60, 80, 100, 120], # Liste des surfaces en m²

'nb\_chambres': [1, 2, 2, 3, 3], # Liste du nombre de chambres

'prix': [150, 180, 220, 280, 310] # Liste des prix correspondants (ex : en milliers d'euros)

}

df = pd.DataFrame(data)

# X = variables explicatives (features), y = variable cible (target)

X = df[['surface', 'nb\_chambres']] # Sélection des colonnes "surface" et "nb\_chambres" comme variables d'entrée

y = df['prix'] # Sélection de la colonne "prix" comme variable à prédire

# Modèle

model = LinearRegression() # Création d'un objet modèle de régression linéaire

model.fit(X, y) # Entraînement du modèle sur les données d’entrée X et les sorties y

# Affichage des résultats du modèle

print("Coefficients :", model.coef\_) # Affiche les coefficients (pentes) du modèle pour chaque variable explicative

print("Intercept :", model.intercept\_) # Affiche l'ordonnée à l'origine (valeur de y quand toutes les variables explicatives = 0)

**4. Évaluation d’un modèle de régression**

**Métriques courantes :**

| **Métrique** | **Description** |
| --- | --- |
| MAE (Mean Absolute Error) | Moyenne des erreurs absolues |
| MSE (Mean Squared Error) | Moyenne des erreurs au carré |
| RMSE (Root Mean Squared Error) | Racine du MSE |
| R² (Score de détermination) | Proportion de variance expliquée |

from sklearn.metrics import mean\_absolute\_error, r2\_score # Import des métriques d’évaluation : MAE et R²

mae = mean\_absolute\_error(y, y\_pred) # Calcul de l'erreur absolue moyenne (MAE) entre les valeurs réelles et les prédictions

r2 = r2\_score(y, y\_pred) # Calcul du coefficient de détermination R² pour mesurer la qualité de la prédiction

print("MAE:", mae) # Affiche la MAE : moyenne des erreurs absolues (plus elle est petite, mieux c’est)

print("R²:", r2) # Affiche le score R² : proportion de la variance de y expliquée par le modèle (proche de 1 = bon modèle)

**5. Régularisation : Ridge & Lasso**

Lorsque le modèle s’ajuste **trop bien aux données** (overfitting), on utilise la **régularisation**. (Améliore la généralisation et éviter la surapprentissage)

**Ridge (L2) :**

* Pénalise les **grands coefficients**
* Ajoute une **contrainte** à la somme des carrés des poids

**Lasso (L1) :**

* Peut **réduire certains coefficients à 0**
* Favorise la **sélection de variables**

from sklearn.linear\_model import Ridge, Lasso # Importation des modèles de régression Ridge et Lasso depuis scikit-learn

ridge = Ridge(alpha=1.0) # Création d'un modèle Ridge avec un paramètre de régularisation alpha = 1.0

ridge.fit(X, y) # Entraînement du modèle Ridge sur les données X (features) et y (target)

lasso = Lasso(alpha=0.1) # Création d'un modèle Lasso avec un paramètre de régularisation alpha = 0.1

lasso.fit(X, y) # Entraînement du modèle Lasso sur les mêmes données

print("Ridge coefficients :", ridge.coef\_) # Affiche les coefficients (poids des variables) obtenus par le modèle Ridge

print("Lasso coefficients :", lasso.coef\_) # Affiche les coefficients obtenus par le modèle Lasso

#### Ridge Regression :

* Ajoute une **pénalité L2** (la somme des carrés des coefficients).
* Tendance à **réduire** les coefficients mais **sans les annuler**.
* Utile quand les variables sont **corrélées** ou pour **éviter l’overfitting**.

#### Lasso Regression :

* Ajoute une **pénalité L1** (la somme des valeurs absolues des coefficients).
* Tendance à **réduire certains coefficients à zéro** → **sélection automatique des variables**.
* Pratique pour **simplifier les modèles** et détecter les variables les plus importantes.

**Exercice pratique**

Prédire le prix d’un bien immobilier à partir des caractéristiques suivantes :

* Surface (m²)
* Nombre de chambres
* Année de construction

**Étapes proposées :**

1. Créer un DataFrame avec 10 biens fictifs
2. Appliquer une régression linéaire multiple
3. Évaluer la performance du modèle (MAE, MSE, R²)
4. Tester Ridge et Lasso

**À retenir**

* La régression linéaire permet de modéliser une relation entre variables.
* L’évaluation se fait avec MSE, MAE, RMSE et R².
* Ridge et Lasso permettent d’éviter le surapprentissage.
* Scikit-learn fournit tous les outils pour implémenter ces modèles.

**Chapitre 3 Classification Supervisée**

Ce chapitre vous permet de :

* Comprendre la classification supervisée et ses applications
* Maîtriser plusieurs algorithmes de classification
* Évaluer la performance des modèles de classification
* Implémenter des modèles simples avec Scikit-learn

**1. Qu’est-ce que la classification ?**

La classification est un type de machine learning **supervisé** où l’objectif est de **prédire une étiquette (catégorie)** à partir de variables d’entrée.

**Exemples :**

| **Cas d’usage** | **Entrées (features)** | **Sortie (classe)** |
| --- | --- | --- |
| Détection d'email spam | Texte de l'email | Spam / Non spam |
| Diagnostic médical | Température, tension, symptôme | Malade / Sain |
| Recrutement | Compétences, diplômes | Accepté / Refusé |

**2. Algorithmes de classification**

**2.1. Logistic Regression**

* Convient pour des classes **binaires** (0 ou 1)
* Utilise une fonction sigmoïde (un graphique pour visualiser sa forme)

from sklearn.linear\_model import LogisticRegression # Importation de la classe LogisticRegression pour la classification binaire ou multiclasse

model = LogisticRegression() # Création d'un modèle de régression logistique avec les paramètres par défaut

model.fit(X\_train, y\_train) # Entraînement du modèle sur les données d'apprentissage (X\_train = variables explicatives, y\_train = étiquettes/classes)

**2.2. k-Nearest Neighbors (k-NN)**

* Classifie selon les **k plus proches voisins**
* Pas d'entraînement, sensible à la distance

from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n\_neighbors=3)

**2.3. Decision Tree**

* Structure en arbre avec des règles conditionnelles
* Interprétable, mais sujet à l’**overfitting**

from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier()

**2.4. Random Forest**

* Ensemble de plusieurs arbres (bagging)
* Réduit l'overfitting, très efficace

from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()

**2.5. Support Vector Machine (SVM)**

* Sépare les classes avec une hyperplane optimale
* Efficace pour les petits datasets bien séparables

from sklearn.svm import SVC

model = SVC()

**2.6. Naive Bayes**

* Basé sur le théorème de Bayes
* Rapide et efficace pour les textes, spam, etc.

from sklearn.naive\_bayes import GaussianNB

model = GaussianNB()

**3. Évaluation d’un modèle de classification**

**Métriques :**

| **Nom** | **Utilité** |
| --- | --- |
| Accuracy | Proportion de bonnes prédictions |
| Precision | Précision sur les vrais positifs |
| Recall | Sensibilité / Rappel |
| F1-score | Moyenne harmonique précision/rappel |
| Matrice de confusion | Détail des vrais/faux positifs/négatifs |
| ROC & AUC | Mesure la performance globale du modèle binaire |

from sklearn.metrics import classification\_report, confusion\_matrix

print(confusion\_matrix(y\_test, y\_pred))

print(classification\_report(y\_test, y\_pred))

**4. Exemple pratique : Classification d’iris**

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import classification\_report

# Chargement du dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Séparation des données

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3)

# Modèle

model = RandomForestClassifier()

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

# Prédiction

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

# Évaluation

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

**5. Visualisation : Matrice de confusion**

La **matrice de confusion** (ou *confusion matrix*) est un outil fondamental pour évaluer les performances d’un modèle de **classification** supervisée. Elle permet de visualiser les prédictions du modèle par rapport aux valeurs réelles.

**Exemple**

**Contexte : Prédiction binaire (malade ou sain)**

Supposons que l’on a un modèle qui classe des patients en deux catégories :

* 1 = malade
* 0 = sain

**Étape 1 : Importer les bibliothèques**

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay

import matplotlib.pyplot as plt

**Étape 2 : Résultats simulés**

# Vérités terrain (valeurs réelles)

y\_true = [0, 1, 0, 1, 0, 1, 1, 0, 1, 0]

# Prédictions du modèle

y\_pred = [0, 1, 0, 0, 0, 1, 1, 1, 1, 0]

**Étape 3 : Calcul et affichage de la matrice de confusion**

# Création de la matrice de confusion

cm = confusion\_matrix(y\_true, y\_pred)

# Affichage graphique

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=["Sain", "Malade"])

disp.plot(cmap='Blues')

plt.title("Matrice de confusion")

plt.show()

**Interprétation de la matrice**

Supposons que la matrice de confusion affichée est la suivante :

Prédit

Sain Malade

Réel Sain 4 1

Malade 1 4

Voici comment lire cette matrice :

|  | **Prédit Sain** | **Prédit Malade** |
| --- | --- | --- |
| **Réel Sain** | **4 (TN)** | **1 (FP)** |
| **Réel Malade** | **1 (FN)** | **4 (TP)** |

**TP (True Positives)** : Malades correctement détectés (4)

* **TN (True Negatives)** : Personnes saines correctement détectées (4)
* **FP (False Positives)** : Faux malades, alors qu’ils sont sains (1)
* **FN (False Negatives)** : Malades non détectés (1)

## Métriques dérivées de la matrice

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

print("Accuracy :", accuracy\_score(y\_true, y\_pred))

print("Précision (malades détectés correctement) :", precision\_score(y\_true, y\_pred))

print("Rappel (sensibilité) :", recall\_score(y\_true, y\_pred))

print("F1-score :", f1\_score(y\_true, y\_pred))

**# Importation des fonctions d'évaluation à partir de la bibliothèque scikit-learn :**

**# - accuracy\_score : pour calculer la précision globale du modèle**

**# - precision\_score : pour calculer la précision (taux de vrais positifs parmi les positifs prédits)**

**# - recall\_score : pour calculer le rappel (sensibilité, taux de détection des vrais positifs)**

**# - f1\_score : pour calculer le F1-score, moyenne harmonique entre précision et rappel**

**from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score**

**# Calcul et affichage de la précision globale :**

**# C'est le pourcentage de bonnes prédictions (positives et négatives confondues).**

**print("Accuracy :", accuracy\_score(y\_true, y\_pred))**

**# Calcul et affichage de la précision :**

**# Elle indique parmi toutes les personnes que le modèle a prédites comme "malades", combien le sont réellement.**

**# Formule : Précision = TP / (TP + FP)**

**print("Précision (malades détectés correctement) :", precision\_score(y\_true, y\_pred))**

**# Calcul et affichage du rappel :**

**# Il indique parmi tous les vrais malades, combien ont été correctement prédits.**

**# Formule : Rappel = TP / (TP + FN)**

**print("Rappel (sensibilité) :", recall\_score(y\_true, y\_pred))**

**# Calcul et affichage du F1-score :**

**# C'est une moyenne harmonique entre la précision et le rappel.**

**# Il est utile quand il y a un déséquilibre entre les classes.**

**# Formule : F1 = 2 \* (précision \* rappel) / (précision + rappel)**

**print("F1-score :", f1\_score(y\_true, y\_pred))**

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.metrics import confusion\_matrix

cm = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=iris.target\_names, yticklabels=iris.target\_names)

plt.xlabel("Prédit")

plt.ylabel("Réel")

plt.title("Matrice de confusion")

plt.show()

**6. TP / Activité pratique**

Construire un modèle pour prédire si un individu est diabétique (jeu de données *Pima Indians Diabetes*).

**Étapes proposées :**

1. Charger le dataset depuis un fichier CSV
2. Nettoyer les données si nécessaire
3. Appliquer un modèle de classification (LogisticRegression ou RandomForest)
4. Évaluer avec précision, rappel, f1-score
5. Visualiser la matrice de confusion

**À retenir**

* La classification permet de prédire des **catégories** à partir de données.
* Il existe de nombreux modèles, chacun adapté à un contexte particulier.
* L’évaluation va au-delà de la simple accuracy : **précision**, **rappel**, **F1-score**, **confusion matrix** sont essentiels.
* Scikit-learn fournit tous les outils pour expérimenter rapidement.

**Chapitre 4 Apprentissage non supervisé**

À la fin de ce chapitre, vous serez capable de :

* Comprendre les principes de l’apprentissage non supervisé
* Utiliser les algorithmes de **clustering** (groupement)
* Réduire la **dimensionnalité** des données
* Visualiser des structures cachées dans un jeu de données
* Appliquer ces méthodes à des cas pratiques

**1. Qu’est-ce que l’apprentissage non supervisé ?**

C’est une catégorie de machine learning où **les données n’ont pas de labels**. L’objectif est de **découvrir des structures** cachées (groupes, similarités, patterns(**motif** ou désigne une **structure régulière**)).

**2. Applications typiques**

| **Domaine** | **Application** |
| --- | --- |
| Marketing | Segmentation de clients |
| Santé | Regroupement de patients par profil |
| Cybersécurité | Détection d’anomalies |
| E-commerce | Recommandation de produits |
| Réseaux sociaux | Regroupement de communautés |

**3. Algorithmes de Clustering**

**3.1. K-Means Clustering**

* Algorithme itératif qui regroupe les données en **K clusters**.
* Chaque point est affecté au **centre (centroïde)** le plus proche.
* Sensible à l’échelle des données et au choix de **K**.

from sklearn.cluster import KMeans

import matplotlib.pyplot as plt

from sklearn.datasets import make\_blobs

# Création de données simulées

X, \_ = make\_blobs(n\_samples=300, centers=3, cluster\_std=0.6)

# Modèle KMeans

kmeans = KMeans(n\_clusters=3)

kmeans.fit(X)

y\_kmeans = kmeans.predict(X)

# Visualisation

plt.scatter(X[:, 0], X[:, 1], c=y\_kmeans, cmap='viridis')

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], s=200, c='red')

plt.title("K-Means Clustering")

plt.show()

***Explication du code :***

make\_blobs est une fonction de sklearn.datasets qui génère des données synthétiques pour le clustering.

n\_samples=300 : on génère 300 points de données.

centers=3 : les points seront regroupés autour de 3 centres (c’est-à-dire 3 vrais groupes).

cluster\_std=0.6 : écart-type (dispersion) autour de chaque centre.

X contient les coordonnées des points générés (features 2D ici).

\_ signifie qu'on ignore les étiquettes réelles retournées par make\_blobs, car KMeans est non supervisé.

# Modèle KMeans

kmeans = KMeans(n\_clusters=3)

On crée un modèle de clustering K-Means.

n\_clusters=3 signifie que l’on souhaite regrouper les données en 3 clusters.

kmeans.fit(X)

Le modèle apprend les centres des clusters en analysant les points X.

Cela applique l’algorithme K-Means pour regrouper les points selon leur proximité.

y\_kmeans = kmeans.predict(X)

Chaque point est assigné à un cluster (0, 1 ou 2) selon le centre le plus proche.

y\_kmeans contient les étiquettes de clusters prédites pour chaque point.

# Visualisation

plt.scatter(X[:, 0], X[:, 1], c=y\_kmeans, cmap='viridis')

Trace un nuage de points à partir de X.

Les couleurs sont définies par les clusters (y\_kmeans).

cmap='viridis' applique une palette de couleurs.

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], s=200, c='red')

Affiche les centres de clusters calculés par le modèle en rouge.

s=200 augmente la taille des points.

kmeans.cluster\_centers\_ contient les coordonnées des centres trouvés.

plt.title("K-Means Clustering")

plt.show()

Titre du graphique et affichage.

***Résultat :***

Ce code montre comment le modèle K-Means regroupe automatiquement des données 2D en 3 clusters sans connaître les étiquettes réelles.

**3.2. Hiérarchical Clustering (Agglomératif)**

* Crée une **arborescence (dendrogramme)** de fusions de clusters.
* Ne nécessite pas toujours de spécifier K à l'avance.
* Peut être **plus coûteux** en temps pour de grands datasets.

from scipy.cluster.hierarchy import dendrogram, linkage

import matplotlib.pyplot as plt

linked = linkage(X, method='ward')

dendrogram(linked)

plt.title("Dendrogramme hiérarchique")

plt.xlabel("Points")

plt.ylabel("Distance")

plt.show()

**3.3. DBSCAN (Density-Based Spatial Clustering)**

* Détecte des **zones de forte densité**.
* Capable de trouver des **clusters de forme arbitraire**.
* Bon pour la **détection d’anomalies**.

from sklearn.cluster import DBSCAN

db = DBSCAN(eps=0.5, min\_samples=5)

y\_db = db.fit\_predict(X)

**4. Réduction de dimensionnalité**

**4.1. PCA (Principal Component Analysis)**

* Réduit les dimensions tout en **préservant la variance maximale**.
* Très utilisé pour **visualiser des données multivariées**.

from sklearn.decomposition import PCA

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(X)

plt.scatter(X\_pca[:, 0], X\_pca[:, 1])

plt.title("Projection PCA")

plt.xlabel("Composante 1")

plt.ylabel("Composante 2")

plt.show()

**4.2. t-SNE (t-Distributed Stochastic Neighbor Embedding)**

* Pour **visualiser des clusters** sur des données très complexes (images, texte…).
* Conserve la **proximité locale** des points.

from sklearn.manifold import TSNE

tsne = TSNE(n\_components=2, perplexity=30)

X\_tsne = tsne.fit\_transform(X)

plt.scatter(X\_tsne[:, 0], X\_tsne[:, 1])

plt.title("Projection t-SNE")

plt.show()

**5. Comment évaluer un clustering ?**

Contrairement au ML supervisé, ici **on n’a pas de “vraie réponse”**.

**Métriques internes :**

* **Inertia (pour K-Means)** : somme des distances intra-cluster
* **Silhouette Score** : mesure la cohésion et séparation des clusters

from sklearn.metrics import silhouette\_score

score = silhouette\_score(X, y\_kmeans)

print("Silhouette score :", score)

**6. TP : Segmentation de clients**

Appliquer le **clustering K-Means** pour segmenter des clients selon leur comportement d’achat.

**Étapes proposées :**

1. Charger un dataset (ex : Mall\_Customers.csv)
2. Prétraiter (normalisation)
3. Utiliser KMeans avec plusieurs valeurs de K
4. Afficher les clusters
5. Évaluer avec silhouette score

**À retenir**

* L’apprentissage non supervisé explore les données sans “réponses”.
* K-Means est l’algorithme de clustering le plus courant.
* PCA et t-SNE aident à **visualiser des données complexes**.
* Les résultats doivent être **interprétés** avec prudence.

**Chapitre 5 Réseaux de Neurones & Deep Learning (Introduction)**

Ce chapitre vous permet de :

* Comprendre le fonctionnement d’un réseau de neurones artificiels
* Connaître les concepts de base du Deep Learning
* Créer un réseau simple avec TensorFlow/Keras
* Appliquer un modèle de classification sur un dataset comme **MNIST**

**1. Qu’est-ce qu’un réseau de neurones artificiels (ANN) ?**

Un **réseau de neurones** est un ensemble de **neurones artificiels** interconnectés, inspirés du cerveau humain. Il transforme une **entrée** (vecteur) en **sortie** via des **fonctions d’activation** et **ajustement de poids**.

**Structure d’un réseau de neurones**

| **Type de couche** | **Rôle** |
| --- | --- |
| **Input Layer** | Reçoit les données |
| **Hidden Layers** | Traitent les données avec des neurones |
| **Output Layer** | Donne la sortie finale (classe, valeur…) |

**Exemple :**

Un réseau pour reconnaître des chiffres manuscrits :

* Entrée : image 28x28 pixels = 784 pixels
* Sortie : chiffre de 0 à 9 (10 neurones)

**2. Fonctionnement d’un réseau**

**Étapes :**

1. **Propagation avant (Forward Propagation)** : Calcul de la sortie
2. **Calcul de l’erreur** : Différence entre sortie attendue et prédite
3. **Rétropropagation (Backpropagation)** : Mise à jour des poids
4. **Répétition (Epochs)** : Plusieurs passages pour converger

**3. Fonctions d’activation**

| **Fonction** | **Utilité** |
| --- | --- |
| **ReLU** | Rapide, utilisée dans les couches cachées |
| **Sigmoïde** | Pour les sorties binaires |
| **Softmax** | Pour des sorties multi-classes |

**4. Librairies du Deep Learning**

| **Librairie** | **Description** |
| --- | --- |
| **TensorFlow** | Framework de Google pour l’apprentissage automatique |
| **Keras** | Interface haut-niveau de TensorFlow pour créer des modèles rapidement |

**5. Exemple pratique : Classification de chiffres manuscrits (MNIST)**

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.utils import to\_categorical

# Chargement des données

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# Prétraitement

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

y\_train = to\_categorical(y\_train)

y\_test = to\_categorical(y\_test)

# Modèle

model = Sequential([

Flatten(input\_shape=(28, 28)),

Dense(128, activation='relu'),

Dense(64, activation='relu'),

Dense(10, activation='softmax')

])

# Compilation

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Entraînement

model.fit(X\_train, y\_train, epochs=5, batch\_size=32, validation\_split=0.2)

# Évaluation

loss, accuracy = model.evaluate(X\_test, y\_test)

print(f"Test Accuracy: {accuracy\*100:.2f}%")

**6. Visualisation de prédictions**

import matplotlib.pyplot as plt

import numpy as np

predictions = model.predict(X\_test)

plt.imshow(X\_test[0], cmap='gray')

plt.title(f"Prédit : {np.argmax(predictions[0])}")

plt.show()

**7. Concepts importants**

| **Terme** | **Signification** |
| --- | --- |
| **Epoch** | Une passe complète sur le jeu de données |
| **Batch Size** | Nombre d’échantillons traités avant mise à jour des poids |
| **Loss function** | Fonction à minimiser (ex: cross-entropy) |
| **Optimizer** | Algorithme d’optimisation (SGD, Adam, etc.) |

**8. Limitations et précautions**

* Les réseaux de neurones nécessitent **beaucoup de données**
* Attention à l’**overfitting**
* Importance de la **normalisation** des données
* Les **hyperparamètres** (nombre de neurones, learning rate, etc.) influencent les performances

**9. TP : Reconnaissance de chiffres manuscrits**

1. Charger les données MNIST (ou Fashion-MNIST)
2. Construire un réseau avec au moins 2 couches cachées
3. Tester ReLU et Sigmoïde comme activation
4. Afficher les erreurs et la courbe de validation

**À retenir**

* Un réseau de neurones est une suite de couches transformant l’entrée vers une sortie.
* Les fonctions d’activation comme ReLU et Softmax sont essentielles.
* Keras (via TensorFlow) permet de construire des modèles très facilement.
* Le dataset MNIST est un bon point de départ.

**Chapitre 6 Outils avancés et techniques pratiques**

**Objectifs :**

* Créer des **pipelines de traitement** avec Scikit-learn
* Sauvegarder et recharger vos modèles
* Utiliser des outils pour automatiser l'entraînement
* Intégrer le **traitement de texte** (NLP de base)
* Manipuler des ensembles de données complexes

**1. Pipelines avec Scikit-learn**

Un **pipeline** est une chaîne de traitements : nettoyage, encodage, normalisation, modélisation…

**Avantages :**

* Automatisation
* Reproductibilité
* Prévention des fuites de données (**data leakage**)

**Exemple :**

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.linear\_model import LogisticRegression

pipeline = Pipeline([

('scaler', StandardScaler()),

('classifier', LogisticRegression())

])

pipeline.fit(X\_train, y\_train)

y\_pred = pipeline.predict(X\_test)

**2. Sauvegarde et chargement de modèles**

Utile pour la mise en production ou l’inférence ultérieure.

**Avec joblib :**

import joblib

# Sauvegarde

joblib.dump(model, 'modele\_sauvegarde.pkl')

# Chargement

model\_reloaded = joblib.load('modele\_sauvegarde.pkl')

**3. Traitement de texte (NLP de base)**

Le NLP (Natural Language Processing) est important pour analyser du **texte** (emails, tweets, documents...).

**Étapes typiques :**

* Nettoyage (suppression ponctuations, minuscules)
* Tokenisation (découper en mots)
* Stop words removal
* Transformation : Bag of Words, TF-IDF

**Exemple TF-IDF + Naive Bayes :**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.pipeline import make\_pipeline

textes = ["ce produit est génial", "je déteste ce produit", "super service", "nul"]

labels = [1, 0, 1, 0]

model = make\_pipeline(TfidfVectorizer(), MultinomialNB())

model.fit(textes, labels)

print(model.predict(["j'adore ce service"])) # --> [1]

**4. Encodage de variables catégorielles**

**4.1. Label Encoding**

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

y\_encoded = le.fit\_transform(["chat", "chien", "chat", "oiseau"])

**4.2. One-Hot Encoding**

import pandas as pd

df = pd.DataFrame({'animal': ['chat', 'chien', 'chat', 'oiseau']})

df\_encoded = pd.get\_dummies(df)

**5. Cross-validation & Grid Search**

**Cross-validation (validation croisée)**

Permet de vérifier la **robustesse** d’un modèle.

from sklearn.model\_selection import cross\_val\_score

score = cross\_val\_score(model, X, y, cv=5).mean()

**Grid Search : optimiser les hyperparamètres**

from sklearn.model\_selection import GridSearchCV

params = {'n\_neighbors': [3, 5, 7]}

grid = GridSearchCV(KNeighborsClassifier(), param\_grid=params, cv=5)

grid.fit(X\_train, y\_train)

print(grid.best\_params\_)

**6. Gérer les classes déséquilibrées**

Lorsque certaines classes sont surreprésentées (ex : 90% vs 10%), la précision peut être trompeuse.

**Solutions :**

* **Oversampling** (SMOTE)
* **Undersampling**
* **Balanced accuracy / weighted F1-score**

from imblearn.over\_sampling import SMOTE

sm = SMOTE()

X\_resampled, y\_resampled = sm.fit\_resample(X\_train, y\_train)

**7. Chargement de jeux de données réels**

**7.1. Depuis sklearn.datasets**

from sklearn.datasets import load\_wine

data = load\_wine()

X = data.data

y = data.target

**7.2. Depuis un fichier CSV**

import pandas as pd

df = pd.read\_csv('fichier.csv')

**8. Bonnes pratiques pour vos projets ML**

* Créer des **fonctions réutilisables**
* Organiser les étapes en train.py, utils.py, config.json
* Ajouter des **logs** et des **graphiques d'apprentissage**
* Séparer clairement entraînement / validation / test
* Évaluer le **coût de l’erreur** selon le contexte métier

**9. TP proposé**

Créer un pipeline complet pour :

* Charger un dataset texte (ex. : critiques de films IMDB)
* Appliquer TF-IDF + classification (Naive Bayes)
* Évaluer la performance
* Sauvegarder le modèle
* Recharger pour tester une prédiction

**À retenir**

* Les **pipelines** permettent d’automatiser les chaînes de traitement.
* joblib sert à **sauvegarder les modèles** entraînés.
* Le **traitement de texte** nécessite des vecteurs (TF-IDF).
* Le **choix des bons hyperparamètres** est crucial (GridSearch, CV).
* Des classes déséquilibrées nécessitent des **techniques de rééquilibrage**.

**Chapitre 7 Validation de modèles et interprétabilité**

**Objectifs :**

* Évaluer correctement les performances d’un modèle
* Choisir la bonne **métrique** selon le type de problème
* Utiliser la **validation croisée**
* Éviter les biais d’évaluation (overfitting, leakage)
* Comprendre les prédictions d’un modèle (interprétabilité locale et globale)

**1. Métriques d’évaluation selon le type de problème**

**1.1 Pour la régression :**

* MSE (Mean Squared Error)
* RMSE (Racine de MSE)
* MAE (Mean Absolute Error)
* R² (Score de détermination)

from sklearn.metrics import mean\_squared\_error, r2\_score

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

**1.2 Pour la classification :**

* Accuracy
* Precision, Recall
* F1-score
* ROC AUC, Matrice de confusion

from sklearn.metrics import classification\_report, confusion\_matrix

print(classification\_report(y\_test, y\_pred))

print(confusion\_matrix(y\_test, y\_pred))

**2. Validation croisée (Cross-validation)**

Technique robuste d’évaluation pour éviter l’**overfitting**.

**Exemple :**

python

CopierModifier

from sklearn.model\_selection import cross\_val\_score

scores = cross\_val\_score(model, X, y, cv=5)

print("Moyenne score CV :", scores.mean())

**3. Techniques d’échantillonnage**

* **Hold-out** (train/test)
* **K-Fold** (validation croisée classique)
* **Stratified K-Fold** (préserve les proportions de classes)
* **Leave-One-Out (LOO)**

**4. Biais fréquents dans l’évaluation**

* **Overfitting** : sur-apprentissage sur les données d'entraînement
* **Data leakage** : information future utilisée par erreur
* **Imbalanced classes** : affecte la précision

**Solution : évaluer avec les bonnes métriques et split bien fait.**

**5. Interprétabilité des modèles**

**5.1 Interprétabilité globale**

Comprendre comment **le modèle** prend ses décisions globalement.

* Feature Importance (importance des variables)
* Coefficients des modèles linéaires
* permutation importance

from sklearn.inspection import permutation\_importance

result = permutation\_importance(model, X\_test, y\_test)

**5.2 Interprétabilité locale**

Comprendre **une prédiction individuelle**.

**Outils :**

* **SHAP (SHapley Additive exPlanations)**
* **LIME (Local Interpretable Model-agnostic Explanations)**

import shap

explainer = shap.Explainer(model, X\_train)

shap\_values = explainer(X\_test)

shap.plots.waterfall(shap\_values[0])

Ces méthodes expliquent **quelles variables** ont le plus influencé la prédiction d’un cas particulier.

**6. Visualisation de la performance**

* Courbe ROC / AUC
* Matrice de confusion (heatmap)
* Courbe de précision/rappel
* Learning curves

from sklearn.metrics import RocCurveDisplay

RocCurveDisplay.from\_estimator(model, X\_test, y\_test)

**7. TP proposé**

**Sujet : Évaluer un modèle de classification binaire sur des données médicales**

Objectif : Prédire une maladie (ex : diabète)

* Séparer données (train/test stratifié)
* Entraîner un modèle (Random Forest)
* Évaluer via F1-score + AUC
* Afficher la matrice de confusion
* Interpréter les prédictions avec SHAP

**À retenir**

* L’évaluation dépend du **contexte métier** (précision vs rappel…).
* Utilisez la **validation croisée** pour fiabiliser vos scores.
* Apprenez à interpréter les modèles pour éviter la boîte noire.
* L’interprétabilité est un **facteur clé d’adoption en entreprise**.

**Chapitre 7 Introduction à l’Apprentissage par Renforcement**

**Objectifs :**

* Comprendre les concepts fondamentaux de l’apprentissage par renforcement
* Décrire l’interaction entre un **agent**, un **environnement** et des **récompenses**
* Différencier les types d’algorithmes RL (Q-learning, SARSA, Policy Gradient)
* Appliquer un algorithme simple sur un environnement type gym

**1. Qu’est-ce que l’Apprentissage par Renforcement ?**

L’apprentissage par renforcement est une **méthode d’apprentissage basée sur des interactions**. Un agent apprend en recevant des **récompenses** ou **punitions** après avoir pris des décisions.

**But : Maximiser la récompense cumulée sur le long terme**

**2. Les composants du RL**

| **Élément** | **Description** |
| --- | --- |
| **Agent** | Celui qui agit |
| **Environnement** | Le monde avec lequel l’agent interagit |
| **État (state)** | Situation actuelle |
| **Action** | Décision de l’agent |
| **Récompense** | Feedback (valeur numérique) |
| **Politique (π)** | Stratégie de l’agent |
| **Fonction de valeur (V, Q)** | Estimation du bénéfice futur |

**3. Processus de Décision de Markov (MDP)**

Le RL repose sur un **MDP** :

* Ensemble d’états S
* Ensemble d’actions A
* Fonction de transition T(s, a, s')
* Fonction de récompense R(s, a)
* Objectif : apprendre une **politique π(s)** optimale

**4. Paradigme de l’apprentissage**

**4.1 Exploration vs Exploitation**

* **Exploitation** : choisir l’action qui donne le meilleur gain connu
* **Exploration** : essayer de nouvelles actions pour découvrir de meilleures stratégies

**4.2 Politique :**

* **Déterministe** : π(s) = a
* **Stochastique** : π(a|s) = probabilité de choisir a dans s

**5. Algorithmes classiques**

**5.1 Q-Learning (off-policy)**

Q(s,a) ← Q(s,a) + α[r + γ \* max Q(s’,a’) - Q(s,a)]

**5.2 SARSA (on-policy)**

Q(s,a) ← Q(s,a) + α[r + γ \* Q(s’,a’) - Q(s,a)]

**5.3 Deep Q-Network (DQN)**

* Remplace la table Q par un réseau de neurones (utilisé chez DeepMind avec Atari)

**6. Implémentation simple avec OpenAI Gym**

**Exemple avec Q-learning sur FrozenLake**

import gym

import numpy as np

env = gym.make("FrozenLake-v1", is\_slippery=False)

Q = np.zeros((env.observation\_space.n, env.action\_space.n))

alpha = 0.8

gamma = 0.95

episodes = 1000

for episode in range(episodes):

state = env.reset()[0]

done = False

while not done:

action = np.argmax(Q[state] + np.random.randn(1, env.action\_space.n)[0])

new\_state, reward, done, \_, \_ = env.step(action)

Q[state, action] += alpha \* (reward + gamma \* np.max(Q[new\_state]) - Q[state, action])

state = new\_state

**7. Notions avancées (pour aller plus loin)**

* **Policy Gradient Methods** (REINFORCE, PPO)
* **Actor-Critic**
* **Multi-Armed Bandits**
* **Model-Based RL**
* **Enseignement par imitation (Imitation learning)**

**8. Applications du RL**

| **Domaine** | **Exemple** |
| --- | --- |
| Jeux vidéo | AlphaGo, Dota2 (OpenAI Five), Atari |
| Robotique | Contrôle de bras robotisé |
| Finance | Optimisation de portefeuilles |
| Automobile | Véhicules autonomes |
| Logistique | Optimisation de trajets, entrepôts |

**9. TP proposé**

**Sujet : Q-Learning dans FrozenLake ou Taxi-v3**

* Implémentez un Q-Learning
* Affichez la **politique apprise**
* Évaluez la performance (taux de réussite)
* Visualisez la progression des scores

Extensions possibles : essayer **DQN avec PyTorch** ou **stable-baselines3**

**À retenir**

* Le RL apprend par **essais et erreurs** dans un environnement inconnu
* Q-learning et SARSA sont des algorithmes fondamentaux
* Le **trade-off exploration/exploitation** est au cœur du RL
* Le RL est puissant mais **complexe à stabiliser**

**Chapitre 8 Déploiement de modèles de Machine Learning**

**Objectifs :**

* Expliquer les enjeux du déploiement de modèles ML en production
* Préparer un modèle pour l’inférence (sérialisation, dépendances)
* Créer une API pour exposer un modèle ML (avec Flask, FastAPI)
* Conteneuriser l’application avec Docker
* Déployer sur un cloud (Heroku, AWS, etc.)

**1. Cycle de vie d’un projet ML**

| **Étape** | **Description** |
| --- | --- |
| Prétraitement | Nettoyage, encodage, scaling |
| Entraînement | Séparation train/test, entraînement |
| Évaluation | Validation croisée, métriques |
| **Déploiement** | Mise à disposition du modèle |
| Monitoring | Suivi des performances en production |

**2. Sérialisation du modèle**

**2.1 Outils courants**

* pickle / joblib : pour les modèles scikit-learn
* torch.save() : pour PyTorch
* model.save() : pour TensorFlow/Keras

**Exemple (scikit-learn)**

import joblib

joblib.dump(model, 'modele\_rf.joblib')

# Chargement

model = joblib.load('modele\_rf.joblib')

**3. Créer une API avec Flask ou FastAPI**

**3.1 Exemple avec Flask**

from flask import Flask, request, jsonify

import joblib

app = Flask(\_\_name\_\_)

model = joblib.load("modele\_rf.joblib")

@app.route("/predict", methods=["POST"])

def predict():

data = request.json

features = [data['age'], data['taille']]

prediction = model.predict([features])

return jsonify({'prediction': int(prediction[0])})

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

**3.2 Exemple avec FastAPI (plus moderne)**

from fastapi import FastAPI

from pydantic import BaseModel

import joblib

app = FastAPI()

model = joblib.load("modele\_rf.joblib")

class InputData(BaseModel):

age: float

taille: float

@app.post("/predict")

def predict(data: InputData):

features = [[data.age, data.taille]]

prediction = model.predict(features)

return {"prediction": int(prediction[0])}

**4. Conteneurisation avec Docker**

**4.1 Dockerfile**

FROM python:3.10

WORKDIR /app

COPY . .

RUN pip install -r requirements.txt

CMD ["python", "app.py"]

**4.2 Commandes**

docker build -t ml-api .

docker run -p 5000:5000 ml-api

**5. Déploiement sur le cloud**

**5.1 Heroku (gratuit et simple)**

* Créez un fichier Procfile :

web: python app.py

* Déploiement :

heroku login

heroku create

git push heroku main

**5.2 Alternatives cloud**

| **Plateforme** | **Avantage** |
| --- | --- |
| **Heroku** | Facile pour débuter |
| **AWS EC2 / Sagemaker** | Puissant mais complexe |
| **Render** | Gratuit comme Heroku |
| **Google Cloud Run** | Serveurless, performant |
| **Azure ML** | Intégration entreprise |

**6. Monitoring et mise à jour**

* **Logs & erreurs** : via logging, Sentry ou Prometheus
* **Retraining** : pipeline pour réentraînement automatique
* **Versioning** : MLflow, DVC, Git

**7. TP proposé**

**Sujet : Déploiement d’un modèle de prédiction de revenus**

* Créer un modèle de classification avec scikit-learn
* Sauvegarder le modèle avec joblib
* Créer une API avec **FastAPI**
* Conteneuriser avec **Docker**
* Déployer sur **Render** ou **He**

**Chapitre 9 Études de cas & Projets**

**Objectifs :**

* Consolider vos acquis par la pratique sur des cas réels
* Apprendre à gérer un projet complet de ML de A à Z
* Développer des compétences en nettoyage, modélisation, validation, interprétation
* S’exercer à la présentation des résultats et à la rédaction de rapports

**1. Organisation**

Chaque étude de cas comporte :

* Description du problème métier
* Analyse exploratoire des données (EDA)
* Prétraitement et nettoyage
* Sélection et entraînement de modèles
* Évaluation et optimisation
* Interprétation et communication des résultats
* Livrables : code, rapport, présentation

**2. Étude de cas 1 : Prédiction du churn client (taux d’attrition)**

* **Contexte** : Une entreprise télécom veut prédire quels clients risquent de quitter
* **Données** : Profil client, historique consommation, plaintes, abonnements
* **Objectif** : Construire un modèle de classification binaire (churn / non churn)
* **Points clés** : Gestion des classes déséquilibrées, métriques adaptées (F1-score), interprétabilité (SHAP)

**3. Étude de cas 2 : Analyse prédictive des ventes**

* **Contexte** : Prévoir les ventes mensuelles d’un magasin
* **Données** : Historique des ventes, promotions, saisonnalité, météo
* **Objectif** : Modèle de régression pour prévoir le chiffre d’affaires
* **Points clés** : Feature engineering temporel, validation temporelle, gestion des données manquantes

**4. Étude de cas 3 : Détection d’anomalies dans les transactions bancaires**

* **Contexte** : Identifier les transactions frauduleuses
* **Données** : Historique des transactions, montants, localisation, heure
* **Objectif** : Apprentissage non supervisé (clustering) ou supervisé (classification)
* **Points clés** : Techniques d’oversampling, détection d’outliers, évaluation sur classes rares

**5. Étude de cas 4 : Classification d’images**

* **Contexte** : Classer des images de fruits en différentes catégories
* **Données** : Images prétraitées, labels
* **Objectif** : Construire un réseau de neurones convolutionnel simple (CNN)
* **Points clés** : Data augmentation, entraînement GPU, overfitting

**6. Étude de cas 5 : Traitement de texte (NLP)**

* **Contexte** : Analyse de sentiment sur des critiques de films
* **Données** : Textes, labels positifs/négatifs
* **Objectif** : Modèle de classification texte avec TF-IDF + Naive Bayes ou réseaux de neurones
* **Points clés** : Nettoyage de texte, tokenisation, métriques adaptées

**7. Gestion de projet et bonnes pratiques**

* Structurer un projet (dossiers, scripts, notebooks)
* Versionner le code (Git)
* Documenter et commenter
* Présenter des résultats clairs
* Gérer les données sensibles et éthiques

**8. TP final : Projet complet**

**Objectif :**

* Choisir une étude de cas ou proposer un projet personnel
* Appliquer toutes les étapes : exploration, modélisation, validation, interprétation, déploiement simple
* Rédiger un rapport final clair et complet
* Présenter oralement (si en formation)

**Résultats attendus**

* Modèle performant et validé
* Code propre et reproductible
* Rapport documenté avec analyses et visualisations
* Compréhension métier et techniques démontrée

**APPLICATION 01 :**

Prédire le **prix d’une maison** à partir de sa **surface (en m²)** en utilisant une **régression linéaire**.

# Importation des bibliothèques nécessaires

import numpy as np # Pour manipuler des tableaux numériques

import pandas as pd # Pour créer et gérer des tableaux de données

import matplotlib.pyplot as plt # Pour afficher les graphiques

from sklearn.linear\_model import LinearRegression # Régression linéaire

from sklearn.metrics import mean\_squared\_error, r2\_score # Évaluation des performances

# Création des données (surface en m² et prix en milliers d’euros)

data = {

'surface': [30, 50, 70, 90, 110], # Entrée (variable explicative)

'prix': [100, 150, 200, 250, 300] # Sortie (variable cible)

}

# Transformation en DataFrame pandas pour faciliter l’analyse

df = pd.DataFrame(data)

# Séparation des variables explicatives (X) et de la cible (y)

X = df[['surface']] # X doit être un tableau 2D (colonne surface)

y = df['prix'] # y est un tableau 1D (prix)

# Création et entraînement du modèle de régression linéaire

model = LinearRegression() # Instanciation du modèle

model.fit(X, y) # Apprentissage à partir des données

# Prédiction sur les mêmes surfaces (pour évaluer)

y\_pred = model.predict(X) # y\_pred contient les prix prédits

# Affichage des résultats

plt.scatter(X, y, color='blue', label="Données réelles") # Données d'origine

plt.plot(X, y\_pred, color='red', label="Régression linéaire") # Ligne de régression

plt.xlabel("Surface (m²)")

plt.ylabel("Prix (en milliers €)")

plt.title("Prix vs Surface")

plt.legend()

plt.grid(True)

plt.show()

**APPLICATIO 02 :**

Prédire si un étudiant **réussit ou échoue à un examen** en fonction de ses heures d’étude.

### ****Objectif****

Utiliser un **modèle de classification** (régression logistique) pour prédire une **étiquette binaire** (réussite = 1, échec = 0) à partir de données simples (heures d’étude).

# Importation des bibliothèques nécessaires

import numpy as np # Pour les tableaux numériques

import pandas as pd # Pour créer un DataFrame

import matplotlib.pyplot as plt # Pour les graphiques

from sklearn.linear\_model import LogisticRegression # Modèle de classification

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix # Évaluation

# Création des données

# X : heures d'étude

# y : 1 = réussite, 0 = échec

data = {

'heures': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],

'réussite': [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

}

# Création du DataFrame

df = pd.DataFrame(data)

python

CopierModifier

# Séparation des variables explicatives et de la cible

X = df[['heures']] # Variable d'entrée (2D)

y = df['réussite'] # Cible (1D)

# Création du modèle de régression logistique

model = LogisticRegression()

model.fit(X, y) # Entraînement du modèle

# Prédictions sur les mêmes données

y\_pred = model.predict(X) # Classe prédite (0 ou 1)

y\_prob = model.predict\_proba(X) # Probabilité associée à chaque classe

# Affichage des prédictions

print("Prédictions :", y\_pred)

print("Probabilités (classe 1) :", y\_prob[:, 1])

# Évaluation du modèle

accuracy = accuracy\_score(y, y\_pred)

print(f"Taux de bonne classification : {accuracy:.2f}")

# Rapport complet de classification

print("\nRapport de classification :")

print(classification\_report(y, y\_pred))

# Matrice de confusion

print("Matrice de confusion :")

print(confusion\_matrix(y, y\_pred))

# Visualisation

plt.scatter(X, y, color='blue', label='Données réelles') # Points d'origine

plt.plot(X, y\_prob[:, 1], color='red', label='Probabilité de réussite') # Courbe sigmoïde

plt.xlabel("Heures d'étude")

plt.ylabel("Probabilité de réussite")

plt.title("Classification binaire - Régression logistique")

plt.legend()

plt.grid(True)

plt.show()

### ****Interprétation des résultats****

| **Élément** | **Explication** |
| --- | --- |
| LogisticRegression() | Modèle de classification supervisé |
| fit(X, y) | Le modèle apprend à séparer les classes |
| predict(X) | Classe estimée (0 ou 1) |
| predict\_proba(X) | Probabilité pour chaque classe |
| accuracy\_score(y, y\_pred) | % de prédictions correctes |
| classification\_report | Précision, rappel, F1-score |
| confusion\_matrix | Répartition : Vrai Positif / Négatif, etc. |

### Exemple de sortie attendue

Prédictions : [0 0 0 0 0 1 1 1 1 1]

Probabilités (classe 1) : [0.03 0.07 0.15 0.31 0.5 0.69 0.84 0.93 0.97 0.99]

Taux de bonne classification : 1.00

Rapport de classification :

precision recall f1-score support

0 1.00 1.00 1.00 5

1 1.00 1.00 1.00 5

accuracy 1.00 10

macro avg 1.00 1.00 1.00 10

weighted avg 1.00 1.00 1.00 10

Matrice de confusion :

[[5 0]

[0 5]]

**APPLICATION 03 :**

**Segmentation de clients** à partir de leurs comportements d’achat.

Un supermarché a collecté des données sur ses clients : fréquence de visite, montant moyen dépensé, et fidélité (note sur 100).  
Vous devez segmenter ces clients pour adapter des campagnes marketing.

**Objectif**

* Utiliser le **clustering KMeans** pour segmenter des clients
* Visualiser les groupes pour interpréter les **profils types**
* Comprendre les applications business de l’apprentissage non supervisé

**Jeu de données simulé**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

import seaborn as sns

# Création de données clients simulées

np.random.seed(42)

data = {

'frequence\_visite': np.random.randint(1, 15, 100), # visites par mois

'depense\_moyenne': np.random.normal(200, 50, 100), # en €

'fidélité': np.random.randint(30, 100, 100) # score de fidélité sur 100

}

df = pd.DataFrame(data)

# Affichage

print(df.head())

**Clustering avec KMeans**

from sklearn.preprocessing import StandardScaler

# Normalisation des données

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(df)

# Modèle KMeans avec 3 clusters

kmeans = KMeans(n\_clusters=3, random\_state=42)

df['cluster'] = kmeans.fit\_predict(X\_scaled)

# Centres des clusters

centroids = pd.DataFrame(scaler.inverse\_transform(kmeans.cluster\_centers\_), columns=df.columns[:-1])

print("Centres des clusters :\n", centroids)

**Visualisation des segments**

# Visualisation 2D

sns.scatterplot(data=df, x='depense\_moyenne', y='fidélité', hue='cluster', palette='Set2')

plt.title("Segmentation de clients (KMeans)")

plt.xlabel("Dépense Moyenne (€)")

plt.ylabel("Score de Fidélité")

plt.grid(True)

plt.show()

**Interprétation possible des clusters**

| **Cluster** | **Fréquence** | **Dépense moyenne** | **Fidélité** | **Interprétation** |
| --- | --- | --- | --- | --- |
| 0 | Haute | Élevée | Élevée | Clients VIP |
| 1 | Moyenne | Moyenne | Moyenne | Clients standards |
| 2 | Faible | Basse | Basse | Clients occasionnels |

Ces insights peuvent aider à personnaliser les promotions :

* Récompenses pour les VIP
* Encouragements pour les fidèles moyens
* Incitations pour les occasionnels

**Explication pédagogique**

| **Étape** | **Rôle** |
| --- | --- |
| StandardScaler() | Normalise les données pour éviter le biais dû aux unités différentes |
| KMeans() | Regroupe les clients par similarité |
| fit\_predict() | Apprend les clusters et attribue chaque client |
| hue='cluster' | Colorie les clients selon leur groupe |

**APPLICATION 04 :**

Un hôpital cherche à **segmenter ses patients** afin d'identifier des groupes à risque pour mettre en place des campagnes de prévention ciblées.

**Étape 1 : Importation des bibliothèques**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.cluster import KMeans

from sklearn.preprocessing import StandardScaler

**Étape 2 : Création de données simulées**

np.random.seed(42)

# Génération de données médicales simulées

data = {

'age': np.random.randint(20, 80, 200), # Âge entre 20 et 80 ans

'imc': np.random.normal(25, 5, 200), # Indice de masse corporelle moyen = 25

'pression': np.random.normal(120, 15, 200) # Tension artérielle systolique

}

df = pd.DataFrame(data)

print(df.head())

**Commentaire :**

* age : indicateur de vieillissement
* imc : reflète le surpoids ou l'obésité
* pression : indicateur de santé cardiovasculaire

**Étape 3 : Normalisation des données**

# Mise à l'échelle des données

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(df)

 **Création d’un objet StandardScaler** de la bibliothèque scikit-learn.

 Cet objet sert à **standardiser les données**, c’est-à-dire les transformer pour qu’elles aient une moyenne nulle (0) et un écart-type égal à 1.

 Cette normalisation est importante pour que les variables, même si elles ont des unités ou échelles différentes, soient comparables et aient un poids équivalent dans les algorithmes.

* **fit\_transform** applique deux opérations successives sur le DataFrame df :
  1. **fit** : calcule la moyenne et l’écart-type de chaque colonne dans df.
  2. **transform** : utilise ces statistiques pour **centrer (soustraire la moyenne)** et **réduire (diviser par l’écart-type)** les données.
* Le résultat est un **tableau numpy X\_scaled** où chaque colonne est normalisée.
* Cette étape est cruciale avant d’appliquer des algorithmes sensibles aux échelles, comme le clustering KMeans, afin d’éviter qu’une variable domine les autres simplement parce qu’elle a des valeurs plus grandes.

Le **clustering est sensible aux échelles** → on standardise toutes les colonnes.

**Étape 4 : Clustering avec KMeans**

kmeans = KMeans(n\_clusters=3, random\_state=0)

df['cluster'] = kmeans.fit\_predict(X\_scaled)

fit\_predict() applique KMeans et affecte chaque patient à un groupe.

 **Création d’un modèle KMeans** avec la bibliothèque scikit-learn.

 n\_clusters=3 signifie que l’on souhaite **regrouper les données en 3 clusters (groupes)**.

 random\_state=0 fixe la graine aléatoire pour que les résultats soient **répétables** (mêmes clusters à chaque exécution).

 **fit\_predict() est une méthode** qui fait deux choses :

1. **fit(X\_scaled)** : entraîne le modèle KMeans sur les données normalisées X\_scaled, c’est-à-dire qu’il **cherche les centres des 3 groupes** qui minimisent la variance intra-groupe.
2. **predict(X\_scaled)** : attribue à chaque point (client/patient) le **numéro du cluster auquel il appartient** (0, 1 ou 2 ici).

 Ces numéros de cluster sont **ajoutés comme nouvelle colonne 'cluster'** dans le DataFrame df.

 Ainsi, on sait à quel groupe chaque individu appartient selon la segmentation KMeans.

**Étape 5 : Visualisation des clusters**

sns.scatterplot(data=df, x='imc', y='pression', hue='cluster', palette='Set2')

plt.title("Clustering des patients selon IMC et Pression")

plt.xlabel("IMC")

plt.ylabel("Pression artérielle")

plt.grid(True)

plt.show()

* Utilisation de la fonction **scatterplot de la bibliothèque Seaborn** pour tracer un nuage de points.
* data=df : la source des données est le DataFrame df.
* x='imc' : les valeurs de l’axe horizontal correspondent à la colonne imc (indice de masse corporelle).
* y='pression' : les valeurs de l’axe vertical correspondent à la colonne pression (pression artérielle).
* hue='cluster' : les points sont **colorés selon la colonne cluster**, c’est-à-dire selon le groupe auquel chaque patient appartient (résultat du clustering).
* palette='Set2' : choix d’une palette de couleurs agréable et distincte pour bien différencier les clusters.

**Étape 6 : Profil de chaque groupe**

# Moyennes des variables par cluster

profil\_clusters = df.groupby('cluster').mean()

print(profil\_clusters)

**Exemple d’interprétation :**

| **Cluster** | **Âge moyen** | **IMC moyen** | **Pression moyenne** | **Interprétation** |
| --- | --- | --- | --- | --- |
| 0 | 70 ans | 30 | 145 | Patients âgés et à risque cardiovasculaire |
| 1 | 40 ans | 22 | 110 | Patients jeunes, en bonne santé |
| 2 | 55 ans | 28 | 130 | Patients d’âge moyen, surpoids modéré |

**Conclusion**

* Nous avons utilisé **KMeans** pour segmenter des patients **sans étiquette**.
* Grâce à cette méthode, on peut **identifier des groupes à risque**, même sans diagnostic préalable.
* C'est une technique utile pour la **prévention**, le **triage**, ou l'**analyse exploratoire**.

**APPLICATION 05 :**

* Comprendre comment un agent d’apprentissage par renforcement peut apprendre à modifier une image par actions successives.
* Implémenter un agent qui ajuste la luminosité d’une image pour atteindre une luminosité cible.
* Appliquer la méthode Q-Learning dans un environnement très simple.

**Contexte**

L’agent reçoit une image en niveaux de gris et peut appliquer 3 actions :

* Augmenter la luminosité (+10 unités)
* Diminuer la luminosité (-10 unités)
* Ne rien faire

L’objectif est d’ajuster la luminosité moyenne de l’image pour qu’elle se rapproche d’une valeur cible (ex : 128 sur 0–255).

La récompense est basée sur la proximité à la luminosité cible : plus l’agent se rapproche, meilleure la récompense.

**Étape 1 : Préparation des bibliothèques**

import numpy as np

import matplotlib.pyplot as plt

**Étape 2 : Création d’une image simulée (niveaux de gris)**

# Image 8x8 pixels avec luminosité moyenne initiale ~ 100

image = np.ones((8, 8)) \* 100

**Étape 3 : Définition de l’environnement RL**

class BrightnessEnv:

def \_\_init\_\_(self, image, target=128):

self.image = image.copy()

self.target = target

self.state = self.get\_brightness()

def get\_brightness(self):

# Moyenne des pixels = état

return np.mean(self.image)

def step(self, action):

# Actions: 0=augmenter, 1=diminuer, 2=ne rien faire

if action == 0:

self.image = np.clip(self.image + 10, 0, 255)

elif action == 1:

self.image = np.clip(self.image - 10, 0, 255)

# Sinon pas de changement

self.state = self.get\_brightness()

# Récompense négative de la distance à la cible (on veut minimiser l’écart)

reward = -abs(self.target - self.state)

# Episode terminé si on est très proche de la cible (écart ≤ 1)

done = abs(self.target - self.state) <= 1

return self.state, reward, done

**Étape 4 : Initialisation de la table Q**

# États discrets : luminosité moyenne arrondie (0-255)

# 3 actions possibles : augmenter, diminuer, rien

Q = np.zeros((256, 3))

**Étape 5 : Paramètres RL**

alpha = 0.1 # taux apprentissage

gamma = 0.9 # facteur discount

epsilon = 0.2 # exploration

episodes = 500

**Étape 6 : Boucle d’apprentissage Q-Learning**

env = BrightnessEnv(image)

for ep in range(episodes):

state = int(env.state)

done = False

while not done:

# Politique epsilon-greedy

if np.random.rand() < epsilon:

action = np.random.choice(3)

else:

action = np.argmax(Q[state])

next\_state, reward, done = env.step(action)

next\_state = int(next\_state)

# Mise à jour Q

Q[state, action] = Q[state, action] + alpha \* (reward + gamma \* np.max(Q[next\_state]) - Q[state, action])

state = next\_state

# Reset image pour nouvel épisode

env.image = image.copy()

env.state = env.get\_brightness()

**Étape 7 : Test de la politique apprise**

env = BrightnessEnv(image)

state = int(env.state)

done = False

actions\_taken = []

while not done:

action = np.argmax(Q[state])

state, reward, done = env.step(action)

state = int(state)

actions\_taken.append(action)

print("Actions prises pour ajuster la luminosité :", actions\_taken)

print("Luminosité finale :", state)

**Explications**

| **Élément** | **Description** |
| --- | --- |
| **État** | Luminosité moyenne actuelle de l’image (entier 0–255) |
| **Actions** | Augmenter, diminuer ou rien (3 actions) |
| **Récompense** | Négative de la distance à la luminosité cible, donc l’agent cherche à réduire l’écart |
| **Q-Table** | Matrice 256x3 estimant la valeur d’une action dans un état donné |
| **Politique epsilon-greedy** | Balance entre exploration aléatoire et exploitation du meilleur choix |

**Conclusion**

* Cet exercice illustre comment un agent RL peut apprendre à modifier une image en agissant directement sur ses pixels (ici par une abstraction simple).
* Ce principe peut être étendu à des tâches plus complexes, par exemple ajustement automatique, restauration ou filtrage via RL.