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 donn es =====
# Chargement du dataset Salary_Data.csv
fichier_local = "Salary_Data.csv"
```

```
data = pd.read_csv(fichier_local)

# Aper u des donn es
print("Aper u :")
print(data.head())

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

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

# Variables
# X : YearsExperience (ann es d'exp rience)
# 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 ann es d'exp rience")
plt.xlabel("Ann es d'exp rience")
plt.ylabel("Salaire")
plt.grid(True)
plt.show()

# ===== PARTIE 3 : Pr paration des donn es (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 mod le =====
model = LinearRegression()
model.fit(X_train, y_train)

# Param tres appris
a = model.coef_[0]      # Pente
b = model.intercept_    # Ordonn e    l'origine

print("Pente (a) :", a)
print("Ordonn e    l'origine (b) :", b)

# Interpr tation :
# - b : Salaire de base pour 0 ann es d'exp rience
# - a : Augmentation de salaire par ann e d'exp rience

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

```

```
plt.figure(figsize=(7, 5))
plt.scatter(X_train, y_train, label='Donnees (train)')
plt.plot(X_train, y_pred_train, color='red', linewidth=2,
         label='Droite de regression')
plt.title("Ajustement du modele sur les donnees d'entrainement")
plt.xlabel("Annees d'experience")
plt.ylabel("Salaire")
plt.legend()
plt.grid(True)
plt.show()

# ===== PARTIE 6 : Evaluation 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("R²  :", r2)

# Interpretation R² :
# - R² proche de 1 : Le modele explique bien la variance
# - R² faible : Le modele explique mal la variance

# ===== PARTIE 7 : Prediction sur une nouvelle valeur =====
# Exemple : predire le salaire pour 8.5 annees d'experience
experience_nouvelle = [[8.5]]
salaire_prevu = model.predict(experience_nouvelle)
print(f"Salaire predit pour 8.5 annees d'experience : {
    salaire_prevu[0]:.2f}")

# ===== IMPLEMENTATION 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  # Ordonnee    l'origine
        self.cost_history = []

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

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

            # Calcul du cout : 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    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):
        """Pr diction : Y = a*X + b"""
        X = X.flatten()
        return self.a * X + self.b

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

print(f"\nMod le Gradient Descent :")
print(f"Pente (a) : {model_gd.a:.2f}")
print(f"Ordonn e    l'origine (b) : {model_gd.b:.2f}")

# Visualisation de l' volution  du co t
plt.figure(figsize=(10, 5))
plt.plot(model_gd.cost_history)
plt.xlabel('It rations')
plt.ylabel('Co t (MSE)')
plt.title(' volution  du Co t 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):
            # Pr dictions
            y_pred = X_bias.dot(self.weights)

            # Calcul du co t
            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    jour des poids
40              self.weights -= self.learning_rate * gradient
41
42          return self
43
44      def predict(self, X):
45          """
46          Pr diction
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 donn es
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  # Cr ation et entra nement
77  model = MultipleLinearRegression(learning_rate=0.01, n_iterations
        =1000)
78  model.fit(X_train, y_train)
79
80  # Pr dictions
81  y_train_pred = model.predict(X_train)
82  y_test_pred = model.predict(X_test)
83
84  # M triques
85  print("=== M triques d' valuation  ===")
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 probl matique

# Variable cible : MEDV (prix m dian des maisons)
# Variable explicative : LSTAT (pourcentage de statut inf rieur)

# ===== 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 : R gression lin aire simple (degr  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"R gression Lin aire (degr  1):")
print(f"MSE: {mse_linear:.2f}, R  : {r2_linear:.4f}")

# ===== PARTIE 4 : R gression polynomiale degr  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"\nR gression Polynomiale (degr  2):")
print(f"MSE: {mse_poly2:.2f}, R  : {r2_poly2:.4f}")

# ===== PARTIE 5 :  tude  du degr  du polyn me avec BIC =====
degrees = [1, 2, 3, 4, 5, 6]
results = []

for degree in degrees:
    # Cr ation 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)

    # Pr dictions
    y_pred_train = model.predict(X_poly_train)
    y_pred_test = model.predict(X_poly_test)

    # M triques
    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  # Degr  du polyn me
    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"\nDegr  {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}")

```

```
# Cr ation d'un DataFrame pour visualisation
df_results = pd.DataFrame(results)
print("\n=== Tableau R capitulatif ===")
print(df_results.to_string(index=False))

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

# Graphique 1 : BIC en fonction du degr
plt.subplot(1, 2, 1)
plt.plot(df_results['degree'], df_results['BIC'], 'o-', linewidth
    =2, markersize=8)
plt.xlabel('Degr  du polyn me (K)')
plt.ylabel('BIC')
plt.title('BIC en fonction du degr  du polyn me')
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 degr
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('Degr  du polyn me')
plt.ylabel('M trique')
plt.title('MSE et R  en fonction du degr ')
plt.legend()
plt.grid(True)
plt.xticks(degrees)

plt.tight_layout()
plt.show()

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

# ===== PARTIE 7 : Visualisation des mod les =====
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='Donn es', 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'Degr  {degree} (BIC={df_results.loc[idx, "BIC
                "]:.1f})')

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

# Questions d'analyse :
# 1. Comment  voluent  MSE et R  lorsque le degr  augmente ?
#        R  augmente toujours (sur train), mais peut diminuer sur
    test (overfitting)
# 2. Pourquoi le R  augmente toujours ?
#        Plus de param tres = meilleur ajustement aux donn es d'
    entra nement
# 3. Que signifie le param tre K dans la formule du BIC ?
#        K = degr  du polyn me (nombre de param tres)
# 4. Quel est le degr  donnant le plus petit BIC ?
#        C'est le meilleur compromis entre qualit  et simplicit
```

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 m trique"""
        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("M trique non support e")
31
32      def fit(self, X, y):
33          """
34          Entra nement (KNN est un algorithme lazy, on stocke juste
                les donn es)
35          """
36          self.X_train = X
37          self.y_train = y
38          return self
39
40      def predict(self, X):
41          """
42          Pr diction
43          """
44          predictions = []
45          for x in X:
46              # Calculer les distances    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 pond r  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          Probabilit s de pr diction
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]

            # Probabilit s bas es sur les fr quences
            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 donn es
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 diff rents 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 mod le
best_k = 3
model = KNN(k=best_k, distance_metric='euclidean', weights='
    distance')
model.fit(X_train, y_train)

# Pr dictions
y_pred = model.predict(X_test)
y_proba = model.predict_proba(X_test)

# M triques
print("\n=== M triques d' valuation ===")
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 : Arbre de Décision

Concept

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

Mesures d'Impureté

Formule

Entropie :

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i) \tag{22}$$

Gini Impurity :

$$G(S) = 1 - \sum_{i=1}^{c} p_i^2 \tag{23}$$

Information Gain (Gain d'information) :

$$IG(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v) \tag{24}$$

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'impuret  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'impuret  selon le crit re"""
        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 r cursive de l'arbre"""
89          n_samples = len(y)
90
91          # Crit res d'arr t
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'am lioration
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 r cursive
        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):
        """Pr diction pour un   chantillon  """
        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):
        """Pr diction"""
        return np.array([self._predict_sample(x, self.tree) for x
            in X])

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

# Chargement des donn es
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 diff rents hyperparam tres
print("=== Test des Hyperparam tres ===")
```

```
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 mod le
best_model = DecisionTreeClassifier(
    max_depth=3,
    min_samples_split=5,
    criterion='gini',
    random_state=42
)
best_model.fit(X_train, y_train)

# Pr dictions
y_pred = best_model.predict(X_test)

# M triques
print("\n=== M triques d' valuation ===")
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 D cision")
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) :

$$J = \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \tag{25}$$

Centroïde d'un cluster :

$$\boldsymbol{\mu}_i = \frac{1}{|C_i|} \sum_{\mathbf{x} \in C_i} \mathbf{x} \tag{26}$$

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 al atoirement,
            # puis choisir les suivants avec probabilit proportionnelle
            #    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    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)

                # V rifier 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 r sultat
            if inertia < best_inertia:
                best_inertia = inertia
                best_centroids = centroids
                best_labels = labels

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

```
96          self.inertia = best_inertia
97
98          return self
99
100     def predict(self, X):
101         """Pr diction (assignation aux clusters)"""
102         return self._assign_clusters(X, self.centroids)
103
104 # Utilisation
105 import pandas as pd
106
107 # Chargement des donn es (exemple avec donn es de c r ales)
108 data = pd.read_csv('cereal.csv')
109 # S lectionner des features num riques
110 features = ['calories', 'protein', 'fat', 'sodium', 'fiber', 'carbo
        ', 'sugars']
111 X = data[features].values
112
113 # Normalisation (important pour K-means)
114 scaler = StandardScaler()
115 X_scaled = scaler.fit_transform(X)
116
117 # M thode du coude pour choisir k
118 inertias = []
119 silhouette_scores = []
120 k_range = range(2, 11)
121
122 for k in k_range:
123     model = KMeans(n_clusters=k, random_state=42, n_init=10)
124     model.fit(X_scaled)
125     inertias.append(model.inertia_)
126     silhouette_scores.append(silhouette_score(X_scaled, model.
            labels_))
127
128 # Visualisation de la m thode du coude
129 fig, axes = plt.subplots(1, 2, figsize=(15, 5))
130
131 axes[0].plot(k_range, inertias, 'bo-')
132 axes[0].set_xlabel('Nombre de clusters (k)')
133 axes[0].set_ylabel('Inertia')
134 axes[0].set_title('M thode du Coude')
135 axes[0].grid(True)
136
137 axes[1].plot(k_range, silhouette_scores, 'ro-')
138 axes[1].set_xlabel('Nombre de clusters (k)')
139 axes[1].set_ylabel('Score de Silhouette')
140 axes[1].set_title('Score de Silhouette')
141 axes[1].grid(True)
142
143 plt.tight_layout()
144 plt.show()
145
146 # Meilleur k (bas  sur le score de silhouette)
147 best_k = k_range[np.argmax(silhouette_scores)]
148 print(f"Meilleur k selon silhouette: {best_k}")
149
```

```
# 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)

# M triques
print(f"\n=== M triques d' valuation ===")
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('Premi re composante principale' if X_scaled.shape[1] > 2 else 'Feature 1')
plt.ylabel('Deuxi me 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) :

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{27}$$

Root Mean Squared Error (RMSE) :

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2} \tag{28}$$

Mean Absolute Error (MAE) :

$$\text{MAE} = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i| \tag{29}$$

Coefficient de Détermination (R^2) :

$$R^2 = 1 - \frac{\text{SS}_{res}}{\text{SS}_{tot}} = 1 - \frac{\text{Erreur du modèle}}{\text{Variation totale}} \tag{30}$$

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}) :$(\text{40k-60k})^2 + (50k - 60k)^2 + (90k - 60k)^2 = 1,400,000,000$
- Modèle : Salaire = 25k * Expérience + 15k
- Erreur du modèle (SS_{res}) :$(\text{40k-40k})^2 + (50k - 65k)^2 + (90k - 90k)^2 = 225,000,000$
- $R^2 = 1 - (225,000,000/1,400,000,000) = 0.84$ (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) :

$$ACCURACY = \frac{\text{Prédictions correctes}}{\text{Total des prédictions}} = \frac{TP + TN}{TP + TN + FP + FN} \tag{31}$$

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

$$PRECISION = \frac{TP}{TP + FP} \tag{32}$$

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) :

$$RECALL = \frac{TP}{TP + FN} \tag{33}$$

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 :

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{34}$$

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) :

$$\text{Inertia} = \sum_{i=1}^{k} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \tag{35}$$

Score de Silhouette :

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \tag{36}$$

où :

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

Score Davies-Bouldin :

$$DB = \frac{1}{k} \sum_{i=1}^{k} \max_{j \neq i} \left(\frac{\sigma_i + \sigma_j}{d(\boldsymbol{\mu}_i, \boldsymbol{\mu}_j)} \right) \tag{37}$$

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
)

# ===== M TRIQUES DE R GRESSION =====
def regression_metrics(y_true, y_pred):
    """Calculer toutes les m triques de r gression"""
    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("=== M triques de R gression ===")
    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}

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

```
    """Calculer toutes les m triques 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("=== M triques 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
    }

# ===== M TRIQUES DE CLUSTERING =====
def clustering_metrics(X, labels, centroids=None):
    """Calculer les m triques 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("=== M triques 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
# R gression
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 hyperparam tres KNN:", grid_search_knn.
    best_params_)
print("Meilleur score:", grid_search_knn.best_score_)

# ===== Grid Search pour Arbre de D cision =====
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 hyperparam tres Arbre:", grid_search_tree.
    best_params_)
print("Meilleur score:", grid_search_tree.best_score_)

# ===== Validation Crois e 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 g n ral pour l'entra nement d'un mod le 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 donn es
data = pd.read_csv('dataset.csv')
X = data.drop('target', axis=1).values
y = data['target'].values

# 2. Pr processing
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. Cr ation et entra nement du mod le
from sklearn.linear_model import LinearRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.cluster import KMeans

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

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

# 6.  valuation
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 r elles')
plt.ylabel('Pr dictions')
plt.title('Pr dictions vs R alit  ')
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