Résumé Complet - Machine Learning

Formules, Algorithmes et Scripts Python

AmineGR03

26 janvier 2026

Table des matières

Introduction

Paramètres vs Hyperparamètres

Important

Définitions :

- **Paramètres** : Variables apprises par le modèle pendant l'entraînement (ex : poids w, biais b)
- **Hyperparamètres** : Paramètres fixés avant l'entraînement, non appris (ex : taux d'apprentissage α, nombre de voisins k)

Types d'Apprentissage

- **Apprentissage supervisé** : Données étiquetées (régression, classification)
- **Apprentissage non supervisé** : Données non étiquetées (clustering)

Régression Linéaire Simple

Modèle Mathématique

Formule

Équation de la régression linéaire simple :

$$\hat{Y} = aX + b \tag{1}$$

où :

- X : variable indépendante (ex : YearsExperience - années d'expérience)
- Y : variable dépendante (ex : Salary - salaire annuel)
- $\hat{Y}$: valeur prédite par le modèle
- a : pente (coefficient) - mesure de combien Y change quand X augmente d'une unité
- b : ordonnée à l'origine (biais) - valeur de Y lorsque $X = 0$

Interprétation des paramètres :

- b : Salaire de base (ou d'embauche) d'une personne sans expérience
- a : Augmentation de salaire pour chaque année d'expérience supplémentaire
  - Si $a > 0$: La droite "monte", Y augmente avec X
  - Si $a < 0$: La droite "descend", Y diminue avec X
  - Si $a = 0$: La droite est horizontale, X n'a aucune influence sur Y

Fonction de Coût (MSE)

Formule

Mean Squared Error (MSE) - Erreur Quadratique Moyenne :

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

où :

- n : nombre d'observations
- y_i : valeur réelle observée pour l'exemple i
- $ax_i + b$: valeur prédite par le modèle pour l'exemple i
- $(y_i - (ax_i + b))$: résidu (erreur) pour l'exemple i

Objectif : Minimiser $J(a, b)$ pour trouver les meilleurs paramètres a et b.
Note : On met les erreurs au carré pour éviter que les erreurs positives et négatives s'annulent.

Gradient Descent (Descente de Gradient)

Formule

Mise à jour des paramètres :

$$A_i = A_{i-1} - \alpha \frac{\partial J}{\partial A_{i-1}} \tag{3}$$

$$B_i = B_{i-1} - \alpha \frac{\partial J}{\partial B_{i-1}} \tag{4}$$

Dérivées partielles :

$$\frac{\partial J}{\partial A} = \frac{1}{n}\sum_{i=1}^{n}(y_i - (ax_i + b)) \cdot (-x_i) = -\frac{1}{n}\sum_{i=1}^{n}(y_i - (ax_i + b)) \cdot x_i \tag{5}$$

$$\frac{\partial J}{\partial B} = \frac{1}{n}\sum_{i=1}^{n}(y_i - (ax_i + b)) \cdot (-1) = -\frac{1}{n}\sum_{i=1}^{n}(y_i - (ax_i + b)) \tag{6}$$

où :

- α : taux d'apprentissage (learning rate) - hyperparamètre qui contrôle la taille du pas
- A_i, B_i : valeurs des paramètres à l'itération i
- Si α trop grand : risque de dépasser le minimum
- Si α trop petit : convergence trop lente

Processus itératif : À chaque étape, la fonction coût diminue jusqu'à atteindre (ou approcher) le minimum.

Solution Analytique (Équations Normales)

Formule

Formule directe (sans itération) :

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{7}$$

où $\mathbf{X}$ est la matrice des features avec une colonne de 1 pour le biais.

Script Python : Entraînement et Prédiction (TP)

Exemple

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# ===== PARTIE 1 : Chargement des données =====
# Chargement du dataset Salary_Data.csv
fichier_local = "Salary_Data.csv"
```

```
data = pd.read_csv(fichier_local)

# Aperçu des données
print("Aperçu :")
print(data.head())

# Informations
print("\nInfos :")
print(data.info())

# Statistiques descriptives
print("\nDescribe :")
print(data.describe())

# Variables
# X : YearsExperience (années d'expérience)
# Y : Salary (salaire annuel)

# ===== PARTIE 2 : Visualisation initiale =====
plt.figure(figsize=(7, 5))
plt.scatter(data['YearsExperience'], data['Salary'])
plt.title("Salaire en fonction des années d'expérience")
plt.xlabel("Années d'expérience")
plt.ylabel("Salaire")
plt.grid(True)
plt.show()

# ===== PARTIE 3 : Préparation des données (train/test) =====
X = data[['YearsExperience']]  # variable explicative
y = data['Salary']              # variable cible

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

print("Taille train :", X_train.shape, " | Taille test :", X_test.
    shape)

# ===== PARTIE 4 : Entra nement du modèle =====
model = LinearRegression()
model.fit(X_train, y_train)

# Paramètres appris
a = model.coef_[0]      # Pente
b = model.intercept_    # Ordonnée à l'origine

print("Pente (a) :", a)
print("Ordonnée à l'origine (b) :", b)

# Interprétation :
# - b : Salaire de base pour 0 années d'expérience
# - a : Augmentation de salaire par année d'expérience

# ===== PARTIE 5 : Visualisation de la droite de régression =====
y_pred_train = model.predict(X_train)

```

```
plt.figure(figsize=(7, 5))
plt.scatter(X_train, y_train, label='Données (train)')
plt.plot(X_train, y_pred_train, color='red', linewidth=2,
         label='Droite de régression')
plt.title("Ajustement du modèle sur les données d'entra nement")
plt.xlabel("Années d'expérience")
plt.ylabel("Salaire")
plt.legend()
plt.grid(True)
plt.show()

# ===== PARTIE 6 : Évaluation sur l'ensemble de test =====
y_pred_test = model.predict(X_test)

mse = mean_squared_error(y_test, y_pred_test)
r2 = r2_score(y_test, y_pred_test)

print("MSE :", mse)
print("R   :", r2)

# Interprétation R   :
# - R   proche de 1 : Le modèle explique bien la variance
# - R   faible : Le modèle explique mal la variance

# ===== PARTIE 7 : Prédiction sur une nouvelle valeur =====
# Exemple : prédire le salaire pour 8.5 années d'expérience
experience_nouvelle = [[8.5]]
salaire_prevu = model.predict(experience_nouvelle)
print(f"Salaire prédit pour 8.5 années d'expérience : {
    salaire_prevu[0]:.2f}")

# ===== IMPLÉMENTATION MANUELLE AVEC GRADIENT DESCENT =====
class LinearRegressionGD:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate  #    (alpha)
        self.n_iterations = n_iterations
        self.a = 0  # Pente
        self.b = 0  # Ordonnée à l'origine
        self.cost_history = []

    def fit(self, X, y):
        """
        Entra nement avec gradient descent
        """
        n = len(X)
        X = X.flatten()  # Convertir en vecteur 1D

        for iteration in range(self.n_iterations):
            # Prédictions : Y = a*X + b
            y_pred = self.a * X + self.b

            # Calcul du coût : J(a,b) = (1/2n) *   (y - (a*x + b))

            cost = (1/(2*n)) * np.sum((y_pred - y)**2)
            self.cost_history.append(cost)

```

```
            # Calcul des gradients
            dJ_da = -(1/n) * np.sum((y - y_pred) * X)
            dJ_db = -(1/n) * np.sum(y - y_pred)

            # Mise à jour : A = A -    *  J / A
            self.a -= self.learning_rate * dJ_da
            self.b -= self.learning_rate * dJ_db

        return self

    def predict(self, X):
        """Prédiction : Y = a*X + b"""
        X = X.flatten()
        return self.a * X + self.b

# Utilisation de l'implémentation manuelle
model_gd = LinearRegressionGD(learning_rate=0.01, n_iterations
    =1000)
model_gd.fit(X_train.values, y_train.values)

print(f"\nModèle Gradient Descent :")
print(f"Pente (a) : {model_gd.a:.2f}")
print(f"Ordonnée à l'origine (b) : {model_gd.b:.2f}")

# Visualisation de l'évolution du coût
plt.figure(figsize=(10, 5))
plt.plot(model_gd.cost_history)
plt.xlabel('Itérations')
plt.ylabel('Coût (MSE)')
plt.title('Évolution du Coût avec Gradient Descent')
plt.grid(True)
plt.show()
```

Régression Linéaire Multiple

Modèle Mathématique

Formule

Équation de la régression linéaire multiple :

$$\hat{Y} = b + a_1 \times X_1 + a_2 \times X_2 + \cdots + a_n \times X_n \tag{8}$$

Exemple concret :

$$\text{Salaire} = b + a_1 \times \text{Expérience} + a_2 \times \text{Âge} + a_3 \times \text{Niveau d'études} \tag{9}$$

Forme vectorielle :

$$\hat{Y} = \mathbf{X}^T \cdot \mathbf{A} + B \tag{10}$$

où :

- $\mathbf{X} = [X_1, X_2, \ldots, X_n]$: Feature vector (vecteur de features)
- $\mathbf{A} = [a_1, a_2, \ldots, a_n]$: Vecteur de poids (coefficients)
- B : Biais (salaire de base)
- $\hat{Y}$: Prédiction

Notation matricielle :

$$\mathbf{Y} = \mathbf{XA} + B \tag{11}$$

où $\mathbf{X}$ est la matrice des features (m lignes = m exemples, n colonnes = n features).

Fonction de Coût

Formule

MSE pour régression multiple :

$$J(\mathbf{w}) = \frac{1}{2m} \sum_{i=1}^{m} (h(\mathbf{x}^{(i)}) - y^{(i)})^2 = \frac{1}{2m} (\mathbf{Xw} - \mathbf{y})^T (\mathbf{Xw} - \mathbf{y}) \tag{12}$$

Gradient Descent

Formule

Mise à jour vectorielle :

$$\mathbf{w} := \mathbf{w} - \alpha \nabla J(\mathbf{w}) \tag{13}$$

Gradient :

$$\nabla J(\mathbf{w}) = \frac{1}{m} \mathbf{X}^T (\mathbf{Xw} - \mathbf{y}) \tag{14}$$

Forme développée :

$$\frac{\partial J}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(h(\mathbf{x}^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{15}$$

Solution Analytique

Formule

Équations normales :

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{16}$$

Script Python : Entraînement et Prédiction

Exemple

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, r2_score

class MultipleLinearRegression:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.cost_history = []

    def fit(self, X, y):
        """
        Entra nement avec gradient descent
        X: matrice (m, n) - m exemples, n features
        y: vecteur (m,)
        """
        m, n = X.shape

        # Ajouter colonne de 1 pour le biais
        X_bias = np.c_[np.ones(m), X]

        # Initialisation des poids
        self.weights = np.zeros(n + 1)

        for iteration in range(self.n_iterations):
            # Prédictions
            y_pred = X_bias.dot(self.weights)

            # Calcul du coût
            cost = (1/(2*m)) * np.sum((y_pred - y)**2)
            self.cost_history.append(cost)

            # Calcul du gradient
            gradient = (1/m) * X_bias.T.dot(y_pred - y)
```

```
38
39              # Mise à jour des poids
40              self.weights -= self.learning_rate * gradient
41
42          return self
43
44      def predict(self, X):
45          """
46          Prédiction
47          """
48          m = X.shape[0]
49          X_bias = np.c_[np.ones(m), X]
50          return X_bias.dot(self.weights)
51
52      def score(self, X, y):
53          """
54          Score R
55          """
56          y_pred = self.predict(X)
57          ss_res = np.sum((y - y_pred)**2)
58          ss_tot = np.sum((y - np.mean(y))**2)
59          return 1 - (ss_res / ss_tot)
60
61  # Utilisation avec Boston Housing Dataset
62  # Chargement des données
63  data = pd.read_excel('Boston_Housing_Dataset.xlsx')
64  X = data.drop('MEDV', axis=1).values  # Features
65  y = data['MEDV'].values  # Prix des maisons
66
67  # Normalisation
68  scaler = StandardScaler()
69  X_scaled = scaler.fit_transform(X)
70
71  # Division train/test
72  X_train, X_test, y_train, y_test = train_test_split(
73      X_scaled, y, test_size=0.2, random_state=42
74  )
75
76  # Création et entra nement
77  model = MultipleLinearRegression(learning_rate=0.01, n_iterations
        =1000)
78  model.fit(X_train, y_train)
79
80  # Prédictions
81  y_train_pred = model.predict(X_train)
82  y_test_pred = model.predict(X_test)
83
84  # Métriques
85  print("=== Métriques d'Évaluation ===")
86  print(f"MSE Train: {mean_squared_error(y_train, y_train_pred):.2f}"
        )
87  print(f"MSE Test: {mean_squared_error(y_test, y_test_pred):.2f}")
88  print(f"RMSE Train: {np.sqrt(mean_squared_error(y_train,
        y_train_pred)):.2f}")
89  print(f"RMSE Test: {np.sqrt(mean_squared_error(y_test, y_test_pred)
        ):.2f}")
```

```
print(f" R   Train: {model.score(X_train, y_train):.4f}")
print(f" R   Test: {model.score(X_test, y_test):.4f}")

# Coefficients
print("\n=== Coefficients ===")
feature_names = data.drop('MEDV', axis=1).columns
for i, (name, coef) in enumerate(zip(['Biais'] + list(feature_names
    ), model.weights)):
     print(f"{name}: {coef:.4f}")
```

Régression Polynomiale

Concept

La régression polynomiale transforme les features en polynômes de degré d pour capturer des relations non-linéaires.

Formule

Modèle polynomial de degré d :

$$\hat{Y} = B + a_1 X + a_2 X^2 + \cdots + a_d X^d \tag{17}$$

Exemples :

- **Degré 1** : $\hat{Y} = B + a_1 X$ (régression linéaire simple)
- **Degré 2** : $\hat{Y} = B + a_1 X + a_2 X^2$ (parabole)
- **Degré n** : $\hat{Y} = B + a_1 X + a_2 X^2 + \cdots + a_n X^n$

Transformation des features :

$$\mathbf{X}_{poly} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^d \\ 1 & x_2 & x_2^2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^d \end{bmatrix} \tag{18}$$

BIC (Bayesian Information Criterion)

Formule

Formule du BIC :

$$BIC = n \ln(MSE) + K \ln(n) \tag{19}$$

où :

- n : nombre d'observations
- MSE : erreur quadratique moyenne du modèle
- K : degré du polynôme (nombre de paramètres)
- ln : logarithme naturel

Interprétation :

- $n \ln(MSE)$: Mesure la qualité d'ajustement (récompense les modèles qui s'ajustent bien)
  - Plus le MSE est faible, plus ce terme est petit
  - Le ln rend la comparaison plus stable
  - Multiplier par n tient compte de la taille du dataset
- $K \ln(n)$: Pénalité pour la complexité (pénalise les modèles trop complexes)
  - Plus K (degré) est grand, plus la pénalité est forte
  - Évite le sur-apprentissage (overfitting)

Objectif : Minimiser le BIC pour trouver le meilleur compromis entre qualité d'ajustement et simplicité.

Évolution du BIC :

— Le BIC diminue d'abord car le modèle s'ajuste mieux (MSE baisse)
— Puis il remonte quand le modèle devient trop complexe (pénalité $K \ln(n)$ augmente)
— Le minimum du BIC indique le degré optimal

Script Python : Entraînement et Prédiction avec BIC (TP)

Exemple

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

# ===== PARTIE 1 : Chargement du dataset Boston Housing =====
boston = fetch_openml(name='boston', version=1, as_frame=True)
data = boston.frame
data = data.drop('B', axis=1)  # Supprimer colonne problématique

# Variable cible : MEDV (prix médian des maisons)
# Variable explicative : LSTAT (pourcentage de statut inférieur)

# ===== PARTIE 2 : Visualisation initiale =====
X = data[['LSTAT']].values
y = data['MEDV'].values

plt.figure(figsize=(8, 6))
plt.scatter(X, y, alpha=0.6)
plt.xlabel('LSTAT')
plt.ylabel('MEDV')
plt.title('Relation LSTAT - MEDV')
plt.grid(True)
plt.show()

# ===== PARTIE 3 : Régression linéaire simple (degré 1) =====
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

model_linear = LinearRegression()
model_linear.fit(X_train, y_train)

y_pred_linear = model_linear.predict(X_test)
mse_linear = mean_squared_error(y_test, y_pred_linear)
r2_linear = r2_score(y_test, y_pred_linear)

print(f"Régression Linéaire (degré 1):")
print(f"MSE: {mse_linear:.2f}, R  : {r2_linear:.4f}")

# ===== PARTIE 4 : Régression polynomiale degré 2 =====
poly2 = PolynomialFeatures(degree=2)
X_poly2_train = poly2.fit_transform(X_train)
```

```
X_poly2_test = poly2.transform(X_test)

model_poly2 = LinearRegression()
model_poly2.fit(X_poly2_train, y_train)

y_pred_poly2 = model_poly2.predict(X_poly2_test)
mse_poly2 = mean_squared_error(y_test, y_pred_poly2)
r2_poly2 = r2_score(y_test, y_pred_poly2)

print(f"\nRégression Polynomiale (degré 2):")
print(f"MSE: {mse_poly2:.2f}, R  : {r2_poly2:.4f}")

# ===== PARTIE 5 : Étude du degré du polynôme avec BIC =====
degrees = [1, 2, 3, 4, 5, 6]
results = []

for degree in degrees:
    # Création des variables polynomiales
    poly = PolynomialFeatures(degree=degree)
    X_poly_train = poly.fit_transform(X_train)
    X_poly_test = poly.transform(X_test)

    # Entra nement
    model = LinearRegression()
    model.fit(X_poly_train, y_train)

    # Prédictions
    y_pred_train = model.predict(X_poly_train)
    y_pred_test = model.predict(X_poly_test)

    # Métriques
    mse_train = mean_squared_error(y_train, y_pred_train)
    mse_test = mean_squared_error(y_test, y_pred_test)
    r2_train = r2_score(y_train, y_pred_train)
    r2_test = r2_score(y_test, y_pred_test)

    # Calcul du BIC : BIC = n * ln(MSE) + K * ln(n)
    n = len(y_test)
    K = degree  # Degré du polynôme
    bic = n * np.log(mse_test) + K * np.log(n)

    results.append({
        'degree': degree,
        'MSE_train': mse_train,
        'MSE_test': mse_test,
        'R2_train': r2_train,
        'R2_test': r2_test,
        'BIC': bic
    })

    print(f"\nDegré {degree}:")
    print(f"  MSE Train: {mse_train:.2f}, MSE Test: {mse_test:.2f}"
        )
    print(f"  R  Train: {r2_train:.4f}, R  Test: {r2_test:.4f}")
    print(f"  BIC: {bic:.2f}")

```

```python
# Création d'un DataFrame pour visualisation
df_results = pd.DataFrame(results)
print("\n=== Tableau Récapitulatif ===")
print(df_results.to_string(index=False))

# ===== PARTIE 6 : Visualisation du BIC =====
plt.figure(figsize=(12, 5))

# Graphique 1 : BIC en fonction du degré
plt.subplot(1, 2, 1)
plt.plot(df_results['degree'], df_results['BIC'], 'o-', linewidth
    =2, markersize=8)
plt.xlabel('Degré du polynôme (K)')
plt.ylabel('BIC')
plt.title('BIC en fonction du degré du polynôme')
plt.grid(True)
plt.xticks(degrees)

# Identifier le minimum
min_bic_idx = df_results['BIC'].idxmin()
min_degree = df_results.loc[min_bic_idx, 'degree']
min_bic = df_results.loc[min_bic_idx, 'BIC']
plt.axvline(x=min_degree, color='r', linestyle='--',
            label=f'Minimum (K={min_degree})')
plt.legend()

# Graphique 2 : MSE et R  en fonction du degré
plt.subplot(1, 2, 2)
plt.plot(df_results['degree'], df_results['MSE_test'], 'o-',
         label='MSE Test', linewidth=2)
plt.plot(df_results['degree'], df_results['R2_test'], 's-',
         label='R  Test', linewidth=2)
plt.xlabel('Degré du polynôme')
plt.ylabel('Métrique')
plt.title('MSE et R  en fonction du degré')
plt.legend()
plt.grid(True)
plt.xticks(degrees)

plt.tight_layout()
plt.show()

print(f"\n=== Meilleur modèle selon BIC ===")
print(f"Degré optimal: {min_degree}")
print(f"BIC minimum: {min_bic:.2f}")

# ===== PARTIE 7 : Visualisation des modèles =====
X_plot = np.linspace(X.min(), X.max(), 300).reshape(-1, 1)

plt.figure(figsize=(12, 8))
plt.scatter(X, y, alpha=0.5, label='Données', s=30)

colors = ['red', 'blue', 'green', 'orange', 'purple', 'brown']
for idx, degree in enumerate(degrees):
    poly = PolynomialFeatures(degree=degree)
    X_poly_plot = poly.fit_transform(X_plot)
```

```
    model = LinearRegression()
    X_poly_train = poly.fit_transform(X_train)
    model.fit(X_poly_train, y_train)
    y_plot = model.predict(X_poly_plot)

    plt.plot(X_plot, y_plot, color=colors[idx], linewidth=2,
             label=f'Degré {degree} (BIC={df_results.loc[idx, "BIC
                "]:.1f})')

plt.xlabel('LSTAT')
plt.ylabel('MEDV')
plt.title('Comparaison des modèles polynomiales')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Questions d'analyse :
# 1. Comment évoluent MSE et R  lorsque le degré augmente ?
#        R  augmente toujours (sur train), mais peut diminuer sur
    test (overfitting)
# 2. Pourquoi le R  augmente toujours ?
#        Plus de paramètres = meilleur ajustement aux données d'
    entra nement
# 3. Que signifie le paramètre K dans la formule du BIC ?
#        K = degré du polynôme (nombre de paramètres)
# 4. Quel est le degré donnant le plus petit BIC ?
#        C'est le meilleur compromis entre qualité et simplicité
```

Important

Attention au sur-apprentissage (overfitting) : Un degré trop élevé peut mener à un modèle qui mémorise les données d'entraînement mais généralise mal.

Classification : K-Nearest Neighbors (KNN)

Algorithme

Important

Principe : Un point est classé selon la classe majoritaire de ses k plus proches voisins.

Formules

Formule

Distance euclidienne (exemple avec 2 features) :

$$D(Z, A) = \sqrt{(Z_{PH} - A_{PH})^2 + (Z_{Diam} - A_{Diam})^2} \tag{20}$$

Forme générale (n features) :

$$D(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{l=1}^{n} (x_{il} - x_{jl})^2} \tag{21}$$

Exemple concret (plantes) :

- Plante Z : pH=4, Diam=3
- Plante A : pH=2, Diam=1
- Distance : $D(Z, A) = \sqrt{(4-2)^2 + (3-1)^2} = \sqrt{4+4} = \sqrt{8} = 2.83$

Décision de classification (Vote) :

- Pour $K = 1$: On regarde le 1 plus proche voisin, sa classe est la prédiction
- Pour $K = 3$: On regarde les 3 plus proches voisins, la classe majoritaire est la prédiction
- Pour $K = 5$: On regarde les 5 plus proches voisins, la classe majoritaire est la prédiction

Exemple de vote (K=3) :

- Voisin 1 : Comestible (0)
- Voisin 2 : Toxique (1)
- Voisin 3 : Comestible (0)
- Vote : 2 (Comestible) vs 1 (Toxique)
- Décision : COMESTIBLE

Hyperparamètres

- K : Nombre de voisins (hyperparamètre principal)
  - Généralement testé avec des valeurs impaires : 1, 3, 5, 7, 9, 11, 13, 15...
  - K ne doit pas dépasser la taille de l'ensemble d'entraînement
  - K trop petit (ex : K=1) : Sensible au bruit
  - K trop grand : Modèle trop "lisse", ignore les détails locaux
- **Métrique de distance** : euclidienne, Manhattan, Minkowski, etc.
- **Poids** : uniforme ou distance (les voisins proches comptent plus)

Optimisation du K (Grid Search)

Important

Processus d'optimisation :

1. **Répartition des données** : 80% Training Set, 20% Test Set
2. **Test méthodique** : Tester plusieurs valeurs de K (1, 3, 5, 7, 9...)
3. **Évaluation** : Pour chaque K, calculer la précision sur le Test Set
4. **Sélection** : Choisir le K qui donne la meilleure précision

Exemple de résultats :

— K=1 : Précision = 85% (sensible au bruit)
— K=3 : Précision = 94%
— K=5 : Précision = 96% (optimal)
— K=7 : Précision = 95.5%
— K=25 : Précision = 85% (trop lisse)

Le pic de la courbe (K=5) représente le meilleur compromis Biais/Variance.

Script Python : Entraînement et Prédiction

Exemple

```python
import numpy as np
from collections import Counter
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report

class KNN:
    def __init__(self, k=3, distance_metric='euclidean', weights='
        uniform'):
        self.k = k
        self.distance_metric = distance_metric
        self.weights = weights
        self.X_train = None
        self.y_train = None

    def _euclidean_distance(self, x1, x2):
        """Distance euclidienne"""
        return np.sqrt(np.sum((x1 - x2)**2))

    def _manhattan_distance(self, x1, x2):
        """Distance de Manhattan"""
        return np.sum(np.abs(x1 - x2))

    def _compute_distance(self, x1, x2):
        """Calcul de la distance selon la métrique"""
        if self.distance_metric == 'euclidean':
            return self._euclidean_distance(x1, x2)
        elif self.distance_metric == 'manhattan':
            return self._manhattan_distance(x1, x2)
```

```
29          else:
30              raise ValueError("Métrique non supportée")
31
32      def fit(self, X, y):
33          """
34          Entra nement (KNN est un algorithme lazy, on stocke juste
                les données)
35          """
36          self.X_train = X
37          self.y_train = y
38          return self
39
40      def predict(self, X):
41          """
42          Prédiction
43          """
44          predictions = []
45          for x in X:
46              # Calculer les distances à tous les points d'
                    entra nement
47              distances = [self._compute_distance(x, x_train)
48                           for x_train in self.X_train]
49
50              # Obtenir les k plus proches voisins
51              k_indices = np.argsort(distances)[:self.k]
52              k_nearest_labels = [self.y_train[i] for i in k_indices]
53
54              if self.weights == 'uniform':
55                  # Vote majoritaire simple
56                  most_common = Counter(k_nearest_labels).most_common
                        (1)
57                  predictions.append(most_common[0][0])
58              else:  # weights='distance'
59                  # Vote pondéré par l'inverse de la distance
60                  k_distances = [distances[i] for i in k_indices]
61                  weights = [1/d if d != 0 else 1e10 for d in
                        k_distances]
62                  weighted_votes = {}
63                  for label, weight in zip(k_nearest_labels, weights)
                        :
64                      weighted_votes[label] = weighted_votes.get(
                            label, 0) + weight
65                  predictions.append(max(weighted_votes, key=
                        weighted_votes.get))
66
67          return np.array(predictions)
68
69      def predict_proba(self, X):
70          """
71          Probabilités de prédiction
72          """
73          probabilities = []
74          for x in X:
75              distances = [self._compute_distance(x, x_train)
76                           for x_train in self.X_train]
77              k_indices = np.argsort(distances)[:self.k]
```

```
            k_nearest_labels = [self.y_train[i] for i in k_indices]

            # Probabilités basées sur les fréquences
            label_counts = Counter(k_nearest_labels)
            total = sum(label_counts.values())
            proba = {label: count/total for label, count in
                label_counts.items()}
            probabilities.append(proba)

        return probabilities

# Utilisation
from sklearn.datasets import load_iris

# Chargement des données
iris = load_iris()
X, y = iris.data, iris.target

# Normalisation (important pour KNN)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Division train/test
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)

# Test avec différents k
for k in [1, 3, 5, 7, 10]:
    model = KNN(k=k, distance_metric='euclidean', weights='uniform'
        )
    model.fit(X_train, y_train)

    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

    print(f"k={k:2d}: Accuracy = {accuracy:.4f}")

# Meilleur modèle
best_k = 3
model = KNN(k=best_k, distance_metric='euclidean', weights='
    distance')
model.fit(X_train, y_train)

# Prédictions
y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)

# Métriques
print("\n=== Métriques d'Évaluation ===")
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print("\nMatrice de confusion:")
print(confusion_matrix(y_test, y_pred))
print("\nRapport de classification:")
print(classification_report(y_test, y_pred, target_names=iris.
    target_names))
```

Classification : Support Vector Machine (SVM)

Introduction

Important

Qu'est-ce qu'un SVM ?
Le SVM est un algorithme d'apprentissage supervisé utilisé principalement pour :

- **Classification** : Assigner des données à différentes catégories
- **Régression** : Prédire des valeurs continues (SVR)

Principe fondamental : Trouver le meilleur hyperplan qui sépare les données de différentes classes en maximisant la distance (marge) entre cet hyperplan et les points les plus proches de chaque classe.

Pourquoi les SVM sont populaires ?

- Performance élevée même avec des données complexes
- Robustesse face au surapprentissage grâce à la maximisation de la marge
- Capacité à gérer des espaces de haute dimension
- Base théorique solide

SVM Linéaire : Principe

Formule

Hyperplan de séparation :

$$f(\mathbf{x}) = \mathbf{W}^T \cdot \mathbf{X} + B = 0 \tag{22}$$

Frontières de marge :

$$\mathbf{W}^T \cdot \mathbf{X} + B = +1 \quad \text{(Frontière gauche, Classe +1)} \tag{23}$$

$$\mathbf{W}^T \cdot \mathbf{X} + B = 0 \quad \text{(Frontière de décision)} \tag{24}$$

$$\mathbf{W}^T \cdot \mathbf{X} + B = -1 \quad \text{(Frontière droite, Classe -1)} \tag{25}$$

Règle de décision :

- Si $f(\mathbf{x}) > +1$: Classe +1
- Si $f(\mathbf{x}) < -1$: Classe -1
- Si $f(\mathbf{x}) = 0$: Sur la frontière de décision

Objectif : Maximiser la marge (distance entre les deux hyperplans parallèles) pour maximiser la capacité de généralisation.

Calcul de la Marge

Formule

Calcul de la distance entre les deux hyperplans :
Soit deux points support vectors :

- $\mathbf{x}^+$ sur $\mathbf{W}^T \cdot \mathbf{x}^+ + B = +1$
- $\mathbf{x}^-$ sur $\mathbf{W}^T \cdot \mathbf{x}^- + B = -1$

On a : $\mathbf{x}^+ = \mathbf{x}^- + r\mathbf{w}$ où r est la distance le long de la direction perpendiculaire.
Calcul de r :

$$\mathbf{W}^T \cdot (\mathbf{x}^- + r\mathbf{w}) + B = +1 \tag{26}$$
$$r\|\mathbf{w}\|^2 + \mathbf{W}^T \cdot \mathbf{x}^- + B = +1 \tag{27}$$
$$r\|\mathbf{w}\|^2 - 1 = +1 \tag{28}$$
$$r\|\mathbf{w}\|^2 = 2 \tag{29}$$
$$r = \frac{2}{\|\mathbf{w}\|^2} \tag{30}$$

Marge : La distance entre les deux hyperplans est $r = \frac{2}{\|\mathbf{w}\|^2}$
Objectif : Maximiser la marge $\Rightarrow$ Minimiser $\|\mathbf{w}\|^2$

SVM Hard Margin (Marge Rigide)

Formule

Problème primal :

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \quad \text{sous la contrainte} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \tag{31}$$

Conditions :

- $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$ pour tous les points
- $y_i = +1 : \mathbf{w}^T\mathbf{x}_i + b \geq 1$
- $y_i = -1 : \mathbf{w}^T\mathbf{x}_i + b \leq -1$

Forme unifiée : $y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1$

Optimisation avec Multiplicateurs de Lagrange

Formule

Lagrangien :

$$L(\mathbf{w}, b, \boldsymbol{\varphi}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_i \varphi_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1] \tag{32}$$

où $\varphi_i \geq 0$ sont les multiplicateurs de Lagrange.
Conditions KKT (Karush-Kuhn-Tucker) :

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_i \varphi_i y_i \mathbf{x}_i \tag{33}$$
$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_i \varphi_i y_i = 0 \tag{34}$$
$$\varphi_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1] = 0 \quad \text{(condition de complémentarité)} \tag{35}$$

Interprétation :

- Si $\varphi_i = 0$: Point loin de la marge, ne contribue pas à la solution
- Si $\varphi_i > 0$: Point sur la marge (Support Vector), définit $\mathbf{w}$

— $\mathbf{w} = \sum_{\text{support vectors}} \varphi_i y_i \mathbf{x}_i$

Parcimonie du modèle SVM :

— Dataset : 10,000 points d'entraînement

— Support vectors : 50-200 points seulement (1-2%)

— Points avec $\varphi_i = 0$: 9,800 points (98%)

— Impact : Modèle final très compact, prédiction rapide

Problème Dual

Formule

Formulation duale :

$$\max_{\boldsymbol{\alpha}} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \tag{36}$$

Sous les contraintes :

$$\alpha_i \geq 0 \quad \forall i \tag{37}$$

$$\sum_i \alpha_i y_i = 0 \tag{38}$$

Avantages du dual :

— Plus facile à résoudre numériquement

— Les données apparaissent uniquement via des produits scalaires $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$

— Permet l'utilisation du kernel trick pour les cas non-linéaires

SVM Soft Margin (Marge Souple)

Important

Pourquoi le Hard Margin ne suffit pas ?
Le SVM Hard Margin suppose que les données sont parfaitement séparables :

— Aucun point ne doit être dans la marge

— Aucun point ne doit être mal classé

Problème : Dans la réalité, les données contiennent du bruit, des valeurs aberrantes et des classes qui se chevauchent.

Formule

Slack variables (Variables de relâchement) :
On introduit $\xi_i \geq 0$ pour permettre à un point de violer la contrainte de marge.
Contrainte modifiée :

$$y_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0 \tag{39}$$

Interprétation de ξ_i :

— $\xi_i = 0$: Point bien classé, hors de la marge

— $0 < \xi_i < 1$: Point dans la marge mais bien classé

— $\xi_i \geq 1$: Point mal classé

Problème d'optimisation :

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_i \xi_i \tag{40}$$

Sous les contraintes :

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i \tag{41}$$

$$\xi_i \geq 0 \tag{42}$$

Paramètre C :

— C grand : Les erreurs coûtent cher ⇒ modèle très strict (marge étroite)

— C petit : Plus de flexibilité ⇒ modèle plus tolérant (marge large)

— C contrôle le compromis entre maximiser la marge et pénaliser les erreurs

Problème dual (Soft Margin) :

$$\max_{\boldsymbol{\alpha}} \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i\alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j \rangle \tag{43}$$

Sous les contraintes :

$$0 \leq \alpha_i \leq C \quad \forall i \tag{44}$$

$$\sum_i \alpha_i y_i = 0 \tag{45}$$

Différence avec Hard Margin : La contrainte $\alpha_i \leq C$ limite la valeur maximale des multiplicateurs de Lagrange.

SVM Non-Linéaire : Kernel Trick

Important

Limitation du SVM linéaire :
Le SVM linéaire fonctionne parfaitement lorsque les classes sont séparables par une ligne droite (ou un hyperplan). Mais que faire lorsque les données ont une structure plus complexe (ex : cercles concentriques) ?
La solution : SVM non-linéaire
Grâce au kernel trick, nous transformons les données dans un espace de dimension supérieure où elles deviennent linéairement séparables, sans calculer explicitement cette transformation.

Formule

Qu'est-ce qu'un noyau (kernel) ?
Un noyau est une fonction de similarité entre deux points. Dans un SVM, cette similarité est calculée sous forme de produit scalaire.

Kernel linéaire :

$$K(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle = \mathbf{x}_i^T \mathbf{x}_j \tag{46}$$

Transformation dans un espace de dimension supérieure :

$$\mathbf{x} \to \phi(\mathbf{x}) \tag{47}$$

Dans ce nouvel espace, la similarité devient :

$$K(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle \tag{48}$$

Kernel trick : Permet de calculer cette similarité comme si l'on travaillait dans l'espace transformé, sans jamais calculer explicitement $\phi(\mathbf{x})$.
Formulation duale avec kernel :

$$\max_{\boldsymbol{\alpha}} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \tag{49}$$

Les données interviennent uniquement via le kernel $K(\mathbf{x}_i, \mathbf{x}_j)$.

Kernels Courants

Formule

1. Kernel Linéaire :

$$K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j \tag{50}$$

Quand l'utiliser :

- Données linéairement séparables
- Données très haute dimension (texte, TF-IDF, bag-of-words)
- Modèle très rapide et robuste
- Souvent le meilleur choix quand features >> nombre d'échantillons

2. Kernel Polynomial :

$$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T \mathbf{x}_j + c)^d \tag{51}$$

où :

- d : degré du polynôme
- c : coefficient (souvent 1)

Quand l'utiliser :

- Relations non linéaires mais régulières
- Utile quand on soupçonne une relation quadratique/cubique entre les features
- Frontières non linéaires mais "propres"
- Attention : peut devenir coûteux si le degré est élevé

Exemple (degré 2) :

$$(\mathbf{x}_i^T \mathbf{x}_j + c)^2 = (\mathbf{x}_i^T \mathbf{x}_j)^2 + 2c(\mathbf{x}_i^T \mathbf{x}_j) + c^2 \tag{52}$$

3. Kernel RBF (Radial Basis Function / Gaussien) :

$$K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2} \tag{53}$$

où γ (gamma) contrôle l'influence des points proches.

Quand l'utiliser :

- Données fortement non linéaires
- Frontières complexes, tordues ou irrégulières
- Cas général : souvent le kernel le plus performant
- Mapping implicite en dimension infinie

Interprétation de γ :

- γ petit (ex : 0.1) : Vue large, points éloignés semblent encore similaires
- γ moyen (ex : 1) : Équilibré, influence locale modérée
- γ grand (ex : 5) : Vue étroite, seuls les points très proches comptent (risque d'overfitting)

Exemple de calcul RBF :

- Points : $\mathbf{x}_i = [0, 0]$, $A = [1, 0]$, $B = [3, 0]$
- Distance : $D_A = 1$, $D_B = 3$
- Pour $\gamma = 0.1$: $K(\mathbf{x}_i, A) = e^{-0.1 \times 1} = 0.90$, $K(\mathbf{x}_i, B) = e^{-0.1 \times 9} = 0.41$
- Pour $\gamma = 1$: $K(\mathbf{x}_i, A) = e^{-1} = 0.37$, $K(\mathbf{x}_i, B) = e^{-9} \approx 0.00$
- Pour $\gamma = 5$: $K(\mathbf{x}_i, A) = e^{-5} = 0.006$, $K(\mathbf{x}_i, B) = e^{-45} \approx 0.00$

Script Python : SVM Linéaire (TP)

Exemple

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report

# ===== PARTIE 1 : Chargement des données =====
# Fichier : iris_svm_lineaire_pret.csv
data = pd.read_csv('iris_svm_lineaire_pret.csv')

# Aperçu
print("Aperçu des données :")
print(data.head())
print(f"\nNombre d'observations : {len(data)}")

# ===== PARTIE 2 : Préparation des données =====
X = data[['feature1', 'feature2']].values  # Features
y = data['class'].values                   # Classe cible

# Séparation train/test
```

```python
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Normalisation (recommandé pour SVM)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# ===== PARTIE 3 : Entra nement d'un SVM Linéaire =====
# Créer un modèle SVM linéaire avec C = 1
svm_linear = SVC(kernel='linear', C=1.0, random_state=42)
svm_linear.fit(X_train_scaled, y_train)

# Prédictions
y_pred_train = svm_linear.predict(X_train_scaled)
y_pred_test = svm_linear.predict(X_test_scaled)

# Métriques
print("\n=== Métriques SVM Linéaire ===")
print(f"Accuracy Train: {accuracy_score(y_train, y_pred_train):.4f}
    ")
print(f"Accuracy Test: {accuracy_score(y_test, y_pred_test):.4f}")

# Informations sur les support vectors
print(f"\nNombre de vecteurs de support: {len(svm_linear.support_)}
    ")
print(f"Support vectors indices: {svm_linear.support_}")
print(f"Support vectors: \n{svm_linear.support_vectors_}")

# Coefficients
print(f"\nCoefficients (w): {svm_linear.coef_[0]}")
print(f"Biais (b): {svm_linear.intercept_[0]}")

# ===== PARTIE 4 : Visualisation =====
def plot_svm_decision_boundary(X, y, model, title):
    """Visualiser l'hyperplan, les marges et les support vectors"""
    plt.figure(figsize=(10, 8))

    # Nuage de points
    plt.scatter(X[y == 0, 0], X[y == 0, 1], c='red', marker='o',
                label='Classe 0', s=50, alpha=0.6)
    plt.scatter(X[y == 1, 0], X[y == 1, 1], c='blue', marker='s',
                label='Classe 1', s=50, alpha=0.6)

    # Support vectors
    plt.scatter(model.support_vectors_[:, 0],
                model.support_vectors_[:, 1],
                s=200, facecolors='none', edgecolors='black',
                linewidth=2, label='Support Vectors')

    # Frontière de décision
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

```

```
    # Créer une grille pour la frontière
    xx = np.linspace(xlim[0], xlim[1], 100)
    yy = np.linspace(ylim[0], ylim[1], 100)
    YY, XX = np.meshgrid(yy, xx)
    xy = np.vstack([XX.ravel(), YY.ravel()]).T
    Z = model.decision_function(xy).reshape(XX.shape)

    # Contour de la frontière de décision
    ax.contour(XX, YY, Z, colors='black', levels=[-1, 0, 1],
               alpha=0.5, linestyles=['--', '-', '--'], linewidths
                  =2)

    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(title)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

# Visualisation
plot_svm_decision_boundary(X_train_scaled, y_train, svm_linear,
                           'SVM Linéaire - Hyperplan et Marges')

# ===== PARTIE 5 : Test avec différentes valeurs de C =====
C_values = [0.1, 1, 10, 100]
results = []

for C in C_values:
    svm = SVC(kernel='linear', C=C, random_state=42)
    svm.fit(X_train_scaled, y_train)

    y_pred = svm.predict(X_test_scaled)
    acc = accuracy_score(y_test, y_pred)
    n_sv = len(svm.support_)

    results.append({
        'C': C,
        'Accuracy': acc,
        'N_Support_Vectors': n_sv
    })

    print(f"C={C:6.1f}: Accuracy={acc:.4f}, Support Vectors={n_sv}"
       )

df_results = pd.DataFrame(results)
print("\n=== Résultats selon C ===")
print(df_results.to_string(index=False))

# Questions d'analyse :
# 1. Quand C est grand, la marge devient-elle large ou stricte ?
#        C grand : marge stricte (étroite), moins d'erreurs toléré
   es
# 2. Combien de support vectors y a-t-il pour chaque C ?
#        C grand : plus de support vectors (modèle plus strict)
```

Script Python : SVM RBF (Non-Linéaire) (TP)

Exemple

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix

# ===== PARTIE 1 : Chargement des données =====
# Fichier : breast_cancer_svm_rbf_2d_ready.csv
data = pd.read_csv('breast_cancer_svm_rbf_2d_ready.csv')

print("Aperçu des données :")
print(data.head())
print(f"\nNombre d'observations : {len(data)}")

# ===== PARTIE 2 : Préparation des données =====
X = data[['feature1', 'feature2']].values
y = data['target'].values

# Visualisation initiale
plt.figure(figsize=(8, 6))
plt.scatter(X[y == 0, 0], X[y == 0, 1], c='red', marker='o',
            label='Classe 0', s=50, alpha=0.6)
plt.scatter(X[y == 1, 0], X[y == 1, 1], c='blue', marker='s',
            label='Classe 1', s=50, alpha=0.6)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Données - Structure Non-Linéaire')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Séparation train/test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Normalisation
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# ===== PARTIE 3 : SVM Linéaire (modèle de référence) =====
svm_linear = SVC(kernel='linear', C=1.0, random_state=42)
svm_linear.fit(X_train_scaled, y_train)

y_pred_linear = svm_linear.predict(X_test_scaled)
acc_linear = accuracy_score(y_test, y_pred_linear)

print(f"\n=== SVM Linéaire ===")
print(f"Accuracy: {acc_linear:.4f}")
print(f"Support Vectors: {len(svm_linear.support_)}")
```

```
54
55 # ===== PARTIE 4 : SVM RBF =====
56 # Créer un modèle SVM à noyau RBF (C=1, gamma=1)
57 svm_rbf = SVC(kernel='rbf', C=1.0, gamma=1.0, random_state=42)
58 svm_rbf.fit(X_train_scaled, y_train)
59
60 y_pred_rbf = svm_rbf.predict(X_test_scaled)
61 acc_rbf = accuracy_score(y_test, y_pred_rbf)
62
63 print(f"\n=== SVM RBF (C=1, gamma=1) ===")
64 print(f"Accuracy: {acc_rbf:.4f}")
65 print(f"Support Vectors: {len(svm_rbf.support_)}")
66
67 # ===== PARTIE 5 : Test avec différentes valeurs de C et gamma
    =====
68 C_values = [0.1, 1, 10, 100]
69 gamma_values = [0.01, 0.1, 1, 10]
70
71 results = []
72
73 for C in C_values:
74     for gamma in gamma_values:
75         svm = SVC(kernel='rbf', C=C, gamma=gamma, random_state=42)
76         svm.fit(X_train_scaled, y_train)
77
78         y_pred = svm.predict(X_test_scaled)
79         acc = accuracy_score(y_test, y_pred)
80         n_sv = len(svm.support_)
81
82         results.append({
83             'C': C,
84             'gamma': gamma,
85             'Accuracy': acc,
86             'N_Support_Vectors': n_sv
87         })
88
89 df_results = pd.DataFrame(results)
90 print("\n=== Résultats selon C et gamma ===")
91 print(df_results.to_string(index=False))
92
93 # Identifier les meilleurs paramètres
94 best_idx = df_results['Accuracy'].idxmax()
95 best_params = df_results.loc[best_idx]
96 print(f"\n=== Meilleurs paramètres ===")
97 print(f"C: {best_params['C']}, gamma: {best_params['gamma']}")
98 print(f"Accuracy: {best_params['Accuracy']:.4f}")
99
100 # ===== PARTIE 6 : Visualisation des frontières =====
101 def plot_svm_rbf_decision_boundary(X, y, model, title):
102     """Visualiser la frontière de décision RBF"""
103     plt.figure(figsize=(10, 8))
104
105     # Nuage de points
106     plt.scatter(X[y == 0, 0], X[y == 0, 1], c='red', marker='o',
107                 label='Classe 0', s=50, alpha=0.6)
108     plt.scatter(X[y == 1, 0], X[y == 1, 1], c='blue', marker='s',
```

```
                label='Classe 1', s=50, alpha=0.6)

    # Support vectors
    plt.scatter(model.support_vectors_[:, 0],
                model.support_vectors_[:, 1],
                s=200, facecolors='none', edgecolors='black',
                linewidth=2, label='Support Vectors')

    # Frontière de décision
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    xx = np.linspace(xlim[0], xlim[1], 100)
    yy = np.linspace(ylim[0], ylim[1], 100)
    YY, XX = np.meshgrid(yy, xx)
    xy = np.vstack([XX.ravel(), YY.ravel()]).T
    Z = model.decision_function(xy).reshape(XX.shape)

    # Contour de la frontière
    ax.contour(XX, YY, Z, colors='black', levels=[0],
               alpha=0.8, linestyles=['-'], linewidths=2)
    ax.contourf(XX, YY, Z, levels=[-np.inf, 0, np.inf],
                alpha=0.3, colors=['red', 'blue'])

    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(title)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

# Visualisation SVM RBF
svm_rbf_best = SVC(kernel='rbf', C=best_params['C'],
                   gamma=best_params['gamma'], random_state=42)
svm_rbf_best.fit(X_train_scaled, y_train)

plot_svm_rbf_decision_boundary(X_train_scaled, y_train,
    svm_rbf_best,
                               f'SVM RBF (C={best_params["C"]},
                                  gamma={best_params["gamma"]})')

# ===== PARTIE 7 : Comparaison Linéaire vs RBF =====
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# SVM Linéaire
ax = axes[0]
ax.scatter(X_train_scaled[y_train == 0, 0], X_train_scaled[y_train
    == 0, 1],
           c='red', marker='o', label='Classe 0', s=50, alpha=0.6)
ax.scatter(X_train_scaled[y_train == 1, 0], X_train_scaled[y_train
    == 1, 1],
           c='blue', marker='s', label='Classe 1', s=50, alpha=0.6)
ax.scatter(svm_linear.support_vectors_[:, 0], svm_linear.
    support_vectors_[:, 1],
           s=200, facecolors='none', edgecolors='black', linewidth
```

```
                =2)
xlim = ax.get_xlim()
ylim = ax.get_ylim()
xx = np.linspace(xlim[0], xlim[1], 100)
yy = np.linspace(ylim[0], ylim[1], 100)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = svm_linear.decision_function(xy).reshape(XX.shape)
ax.contour(XX, YY, Z, colors='black', levels=[0], linewidths=2)
ax.set_title(f'SVM Linéaire (Accuracy: {acc_linear:.4f})')
ax.legend()
ax.grid(True, alpha=0.3)

# SVM RBF
ax = axes[1]
ax.scatter(X_train_scaled[y_train == 0, 0], X_train_scaled[y_train
    == 0, 1],
           c='red', marker='o', label='Classe 0', s=50, alpha=0.6)
ax.scatter(X_train_scaled[y_train == 1, 0], X_train_scaled[y_train
    == 1, 1],
           c='blue', marker='s', label='Classe 1', s=50, alpha=0.6)
ax.scatter(svm_rbf_best.support_vectors_[:, 0], svm_rbf_best.
    support_vectors_[:, 1],
           s=200, facecolors='none', edgecolors='black', linewidth
               =2)
xlim = ax.get_xlim()
ylim = ax.get_ylim()
xx = np.linspace(xlim[0], xlim[1], 100)
yy = np.linspace(ylim[0], ylim[1], 100)
YY, XX = np.meshgrid(yy, xx)
xy = np.vstack([XX.ravel(), YY.ravel()]).T
Z = svm_rbf_best.decision_function(xy).reshape(XX.shape)
ax.contour(XX, YY, Z, colors='black', levels=[0], linewidths=2)
ax.contourf(XX, YY, Z, levels=[-np.inf, 0, np.inf],
            alpha=0.3, colors=['red', 'blue'])
ax.set_title(f'SVM RBF (Accuracy: {acc_rbf:.4f})')
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Questions d'analyse :
# 1. Pourquoi le SVM linéaire échoue-t-il sur ces données ?
#        Les données ne sont pas linéairement séparables
# 2. Comment le kernel RBF capture-t-il la structure non-linéaire ?
#        Transformation implicite dans un espace de dimension supé
    rieure
# 3. Quel est l'effet de gamma sur la frontière ?
#        gamma petit : frontière lisse, gamma grand : frontière
    complexe (risque overfitting)
```

Hyperparamètres et Paramètres

Important

Hyperparamètres (fixés avant l'entraînement) :

- C : Paramètre de régularisation
  - Contrôle le compromis entre maximiser la marge et pénaliser les erreurs
  - C grand : Modèle strict, marge étroite, moins d'erreurs tolérées
  - C petit : Modèle tolérant, marge large, plus d'erreurs tolérées
- **Kernel** : Type de kernel (linéaire, polynomial, RBF)
- γ (pour RBF) : Contrôle l'influence des points proches
  - γ petit : Vue large, frontière lisse
  - γ grand : Vue étroite, frontière complexe (risque d'overfitting)
- **degree** (pour polynomial) : Degré du polynôme
- **coef0** (pour polynomial) : Coefficient constant

Paramètres (appris pendant l'entraînement) :

- **w** : Vecteur de poids (coefficients)
- b : Biais (ordonnée à l'origine)
- α_i : Multiplicateurs de Lagrange
- **Support vectors** : Points sur les frontières de marge

Classification : Arbre de Décision

Concept

Un arbre de décision partitionne récursivement l'espace des features selon des règles de décision.

Mesures d'Impureté

Formule

Entropie :

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i) \tag{54}$$

Gini Impurity :

$$G(S) = 1 - \sum_{i=1}^{c} p_i^2 \tag{55}$$

Information Gain (Gain d'information) :

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v) \tag{56}$$

où :

- S : ensemble d'exemples
- A : attribut (feature)
- p_i : proportion de la classe i
- c : nombre de classes
- S_v : sous-ensemble où $A = v$

Algorithme ID3 (récursif)

Hyperparamètres

- **max_depth** : Profondeur maximale de l'arbre
- **min_samples_split** : Nombre minimum d'échantillons pour diviser un nœud
- **min_samples_leaf** : Nombre minimum d'échantillons dans une feuille
- **criterion** : 'gini' ou 'entropy'
- **max_features** : Nombre maximum de features à considérer

Algorithm 1 Construction d'un Arbre de Décision

Require: Dataset D, Features F, Classe cible
Ensure: Arbre de décision
  if Tous les exemples ont la même classe **then**
    return Nœud feuille avec cette classe
  end if
  if F est vide **then**
    return Nœud feuille avec classe majoritaire
  end if
  A^* = attribut avec le meilleur Information Gain
  Créer nœud racine avec A^*
  for chaque valeur v de A^* **do**
    D_v = sous-ensemble de D où $A^* = v$
    if D_v est vide **then**
      Ajouter feuille avec classe majoritaire
    else
      Ajouter sous-arbre récursif avec D_v et $F \setminus \{A^*\}$
    end if
  end for
  return Arbre

Overfitting et Élagage

Important

Overfitting (Sur-apprentissage) :

- L'arbre mémorise les données d'entraînement plutôt que d'apprendre des règles générales
- Accuracy très élevée sur train, mais faible sur test
- Arbre très profond avec beaucoup de branches
- Certaines feuilles contiennent très peu d'exemples

Élagage préventif (pre-pruning) : Limite la croissance de l'arbre via hyperparamètres.

- `max_depth` : Profondeur maximale
- `min_samples_split` : Nombre minimum d'échantillons pour diviser
- `min_samples_leaf` : Nombre minimum d'échantillons dans une feuille

Élagage a posteriori (post-pruning) : Supprime des branches après construction.
Courbes d'apprentissage :

- Quand la profondeur augmente, l'accuracy train continue d'augmenter
- L'accuracy test stagne ou diminue après un certain point
- La divergence entre train et test indique l'overfitting

Compromis Biais-Variance :

- Arbre très profond : Biais faible, Variance élevée (overfitting)
- Arbre très simple : Biais élevé, Variance faible (underfitting)
- Arbre optimal : Meilleur compromis (meilleure accuracy sur test)

Script Python : Entraînement et Prédiction

Exemple

```python
import numpy as np
from collections import Counter
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix,
    classification_report
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

class DecisionTree:
    def __init__(self, max_depth=None, min_samples_split=2,
                 min_samples_leaf=1, criterion='gini'):
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.criterion = criterion
        self.tree = None

    def _gini(self, y):
        """Calcul de l'impureté Gini"""
        if len(y) == 0:
            return 0
        counts = Counter(y)
        proportions = [count/len(y) for count in counts.values()]
        return 1 - sum(p**2 for p in proportions)

    def _entropy(self, y):
        """Calcul de l'entropie"""
        if len(y) == 0:
            return 0
        counts = Counter(y)
        proportions = [count/len(y) for count in counts.values()]
        return -sum(p * np.log2(p) if p > 0 else 0 for p in
            proportions)

    def _impurity(self, y):
        """Calcul de l'impureté selon le critère"""
        if self.criterion == 'gini':
            return self._gini(y)
        else:  # entropy
            return self._entropy(y)

    def _information_gain(self, y_parent, y_left, y_right):
        """Calcul du gain d'information"""
        parent_impurity = self._impurity(y_parent)
        n = len(y_parent)
        n_left, n_right = len(y_left), len(y_right)

        if n == 0:
            return 0

        child_impurity = (n_left/n) * self._impurity(y_left) + \
                         (n_right/n) * self._impurity(y_right)

```

```
52          return parent_impurity - child_impurity
53
54      def _best_split(self, X, y):
55          """Trouve le meilleur split"""
56          best_gain = -1
57          best_feature = None
58          best_threshold = None
59
60          n_features = X.shape[1]
61
62          for feature_idx in range(n_features):
63              # Trier les valeurs uniques
64              thresholds = np.unique(X[:, feature_idx])
65
66              for threshold in thresholds:
67                  # Division
68                  left_mask = X[:, feature_idx] <= threshold
69                  right_mask = ~left_mask
70
71                  if np.sum(left_mask) == 0 or np.sum(right_mask) ==
                        0:
72                      continue
73
74                  y_left = y[left_mask]
75                  y_right = y[right_mask]
76
77                  # Gain d'information
78                  gain = self._information_gain(y, y_left, y_right)
79
80                  if gain > best_gain:
81                      best_gain = gain
82                      best_feature = feature_idx
83                      best_threshold = threshold
84
85          return best_feature, best_threshold, best_gain
86
87      def _build_tree(self, X, y, depth=0):
88          """Construction récursive de l'arbre"""
89          n_samples = len(y)
90
91          # Critères d'arrêt
92          if (self.max_depth is not None and depth >= self.max_depth)
                or \
93             n_samples < self.min_samples_split or \
94             len(np.unique(y)) == 1:
95              return {'class': Counter(y).most_common(1)[0][0], 'leaf
                    ': True}
96
97          # Meilleur split
98          feature, threshold, gain = self._best_split(X, y)
99
100         if gain == 0:  # Pas d'amélioration
101             return {'class': Counter(y).most_common(1)[0][0], 'leaf
                    ': True}
102
103         # Division
```

```
        left_mask = X[:, feature] <= threshold
        right_mask = ~left_mask

        if np.sum(left_mask) < self.min_samples_leaf or \
           np.sum(right_mask) < self.min_samples_leaf:
            return {'class': Counter(y).most_common(1)[0][0], 'leaf
                ': True}

        # Construction récursive
        node = {
            'feature': feature,
            'threshold': threshold,
            'left': self._build_tree(X[left_mask], y[left_mask],
               depth + 1),
            'right': self._build_tree(X[right_mask], y[right_mask],
                depth + 1),
            'leaf': False
        }

        return node

    def fit(self, X, y):
        """Entra nement"""
        self.tree = self._build_tree(X, y)
        return self

    def _predict_sample(self, x, node):
        """Prédiction pour un échantillon"""
        if node['leaf']:
            return node['class']

        if x[node['feature']] <= node['threshold']:
            return self._predict_sample(x, node['left'])
        else:
            return self._predict_sample(x, node['right'])

    def predict(self, X):
        """Prédiction"""
        return np.array([self._predict_sample(x, self.tree) for x
           in X])

# Utilisation avec sklearn (version optimisée)
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris

# Chargement des données
iris = load_iris()
X, y = iris.data, iris.target

# Division train/test
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Test avec différents hyperparamètres
print("=== Test des Hyperparamètres ===")
```

```python
for max_depth in [None, 3, 5, 10]:
    for min_samples_split in [2, 5, 10]:
        model = DecisionTreeClassifier(
            max_depth=max_depth,
            min_samples_split=min_samples_split,
            criterion='gini',
            random_state=42
        )
        model.fit(X_train, y_train)
        train_acc = accuracy_score(y_train, model.predict(X_train))
        test_acc = accuracy_score(y_test, model.predict(X_test))
        print(f"max_depth={max_depth}, min_samples_split={
            min_samples_split}: "
              f"Train={train_acc:.4f}, Test={test_acc:.4f}")

# Meilleur modèle
best_model = DecisionTreeClassifier(
    max_depth=3,
    min_samples_split=5,
    criterion='gini',
    random_state=42
)
best_model.fit(X_train, y_train)

# Prédictions
y_pred = best_model.predict(X_test)

# Métriques
print("\n=== Métriques d'Évaluation ===")
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print("\nMatrice de confusion:")
print(confusion_matrix(y_test, y_pred))
print("\nRapport de classification:")
print(classification_report(y_test, y_pred, target_names=iris.
    target_names))

# Visualisation de l'arbre
plt.figure(figsize=(20, 10))
plot_tree(best_model, feature_names=iris.feature_names,
          class_names=iris.target_names, filled=True)
plt.title("Arbre de Décision")
plt.show()
```

Clustering : K-Means

Algorithme

Important

Principe : Partitionner les données en k clusters en minimisant la somme des distances au carré entre les points et les centroïdes de leurs clusters.

Formules

Formule

Fonction objectif (Inertia) :

$$J = \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \tag{57}$$

Centroïde d'un cluster :

$$\boldsymbol{\mu}_i = \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x} \tag{58}$$

où :

- k : nombre de clusters
- C_i : ensemble des points du cluster i
- $\boldsymbol{\mu}_i$: centroïde du cluster i

Algorithme K-Means

Algorithm 2 K-Means

Require: Données $\mathbf{X}$, nombre de clusters k
Ensure: Centroïdes $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_k$
Initialiser aléatoirement k centroïdes
repeat
Étape d'assignation : Assigner chaque point au cluster le plus proche
$C_i = \{\mathbf{x} : ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \leq ||\mathbf{x} - \boldsymbol{\mu}_j||^2, \forall j\}$
Étape de mise à jour : Recalculer les centroïdes
$\boldsymbol{\mu}_i = \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x}$
until Convergence (centroïdes ne changent plus)
return Centroïdes et assignations

Hyperparamètres

- k : Nombre de clusters (hyperparamètre principal)
- **max_iter** : Nombre maximum d'itérations
- **init** : Méthode d'initialisation ('random', 'k-means++')
- **n_init** : Nombre d'initialisations différentes
- **tol** : Tolérance pour la convergence

Méthode du Coude (Elbow Method)

Important

Pour choisir k, on trace l'inertia en fonction de k et on cherche le "coude" dans la courbe.

Script Python : Entraînement et Prédiction

Exemple

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score, davies_bouldin_score

class KMeansClustering:
    def __init__(self, k=3, max_iter=300, init='k-means++', n_init=10):
        self.k = k
        self.max_iter = max_iter
        self.init = init
        self.n_init = n_init
        self.centroids = None
        self.labels = None
        self.inertia = None

    def _initialize_centroids(self, X):
        """Initialisation des centro des"""
        if self.init == 'random':
            indices = np.random.choice(len(X), self.k, replace=False)
            return X[indices]
        elif self.init == 'k-means++':
            # K-means++ : choisir le premier centro de alé atoirement,
            # puis choisir les suivants avec probabilité proportionnelle
            # à la distance au centro de le plus proche
            centroids = [X[np.random.randint(len(X))]]

            for _ in range(self.k - 1):
                distances = np.array([
                    min([np.linalg.norm(x - c)**2 for c in centroids])
                    for x in X
                ])
                probabilities = distances / distances.sum()
                cumulative_probs = probabilities.cumsum()
                r = np.random.rand()
                idx = np.searchsorted(cumulative_probs, r)
                centroids.append(X[idx])

            return np.array(centroids)

    def _assign_clusters(self, X, centroids):
```

```
        """Assigner chaque point au cluster le plus proche"""
        distances = np.sqrt(((X - centroids[:, np.newaxis])**2).sum
            (axis=2))
        return np.argmin(distances, axis=0)

    def _update_centroids(self, X, labels):
        """Mettre à jour les centro des"""
        centroids = np.array([X[labels == i].mean(axis=0)
                              for i in range(self.k)])
        return centroids

    def _compute_inertia(self, X, labels, centroids):
        """Calculer l'inertia"""
        inertia = 0
        for i in range(self.k):
            cluster_points = X[labels == i]
            if len(cluster_points) > 0:
                inertia += np.sum((cluster_points - centroids[i])
                    **2)
        return inertia

    def fit(self, X):
        """Entra nement"""
        best_inertia = float('inf')
        best_centroids = None
        best_labels = None

        # Essayer plusieurs initialisations
        for init_idx in range(self.n_init):
            # Initialisation
            centroids = self._initialize_centroids(X)

            for iteration in range(self.max_iter):
                # Assignation
                labels = self._assign_clusters(X, centroids)

                # Nouveaux centro des
                new_centroids = self._update_centroids(X, labels)

                # Vérifier la convergence
                if np.allclose(centroids, new_centroids):
                    break

                centroids = new_centroids

            # Calculer l'inertia
            inertia = self._compute_inertia(X, labels, centroids)

            # Garder le meilleur résultat
            if inertia < best_inertia:
                best_inertia = inertia
                best_centroids = centroids
                best_labels = labels

        self.centroids = best_centroids
        self.labels = best_labels
```

```
        self.inertia = best_inertia

        return self

    def predict(self, X):
        """Prédiction (assignation aux clusters)"""
        return self._assign_clusters(X, self.centroids)

# Utilisation
import pandas as pd

# Chargement des données (exemple avec données de céréales)
data = pd.read_csv('cereal.csv')
# Sélectionner des features numériques
features = ['calories', 'protein', 'fat', 'sodium', 'fiber', 'carbo
    ', 'sugars']
X = data[features].values

# Normalisation (important pour K-means)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Méthode du coude pour choisir k
inertias = []
silhouette_scores = []
k_range = range(2, 11)

for k in k_range:
    model = KMeans(n_clusters=k, random_state=42, n_init=10)
    model.fit(X_scaled)
    inertias.append(model.inertia_)
    silhouette_scores.append(silhouette_score(X_scaled, model.
        labels_))

# Visualisation de la méthode du coude
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

axes[0].plot(k_range, inertias, 'bo-')
axes[0].set_xlabel('Nombre de clusters (k)')
axes[0].set_ylabel('Inertia')
axes[0].set_title('Méthode du Coude')
axes[0].grid(True)

axes[1].plot(k_range, silhouette_scores, 'ro-')
axes[1].set_xlabel('Nombre de clusters (k)')
axes[1].set_ylabel('Score de Silhouette')
axes[1].set_title('Score de Silhouette')
axes[1].grid(True)

plt.tight_layout()
plt.show()

# Meilleur k (basé sur le score de silhouette)
best_k = k_range[np.argmax(silhouette_scores)]
print(f"Meilleur k selon silhouette: {best_k}")

```

```python
# Entra nement avec le meilleur k
model = KMeans(n_clusters=best_k, random_state=42, n_init=10)
model.fit(X_scaled)
labels = model.predict(X_scaled)

# Métriques
print(f"\n=== Métriques d'Évaluation ===")
print(f"Nombre de clusters: {best_k}")
print(f"Inertia: {model.inertia_:.2f}")
print(f"Score de Silhouette: {silhouette_score(X_scaled, labels):.4f}")
print(f"Score Davies-Bouldin: {davies_bouldin_score(X_scaled, labels):.4f}")

# Visualisation (si 2D ou avec PCA)
from sklearn.decomposition import PCA

if X_scaled.shape[1] > 2:
    pca = PCA(n_components=2)
    X_2d = pca.fit_transform(X_scaled)
else:
    X_2d = X_scaled

plt.figure(figsize=(10, 8))
scatter = plt.scatter(X_2d[:, 0], X_2d[:, 1], c=labels, cmap='viridis', alpha=0.6)
plt.scatter(model.cluster_centers_[:, 0] if X_scaled.shape[1] == 2
            else pca.transform(model.cluster_centers_)[:, 0],
            model.cluster_centers_[:, 1] if X_scaled.shape[1] == 2
            else pca.transform(model.cluster_centers_)[:, 1],
            c='red', marker='x', s=200, linewidths=3, label='Centro des')
plt.colorbar(scatter)
plt.xlabel('Première composante principale' if X_scaled.shape[1] > 2 else 'Feature 1')
plt.ylabel('Deuxième composante principale' if X_scaled.shape[1] > 2 else 'Feature 2')
plt.title(f'Clustering K-Means (k={best_k})')
plt.legend()
plt.grid(True)
plt.show()
```

Métriques d'Évaluation

Régression

Formule

Mean Squared Error (MSE) :

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{59}$$

Root Mean Squared Error (RMSE) :

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2} \tag{60}$$

Mean Absolute Error (MAE) :

$$\text{MAE} = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i| \tag{61}$$

Coefficient de Détermination (R^2) :

$$R^2 = 1 - \frac{\text{SS}_{res}}{\text{SS}_{tot}} = 1 - \frac{\text{Erreur du modèle}}{\text{Variation totale}} \tag{62}$$

où :

- $\textbf{SS}_{res}(Sommedescarrésdesrésidus) : Erreurdumodèle$
- $\textbf{SS}_{tot}(Sommedescarréstotaux) : Variationtotaleoù = 1\frac{}{n\sum_{i=1}^{n} y_i}$ est la moyenne des valeurs réelles

Interprétation :

- $R^2 = 1$: Le modèle explique toute la variation (parfait)
- $R^2 = 0$: Le modèle n'explique rien (équivalent à prédire la moyenne)
- $R^2 < 0$: Le modèle est pire que la moyenne

Exemple concret :

- 3 observations : Salaires de 40k, 50k, 90k
- Salaire moyen : $(40k + 50k + 90k)/3 = 60k$
- Erreur totale (SS_{tot}) :(40k-60k)^2 + $(50k - 60k)^2 + (90k - 60k)^2 = 1,400,000,000$
- Modèle : Salaire = 25k * Expérience + 15k
- Erreur du modèle (SS_{res}) :(40k-40k)^2 + $(50k - 65k)^2 + (90k - 90k)^2 = 225,000,000$
- $R^2 = 1 - (225,000,000/1,400,000,000) = 0.84$ (84%)

Classification

Formule

Matrice de Confusion (Classification Binaire) :
Exemple concret (200 plantes de test, K=5) :

- TP = 94 : 94 plantes toxiques bien classées comme toxiques
- TN = 98 : 98 plantes comestibles bien classées comme comestibles
- FP = 2 : 2 plantes comestibles classées comme toxiques (moins grave)
- FN = 6 : 6 plantes toxiques classées comme comestibles (très grave !)
- Total = 94 + 98 + 2 + 6 = 200

Accuracy (Exactitude) :

$$ACCURACY = \frac{\text{Prédictions correctes}}{\text{Total des prédictions}} = \frac{TP + TN}{TP + TN + FP + FN} \tag{63}$$

Exemple : $(94 + 98)/200 = 192/200 = 0.96$ (96%)
Precision (Précision) :

$$PRECISION = \frac{TP}{TP + FP} \tag{64}$$

Exemple : $94/(94 + 2) = 94/96 = 0.979$ (97.9%)
Interprétation : Parmi les plantes prédites comme "Toxiques", 97.9% l'étaient vraiment. La précision indique la fiabilité des prédictions positives.
Recall (Rappel) :

$$RECALL = \frac{TP}{TP + FN} \tag{65}$$

Exemple : $94/(94 + 6) = 94/100 = 0.94$ (94%)
Interprétation : Sur toutes les plantes réellement toxiques, on en a trouvé 94%. Le modèle a manqué 6% des plantes toxiques (les FN).
F1-Score :

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{66}$$

Choix de la métrique selon le contexte :

- **Accuracy** : Métrique générale, mais peut être trompeuse si classes déséquilibrées
- **Precision** : Prioritaire si les Faux Positifs sont coûteux
- **Recall** : Prioritaire si les Faux Négatifs sont dangereux (ex : plantes toxiques, fraudes)
- Dans un problème de toxicité, le **Recall** est la métrique prioritaire car on préfère avoir plus de Faux Positifs que de Faux Négatifs

Clustering

Formule

Inertia (Within-cluster sum of squares) :

$$\text{Inertia} = \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \tag{67}$$

Score de Silhouette :

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \tag{68}$$

où :

- $a(i)$: distance moyenne aux points du même cluster
- $b(i)$: distance moyenne aux points du cluster le plus proche

Score Davies-Bouldin :

$$DB = \frac{1}{k} \sum_{i=1}^{k} \max_{j \neq i} \left(\frac{\sigma_i + \sigma_j}{d(\boldsymbol{\mu}_i, \boldsymbol{\mu}_j)} \right) \tag{69}$$

où σ_i est l'écart-type moyen des distances dans le cluster i.

Script Python : Calcul des Métriques

Exemple

```python
import numpy as np
from sklearn.metrics import (
    mean_squared_error, mean_absolute_error, r2_score,
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report,
    silhouette_score, davies_bouldin_score
)

# ===== MÉTRIQUES DE RÉGRESSION =====
def regression_metrics(y_true, y_pred):
    """Calculer toutes les métriques de régression"""
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    mae = mean_absolute_error(y_true, y_pred)
    r2 = r2_score(y_true, y_pred)

    print("=== Métriques de Régression ===")
    print(f"MSE:  {mse:.4f}")
    print(f"RMSE: {rmse:.4f}")
    print(f"MAE:  {mae:.4f}")
    print(f"R  :   {r2:.4f}")

    return {'MSE': mse, 'RMSE': rmse, 'MAE': mae, 'R  ': r2}

# ===== MÉTRIQUES DE CLASSIFICATION =====
def classification_metrics(y_true, y_pred, average='weighted'):
```

```
    """Calculer toutes les métriques de classification"""
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average=average,
        zero_division=0)
    recall = recall_score(y_true, y_pred, average=average,
        zero_division=0)
    f1 = f1_score(y_true, y_pred, average=average, zero_division=0)
    cm = confusion_matrix(y_true, y_pred)

    print("=== Métriques de Classification ===")
    print(f"Accuracy:  {accuracy:.4f}")
    print(f"Precision: {precision:.4f}")
    print(f"Recall:    {recall:.4f}")
    print(f"F1-Score:  {f1:.4f}")
    print("\nMatrice de confusion:")
    print(cm)

    return {
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'F1-Score': f1,
        'Confusion Matrix': cm
    }

# ===== MÉTRIQUES DE CLUSTERING =====
def clustering_metrics(X, labels, centroids=None):
    """Calculer les métriques de clustering"""
    silhouette = silhouette_score(X, labels)
    davies_bouldin = davies_bouldin_score(X, labels)

    # Calcul de l'inertia si centro des fournis
    if centroids is not None:
        inertia = 0
        for i, centroid in enumerate(centroids):
            cluster_points = X[labels == i]
            if len(cluster_points) > 0:
                inertia += np.sum((cluster_points - centroid)**2)
    else:
        inertia = None

    print("=== Métriques de Clustering ===")
    print(f"Score de Silhouette: {silhouette:.4f}")
    print(f"Score Davies-Bouldin: {davies_bouldin:.4f}")
    if inertia is not None:
        print(f"Inertia: {inertia:.2f}")

    return {
        'Silhouette': silhouette,
        'Davies-Bouldin': davies_bouldin,
        'Inertia': inertia
    }

# Exemple d'utilisation
# Régression
y_true_reg = np.array([3, -0.5, 2, 7])
```

```
y_pred_reg = np.array([2.5, 0.0, 2, 8])
regression_metrics(y_true_reg, y_pred_reg)

# Classification
y_true_clf = np.array([0, 1, 2, 2, 0, 1])
y_pred_clf = np.array([0, 2, 2, 2, 0, 1])
classification_metrics(y_true_clf, y_pred_clf)

# Clustering
from sklearn.datasets import make_blobs
X_cluster, _ = make_blobs(n_samples=300, centers=4, random_state
    =42)
labels = KMeans(n_clusters=4, random_state=42).fit_predict(
    X_cluster)
clustering_metrics(X_cluster, labels)
```

Hyperparamètres et Paramètres

Résumé des Hyperparamètres par Modèle

	Prédit : Toxique	**Prédit : Comestible**
Réel : Toxique	TP (Vrai Positif)	FN (Faux Négatif)
Réel : Comestible	FP (Faux Positif)	TN (Vrai Négatif)

Paramètres Appris

Modèle	**Hyperparamètres**	**Description**
Régression Linéaire	`learning_rate` (α) `n_iterations`	Taux d'apprentissage pour gradient descent Nombre d'itérations
Régression Polynomiale	`degree`	Degré du polynôme

KNN	`k`
	`distance_metric`
	
	`weights`
		Nombre de voisins
		Métrique de distance (euclidienne, Manhattan)
		Poids uniformes ou basés sur distance

Arbre de Décision	`max_depth`
	`min_samples_split`
	`min_samples_leaf`
	`criterion`
	`max_features`
		Profondeur maximale
		Min échantillons pour diviser
		Min échantillons dans feuille
		Critère (gini, entropy)
		Max features à considérer

K-Means	`k` (n_clusters)
	`max_iter`
	`init`
	`n_init`
	`tol`
		Nombre de clusters
		Nombre max d'itérations
		Initialisation (random, k-means++)
		Nombre d'initialisations
		Tolérance pour convergence

TABLE 1 – Hyperparamètres principaux par modèle

Optimisation des Hyperparamètres

Exemple

```python
from sklearn.model_selection import GridSearchCV,
    RandomizedSearchCV
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.cluster import KMeans

# ===== Grid Search pour KNN =====
param_grid_knn = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan']
}

knn = KNeighborsClassifier()
grid_search_knn = GridSearchCV(
```

```
    knn, param_grid_knn, cv=5, scoring='accuracy', n_jobs=-1
)
grid_search_knn.fit(X_train, y_train)

print("Meilleurs hyperparamètres KNN:", grid_search_knn.
    best_params_)
print("Meilleur score:", grid_search_knn.best_score_)

# ===== Grid Search pour Arbre de Décision =====
param_grid_tree = {
    'max_depth': [3, 5, 7, 10, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy']
}

tree = DecisionTreeClassifier()
grid_search_tree = GridSearchCV(
    tree, param_grid_tree, cv=5, scoring='accuracy', n_jobs=-1
)
grid_search_tree.fit(X_train, y_train)

print("Meilleurs hyperparamètres Arbre:", grid_search_tree.
    best_params_)
print("Meilleur score:", grid_search_tree.best_score_)

# ===== Validation Croisée pour K-Means =====
from sklearn.model_selection import cross_val_score

k_range = range(2, 11)
silhouette_scores = []

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    labels = kmeans.fit_predict(X)
    score = silhouette_score(X, labels)
    silhouette_scores.append(score)

best_k = k_range[np.argmax(silhouette_scores)]
print(f"Meilleur k pour K-Means: {best_k}")
```

Résumé des Scripts Python Complets

Template Général d'Entraînement

Exemple

```python
# Template général pour l'entra nement d'un modèle ML

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import *

# 1. Chargement des données
data = pd.read_csv('dataset.csv')
X = data.drop('target', axis=1).values
y = data['target'].values

# 2. Préprocessing
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# 3. Division train/test
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42
)

# 4. Création et entra nement du modèle
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.cluster import KMeans

# Exemple pour régression
model = LinearRegression()
model.fit(X_train, y_train)

# 5. Prédictions
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# 6. Évaluation
print("Train MSE:", mean_squared_error(y_train, y_train_pred))
print("Test MSE:", mean_squared_error(y_test, y_test_pred))
print("Train R  :", r2_score(y_train, y_train_pred))
print("Test R  :", r2_score(y_test, y_test_pred))

# 7. Visualisation (optionnel)
import matplotlib.pyplot as plt
plt.scatter(y_test, y_test_pred)
plt.xlabel('Valeurs réelles')
plt.ylabel('Prédictions')
plt.title('Prédictions vs Réalité')
plt.show()
```

Checklist pour l'Examen

Important

Points à retenir pour l'examen :

1. **Formules** : Connaître toutes les formules de coût, gradients, métriques
2. **Gradient Descent** : Comprendre l'algorithme et les mises à jour
3. **Hyperparamètres** : Savoir identifier et ajuster les hyperparamètres
4. **Métriques** : Connaître les métriques appropriées pour chaque tâche
5. **Scripts Python** : Maîtriser les scripts d'entraînement et de prédiction
6. **Overfitting** : Comprendre et éviter le sur-apprentissage
7. **Préprocessing** : Normalisation, division train/test