

Résumé Complet - Machine Learning

Formules, Algorithmes et Scripts Python

AmineGR03

26 janvier 2026

Table des matières

1	Introduction	3
1.1	Paramètres vs Hyperparamètres	3
1.2	Types d'Apprentissage	3
2	Régression Linéaire Simple	4
2.1	Modèle Mathématique	4
2.2	Fonction de Coût (MSE)	4
2.3	Gradient Descent (Descente de Gradient)	5
2.4	Solution Analytique (Équations Normales)	5
2.5	Script Python : Entraînement et Prédiction (TP)	5
3	Régression Linéaire Multiple	9
3.1	Modèle Mathématique	9
3.2	Fonction de Coût	9
3.3	Gradient Descent	9
3.4	Solution Analytique	10
3.5	Script Python : Entraînement et Prédiction	10
4	Régression Polynomiale	13
4.1	Concept	13
4.2	BIC (Bayesian Information Criterion)	13
4.3	Script Python : Entraînement et Prédiction avec BIC (TP)	14
5	Classification : K-Nearest Neighbors (KNN)	18
5.1	Algorithme	18
5.2	Formules	18
5.3	Hyperparamètres	18
5.4	Optimisation du K (Grid Search)	19
5.5	Script Python : Entraînement et Prédiction	19
6	Classification : Arbre de Décision	22
6.1	Concept	22
6.2	Mesures d'Impureté	22
6.3	Algorithme ID3 (récuratif)	22
6.4	Hyperparamètres	22
6.5	Overfitting et Élagage	23
6.6	Script Python : Entraînement et Prédiction	24

7 Clustering : K-Means	28
7.1 Algorithme	28
7.2 Formules	28
7.3 Algorithme K-Means	28
7.4 Hyperparamètres	28
7.5 Méthode du Coude (Elbow Method)	29
7.6 Script Python : Entraînement et Prédiction	29
8 Métriques d'Évaluation	33
8.1 Régression	33
8.2 Classification	34
8.3 Clustering	35
8.4 Script Python : Calcul des Métriques	35
9 Hyperparamètres et Paramètres	38
9.1 Résumé des Hyperparamètres par Modèle	38
9.2 Paramètres Appris	38
9.3 Optimisation des Hyperparamètres	38
10 Résumé des Scripts Python Complets	40
10.1 Template Général d'Entraînement	40
10.2 Checklist pour l'Examen	41

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 :

$$\hat{Y} = aX + b \quad (1)$$

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 (coefficients) - 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 :

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

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 :

$$A_i = A_{i-1} - \alpha \frac{\partial J}{\partial A_{i-1}} \quad (3)$$

$$B_i = B_{i-1} - \alpha \frac{\partial J}{\partial B_{i-1}} \quad (4)$$

Dérivées partielles :

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

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

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) :

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

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

Script Python : Entraînement et Prédiction (TP)

Exemple

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from sklearn.model_selection import train_test_split
5 from sklearn.linear_model import LinearRegression
6 from sklearn.metrics import mean_squared_error, r2_score
7
8 # ===== PARTIE 1 : Chargement des données =====
9 # Chargement du dataset Salary_Data.csv
10 fichier_local = "Salary_Data.csv"

```

```

11 data = pd.read_csv(fichier_local)
12
13 # Aperçu des données
14 print("Aperçu :")
15 print(data.head())
16
17 # Informations
18 print("\nInfos :")
19 print(data.info())
20
21 # Statistiques descriptives
22 print("\nDescribe :")
23 print(data.describe())
24
25 # Variables
26 # X : YearsExperience (années d'expérience)
27 # Y : Salary (salaire annuel)
28
29 # ===== PARTIE 2 : Visualisation initiale =====
30 plt.figure(figsize=(7, 5))
31 plt.scatter(data['YearsExperience'], data['Salary'])
32 plt.title("Salaire en fonction des années d'expérience")
33 plt.xlabel("Années d'expérience")
34 plt.ylabel("Salaire")
35 plt.grid(True)
36 plt.show()
37
38 # ===== PARTIE 3 : Préparation des données (train/test) =====
39 X = data[['YearsExperience']] # variable explicative
40 y = data['Salary'] # variable cible
41
42 X_train, X_test, y_train, y_test = train_test_split(
43     X, y, test_size=0.2, random_state=42
44 )
45
46 print("Taille train :", X_train.shape, " | Taille test :", X_test.
        shape)
47
48 # ===== PARTIE 4 : Entraînement du modèle =====
49 model = LinearRegression()
50 model.fit(X_train, y_train)
51
52 # Paramètres appris
53 a = model.coef_[0] # Pente
54 b = model.intercept_ # Ordonnée l'origine
55
56 print("Pente (a) :", a)
57 print("Ordonnée l'origine (b) :", b)
58
59 # Interprétation :
60 # - b : Salaire de base pour 0 années d'expérience
61 # - a : Augmentation de salaire par année d'expérience
62
63 # ===== PARTIE 5 : Visualisation de la droite de régression =====
64 y_pred_train = model.predict(X_train)
65

```

```

66 plt.figure(figsize=(7, 5))
67 plt.scatter(X_train, y_train, label='Données (train)')
68 plt.plot(X_train, y_pred_train, color='red', linewidth=2,
69           label='Droite de régression')
70 plt.title("Ajustement du modèle sur les données d'entraînement")
71 plt.xlabel("Années d'expérience")
72 plt.ylabel("Salaire")
73 plt.legend()
74 plt.grid(True)
75 plt.show()
76
77 # ===== PARTIE 6 : valuation sur l'ensemble de test =====
78 y_pred_test = model.predict(X_test)
79
80 mse = mean_squared_error(y_test, y_pred_test)
81 r2 = r2_score(y_test, y_pred_test)
82
83 print("MSE :", mse)
84 print("R :", r2)
85
86 # Interprétation R :
87 # - R proche de 1 : Le modèle explique bien la variance
88 # - R faible : Le modèle explique mal la variance
89
90 # ===== PARTIE 7 : Prédiction sur une nouvelle valeur =====
91 # Exemple : prédire le salaire pour 8.5 années d'expérience
92 experience_nouvelle = [[8.5]]
93 salaire_prevu = model.predict(experience_nouvelle)
94 print(f"Salaire prédit pour 8.5 années d'expérience : {salaire_prevu[0]:.2f}")
95
96 # ===== IMPLEMENTATION MANUELLE AVEC GRADIENT DESCENT =====
97 class LinearRegressionGD:
98     def __init__(self, learning_rate=0.01, n_iterations=1000):
99         self.learning_rate = learning_rate # (alpha)
100        self.n_iterations = n_iterations
101        self.a = 0 # Pente
102        self.b = 0 # Ordonnée à l'origine
103        self.cost_history = []
104
105    def fit(self, X, y):
106        """
107            Entrainement avec gradient descent
108        """
109        n = len(X)
110        X = X.flatten() # Convertir en vecteur 1D
111
112        for iteration in range(self.n_iterations):
113            # Prédictions : Y = a*X + b
114            y_pred = self.a * X + self.b
115
116            # Calcul du coût : J(a, b) = (1/2n) * (y - (a*x + b))**2
117            cost = (1/(2*n)) * np.sum((y_pred - y)**2)
118            self.cost_history.append(cost)
119

```

```
120     # Calcul des gradients
121     dJ_da = -(1/n) * np.sum((y - y_pred) * X)
122     dJ_db = -(1/n) * np.sum(y - y_pred)
123
124     # Mise jour : A = A - * J / A
125     self.a -= self.learning_rate * dJ_da
126     self.b -= self.learning_rate * dJ_db
127
128     return self
129
130 def predict(self, X):
131     """Prédiction : Y = a*X + b"""
132     X = X.flatten()
133     return self.a * X + self.b
134
135 # Utilisation de l'implémentation manuelle
136 model_gd = LinearRegressionGD(learning_rate=0.01, n_iterations
137                               =1000)
138 model_gd.fit(X_train.values, y_train.values)
139
140 print(f"\nModèle Gradient Descent :")
141 print(f"Pente (a) : {model_gd.a:.2f}")
142 print(f"Ordonnée l'origine (b) : {model_gd.b:.2f}")
143
144 # Visualisation de l'évolution du coût
145 plt.figure(figsize=(10, 5))
146 plt.plot(model_gd.cost_history)
147 plt.xlabel('Iterations')
148 plt.ylabel('Coût (MSE)')
149 plt.title('Évolution du Coût avec Gradient Descent')
150 plt.grid(True)
151 plt.show()
```

Régression Linéaire Multiple

Modèle Mathématique

Formule

Équation de la régression linéaire multiple :

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

Exemple concret :

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

Forme vectorielle :

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

où :

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

Notation matricielle :

$$\mathbf{Y} = \mathbf{X}\mathbf{A} + B \quad (11)$$

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

Fonction de Coût

Formule

MSE pour régression multiple :

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

Gradient Descent

Formule

Mise à jour vectorielle :

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

Gradient :

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

Forme développée :

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

Solution Analytique

Formule

Équations normales :

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

Script Python : Entraînement et Prédiction

Exemple

```

1 import numpy as np
2 import pandas as pd
3 from sklearn.model_selection import train_test_split
4 from sklearn.preprocessing import StandardScaler
5 from sklearn.metrics import mean_squared_error, r2_score
6
7 class MultipleLinearRegression:
8     def __init__(self, learning_rate=0.01, n_iterations=1000):
9         self.learning_rate = learning_rate
10        self.n_iterations = n_iterations
11        self.weights = None
12        self.cost_history = []
13
14    def fit(self, X, y):
15        """
16            Entrainement avec gradient descent
17            X: matrice (m, n) - m exemples, n features
18            y: vecteur (m,)
19        """
20        m, n = X.shape
21
22        # Ajouter colonne de 1 pour le biais
23        X_bias = np.c_[np.ones(m), X]
24
25        # Initialisation des poids
26        self.weights = np.zeros(n + 1)
27
28        for iteration in range(self.n_iterations):
29            # Predictions
30            y_pred = X_bias.dot(self.weights)
31
32            # Calcul du co
33            cost = (1/(2*m)) * np.sum((y_pred - y)**2)
34            self.cost_history.append(cost)
35
36            # Calcul du gradient
37            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 entraînement
77 model = MultipleLinearRegression(learning_rate=0.01, n_iterations
78                                 =1000)
79 model.fit(X_train, y_train)
80
81 # Prédictions
82 y_train_pred = model.predict(X_train)
83 y_test_pred = model.predict(X_test)
84
85 # Métriques
86 print("==> Métriques d'évaluation ==>")
87 print(f'MSE Train: {mean_squared_error(y_train, y_train_pred):.2f}')
88 print(f'MSE Test: {mean_squared_error(y_test, y_test_pred):.2f}')
89 print(f'RMSE Train: {np.sqrt(mean_squared_error(y_train,
90                                         y_train_pred)):.2f}')
91 print(f'RMSE Test: {np.sqrt(mean_squared_error(y_test, y_test_pred))
92       :.2f}')

```

```
90| print(f"R    Train: {model.score(X_train, y_train):.4f}")  
91| print(f"R    Test: {model.score(X_test, y_test):.4f}")  
92|  
93| # Coefficients  
94| print("\n==== Coefficients ===")  
95| feature_names = data.drop('MEDV', axis=1).columns  
96| for i, (name, coef) in enumerate(zip(['Biais'] + list(feature_names),  
97|     model.weights)):  
98|     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 :

$$\hat{Y} = B + a_1X + a_2X^2 + \cdots + a_dX^d \quad (17)$$

Exemples :

- **Degré 1** : $\hat{Y} = B + a_1X$ (régression linéaire simple)
- **Degré 2** : $\hat{Y} = B + a_1X + a_2X^2$ (parabole)
- **Degré n** : $\hat{Y} = B + a_1X + a_2X^2 + \cdots + a_nX^n$

Transformation des features :

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

BIC (Bayesian Information Criterion)

Formule

Formule du BIC :

$$BIC = n \ln(MSE) + K \ln(n) \quad (19)$$

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

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from sklearn.preprocessing import PolynomialFeatures
5 from sklearn.linear_model import LinearRegression
6 from sklearn.metrics import mean_squared_error, r2_score
7 from sklearn.model_selection import train_test_split
8 from sklearn.datasets import fetch_openml
9
10 # ===== PARTIE 1 : Chargement du dataset Boston Housing =====
11 boston = fetch_openml(name='boston', version=1, as_frame=True)
12 data = boston.frame
13 data = data.drop('B', axis=1) # Supprimer colonne prob matique
14
15 # Variable cible : MEDV (prix median des maisons)
16 # Variable explicative : LSTAT (pourcentage de statut inférieur)
17
18 # ===== PARTIE 2 : Visualisation initiale =====
19 X = data[['LSTAT']].values
20 y = data['MEDV'].values
21
22 plt.figure(figsize=(8, 6))
23 plt.scatter(X, y, alpha=0.6)
24 plt.xlabel('LSTAT')
25 plt.ylabel('MEDV')
26 plt.title('Relation LSTAT - MEDV')
27 plt.grid(True)
28 plt.show()
29
30 # ===== PARTIE 3 : Rgression lin aire simple (degr 1) =====
31 X_train, X_test, y_train, y_test = train_test_split(
32     X, y, test_size=0.2, random_state=42
33 )
34
35 model_linear = LinearRegression()
36 model_linear.fit(X_train, y_train)
37
38 y_pred_linear = model_linear.predict(X_test)
39 mse_linear = mean_squared_error(y_test, y_pred_linear)
40 r2_linear = r2_score(y_test, y_pred_linear)
41
42 print(f"Rgression Lin aire (degr 1):")
43 print(f"MSE: {mse_linear:.2f}, R : {r2_linear:.4f}")
44
45 # ===== PARTIE 4 : Rgression polynomiale degr 2 =====
46 poly2 = PolynomialFeatures(degree=2)
47 X_poly2_train = poly2.fit_transform(X_train)

```

```

48 X_poly2_test = poly2.transform(X_test)
49
50 model_poly2 = LinearRegression()
51 model_poly2.fit(X_poly2_train, y_train)
52
53 y_pred_poly2 = model_poly2.predict(X_poly2_test)
54 mse_poly2 = mean_squared_error(y_test, y_pred_poly2)
55 r2_poly2 = r2_score(y_test, y_pred_poly2)
56
57 print(f"\nRégression Polynomiale (degr 2):")
58 print(f"  MSE: {mse_poly2:.2f}, R : {r2_poly2:.4f}")
59
60 # ===== PARTIE 5 : étude du degré du polynôme avec BIC =====
61 degrees = [1, 2, 3, 4, 5, 6]
62 results = []
63
64 for degree in degrees:
65     # Création des variables polynomiales
66     poly = PolynomialFeatures(degree=degree)
67     X_poly_train = poly.fit_transform(X_train)
68     X_poly_test = poly.transform(X_test)
69
70     # Entrainement
71     model = LinearRegression()
72     model.fit(X_poly_train, y_train)
73
74     # Prédictions
75     y_pred_train = model.predict(X_poly_train)
76     y_pred_test = model.predict(X_poly_test)
77
78     # Métriques
79     mse_train = mean_squared_error(y_train, y_pred_train)
80     mse_test = mean_squared_error(y_test, y_pred_test)
81     r2_train = r2_score(y_train, y_pred_train)
82     r2_test = r2_score(y_test, y_pred_test)
83
84     # Calcul du BIC :  $BIC = n * \ln(MSE) + K * \ln(n)$ 
85     n = len(y_test)
86     K = degree # Degré du polynôme
87     bic = n * np.log(mse_test) + K * np.log(n)
88
89     results.append({
90         'degree': degree,
91         'MSE_train': mse_train,
92         'MSE_test': mse_test,
93         'R2_train': r2_train,
94         'R2_test': r2_test,
95         'BIC': bic
96     })
97
98     print(f"\nDegr {degree}:")
99     print(f"  MSE Train: {mse_train:.2f}, MSE Test: {mse_test:.2f}")
100    )
101    print(f"  R Train: {r2_train:.4f}, R Test: {r2_test:.4f}")
102    print(f"  BIC: {bic:.2f}")

```

```

103 # Cr ation d'un DataFrame pour visualisation
104 df_results = pd.DataFrame(results)
105 print("\n==== Tableau R capitulatif ===")
106 print(df_results.to_string(index=False))
107
108 # ===== PARTIE 6 : Visualisation du BIC =====
109 plt.figure(figsize=(12, 5))
110
111 # Graphique 1 : BIC en fonction du degr
112 plt.subplot(1, 2, 1)
113 plt.plot(df_results['degree'], df_results['BIC'], 'o-', linewidth=2, markersize=8)
114 plt.xlabel('Degr du polyn me (K)')
115 plt.ylabel('BIC')
116 plt.title('BIC en fonction du degr du polyn me')
117 plt.grid(True)
118 plt.xticks(degrees)
119
120 # Identifier le minimum
121 min_bic_idx = df_results['BIC'].idxmin()
122 min_degree = df_results.loc[min_bic_idx, 'degree']
123 min_bic = df_results.loc[min_bic_idx, 'BIC']
124 plt.axvline(x=min_degree, color='r', linestyle='--',
125               label=f'Minimum (K={min_degree})')
126 plt.legend()
127
128 # Graphique 2 : MSE et R en fonction du degr
129 plt.subplot(1, 2, 2)
130 plt.plot(df_results['degree'], df_results['MSE_test'], 'o-', label='MSE Test', linewidth=2)
131 plt.plot(df_results['degree'], df_results['R2_test'], 's-', label='R Test', linewidth=2)
132 plt.xlabel('Degr du polyn me')
133 plt.ylabel('M trique')
134 plt.title('MSE et R en fonction du degr')
135 plt.legend()
136 plt.grid(True)
137 plt.xticks(degrees)
138
139 plt.tight_layout()
140 plt.show()
141
142 print(f"\n==== Meilleur mod le selon BIC ===")
143 print(f"Degr optimal: {min_degree}")
144 print(f"BIC minimum: {min_bic:.2f}")
145
146 # ===== PARTIE 7 : Visualisation des mod les =====
147 X_plot = np.linspace(X.min(), X.max(), 300).reshape(-1, 1)
148
149 plt.figure(figsize=(12, 8))
150 plt.scatter(X, y, alpha=0.5, label='Donn es', s=30)
151
152 colors = ['red', 'blue', 'green', 'orange', 'purple', 'brown']
153 for idx, degree in enumerate(degrees):
154     poly = PolynomialFeatures(degree=degree)
155     X_poly_plot = poly.fit_transform(X_plot)

```

```
158 model = LinearRegression()
159 X_poly_train = poly.fit_transform(X_train)
160 model.fit(X_poly_train, y_train)
161 y_plot = model.predict(X_poly_plot)
162
163 plt.plot(X_plot, y_plot, color=colors[idx], linewidth=2,
164         label=f'Degr {degree} (BIC={df_results.loc[idx, "BIC
165 "]:.1f})')
166
166 plt.xlabel('LSTAT')
167 plt.ylabel('MEDV')
168 plt.title('Comparaison des mod les polynomiales')
169 plt.legend()
170 plt.grid(True, alpha=0.3)
171 plt.show()
172
172 # Questions d'analyse :
173 # 1. Comment voluent MSE et R lorsque le degr augmente ?
174 #       R augmente toujours (sur train), mais peut diminuer sur
175 #       test (overfitting)
176 # 2. Pourquoi le R augmente toujours ?
177 #       Plus de param tres = meilleur ajustement aux donn es d'
178 #       entra nement
179 # 3. Que signifie le param tre K dans la formule du BIC ?
180 #       K = degr du polyn me (nombre de param tres)
181 # 4. Quel est le degr donnant le plus petit BIC ?
182 #       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)

Algorithm

Important

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

Formules

Formule

Distance euclidienne (exemple avec 2 features) :

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

Forme générale (n features) :

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

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 prédition
- Pour $K = 3$: On regarde les 3 plus proches voisins, la classe majoritaire est la prédition
- Pour $K = 5$: On regarde les 5 plus proches voisins, la classe majoritaire est la prédition

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

```

1 import numpy as np
2 from collections import Counter
3 from sklearn.model_selection import train_test_split
4 from sklearn.preprocessing import StandardScaler
5 from sklearn.metrics import accuracy_score, confusion_matrix,
   classification_report
6
7 class KNN:
8     def __init__(self, k=3, distance_metric='euclidean', weights='uniform'):
9         self.k = k
10        self.distance_metric = distance_metric
11        self.weights = weights
12        self.X_train = None
13        self.y_train = None
14
15    def _euclidean_distance(self, x1, x2):
16        """Distance euclidienne"""
17        return np.sqrt(np.sum((x1 - x2)**2))
18
19    def _manhattan_distance(self, x1, x2):
20        """Distance de Manhattan"""
21        return np.sum(np.abs(x1 - x2))
22
23    def _compute_distance(self, x1, x2):
24        """Calcul de la distance selon la m trique"""
25        if self.distance_metric == 'euclidean':
26            return self._euclidean_distance(x1, x2)
27        elif self.distance_metric == 'manhattan':
28            return self._manhattan_distance(x1, x2)

```

```

29         else:
30             raise ValueError("Méthode non supportée")
31
32     def fit(self, X, y):
33         """
34             Entrainement (KNN est un algorithme lazy, on stocke juste
35             les données)
36         """
37         self.X_train = X
38         self.y_train = y
39         return self
40
41     def predict(self, X):
42         """
43             Prédiction
44         """
45         predictions = []
46         for x in X:
47             # Calculer les distances tous les points d'
48             # entraînement
49             distances = [self._compute_distance(x, x_train)
50                         for x_train in self.X_train]
51
52             # Obtenir les k plus proches voisins
53             k_indices = np.argsort(distances)[:self.k]
54             k_nearest_labels = [self.y_train[i] for i in k_indices]
55
56             if self.weights == 'uniform':
57                 # Vote majoritaire simple
58                 most_common = Counter(k_nearest_labels).most_common
59                 (1)
60                 predictions.append(most_common[0][0])
61             else: # weights='distance'
62                 # Vote pondéré par l'inverse de la distance
63                 k_distances = [distances[i] for i in k_indices]
64                 weights = [1/d if d != 0 else 1e10 for d in
65                             k_distances]
66                 weighted_votes = {}
67                 for label, weight in zip(k_nearest_labels, weights):
68                     :
69                     weighted_votes[label] = weighted_votes.get(
70                         label, 0) + weight
71             predictions.append(max(weighted_votes, key=
72                         weighted_votes.get))
73
74         return np.array(predictions)
75
76     def predict_proba(self, X):
77         """
78             Probabilités de prédiction
79         """
80         probabilities = []
81         for x in X:
82             distances = [self._compute_distance(x, x_train)
83                         for x_train in self.X_train]
84             k_indices = np.argsort(distances)[:self.k]

```

```

78     k_nearest_labels = [self.y_train[i] for i in k_indices]
79
80     # Probabilités basées sur les fréquences
81     label_counts = Counter(k_nearest_labels)
82     total = sum(label_counts.values())
83     proba = {label: count/total for label, count in
84             label_counts.items()}
85     probabilities.append(proba)
86
87
88 # Utilisation
89 from sklearn.datasets import load_iris
90
91 # Chargement des données
92 iris = load_iris()
93 X, y = iris.data, iris.target
94
95 # Normalisation (important pour KNN)
96 scaler = StandardScaler()
97 X_scaled = scaler.fit_transform(X)
98
99 # Division train/test
100 X_train, X_test, y_train, y_test = train_test_split(
101     X_scaled, y, test_size=0.2, random_state=42
102 )
103
104 # Test avec différents k
105 for k in [1, 3, 5, 7, 10]:
106     model = KNN(k=k, distance_metric='euclidean', weights='uniform',
107                 )
108     model.fit(X_train, y_train)
109
110     y_pred = model.predict(X_test)
111     accuracy = accuracy_score(y_test, y_pred)
112
113     print(f"k={k:2d}: Accuracy = {accuracy:.4f}")
114
115 # Meilleur modèle
116 best_k = 3
117 model = KNN(k=best_k, distance_metric='euclidean', weights='
118             distance')
119 model.fit(X_train, y_train)
120
121 # Prédictions
122 y_pred = model.predict(X_test)
123 y_proba = model.predict_proba(X_test)
124
125 # Matrices
126 print("\n==== Matrices d'évaluation ====")
127 print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
128 print("\nMatrice de confusion:")
129 print(confusion_matrix(y_test, y_pred))
130 print("\nRapport de classification:")
131 print(classification_report(y_test, y_pred, target_names=iris.
132                             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 :

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

Gini Impurity :

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

Information Gain (Gain d'information) :

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

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 (récuratif)

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

```

1 import numpy as np
2 from collections import Counter
3 from sklearn.model_selection import train_test_split
4 from sklearn.metrics import accuracy_score, confusion_matrix,
   classification_report
5 import matplotlib.pyplot as plt
6 from sklearn.tree import plot_tree
7
8 class DecisionTree:
9     def __init__(self, max_depth=None, min_samples_split=2,
10                  min_samples_leaf=1, criterion='gini'):
11         self.max_depth = max_depth
12         self.min_samples_split = min_samples_split
13         self.min_samples_leaf = min_samples_leaf
14         self.criterion = criterion
15         self.tree = None
16
17     def _gini(self, y):
18         """Calcul de l'impureté Gini"""
19         if len(y) == 0:
20             return 0
21         counts = Counter(y)
22         proportions = [count/len(y) for count in counts.values()]
23         return 1 - sum(p**2 for p in proportions)
24
25     def _entropy(self, y):
26         """Calcul de l'entropie"""
27         if len(y) == 0:
28             return 0
29         counts = Counter(y)
30         proportions = [count/len(y) for count in counts.values()]
31         return -sum(p * np.log2(p) if p > 0 else 0 for p in
32                     proportions)
33
34     def _impurity(self, y):
35         """Calcul de l'impureté selon le critère"""
36         if self.criterion == 'gini':
37             return self._gini(y)
38         else: # entropy
39             return self._entropy(y)
40
41     def _information_gain(self, y_parent, y_left, y_right):
42         """Calcul du gain d'information"""
43         parent_impurity = self._impurity(y_parent)
44         n = len(y_parent)
45         n_left, n_right = len(y_left), len(y_right)
46
47         if n == 0:
48             return 0
49
50         child_impurity = (n_left/n) * self._impurity(y_left) + \
51                         (n_right/n) * self._impurity(y_right)
52
53         return parent_impurity - child_impurity
54
55     def fit(self, X, y):
56         self.tree = self._build_tree(X, y)
57
58     def _build_tree(self, X, y, depth=0):
59         if depth > self.max_depth or len(y) < self.min_samples_leaf:
60             return None
61
62         if len(np.unique(y)) == 1:
63             return Node(y[0])
64
65         best_feature, best_threshold = self._find_best_split(X, y)
66
67         if best_feature is None:
68             return Node(y[0])
69
70         left_index = X[:, best_feature] < best_threshold
71         right_index = X[:, best_feature] >= best_threshold
72
73         left_subtree = self._build_tree(X[left_index], y[left_index],
74                                         depth+1)
75         right_subtree = self._build_tree(X[right_index], y[right_index],
76                                         depth+1)
77
78         return Node(best_feature, best_threshold, left_subtree, right_subtree)
79
80     def predict(self, X):
81         if self.tree is None:
82             return np.array([None])
83
84         return self._predict_node(self.tree, X)
85
86     def _predict_node(self, node, X):
87         if node.is_leaf():
88             return np.array([node.value])
89
90         if X[node.feature] < node.threshold:
91             return self._predict_node(node.left, X)
92         else:
93             return self._predict_node(node.right, X)
94
95     def score(self, X, y):
96         y_pred = self.predict(X)
97         return accuracy_score(y, y_pred)
98
99     def confusion_matrix(self, X, y):
100        y_pred = self.predict(X)
101        return confusion_matrix(y, y_pred)
102
103    def classification_report(self, X, y):
104        y_pred = self.predict(X)
105        return classification_report(y, y_pred)
106
107    def plot(self, X, y):
108        plot_tree(self.tree, feature_names=X.columns, class_names=y)
109
110    def __str__(self):
111        return f"DecisionTree(max_depth={self.max_depth}, min_samples_split={self.min_samples_split}, min_samples_leaf={self.min_samples_leaf}, criterion='{self.criterion}')"
112
113    def __repr__(self):
114        return self.__str__()
115
116    def __eq__(self, other):
117        if not isinstance(other, DecisionTree):
118            return False
119
120        if self.max_depth != other.max_depth:
121            return False
122
123        if self.min_samples_split != other.min_samples_split:
124            return False
125
126        if self.min_samples_leaf != other.min_samples_leaf:
127            return False
128
129        if self.criterion != other.criterion:
130            return False
131
132        if self.tree != other.tree:
133            return False
134
135        return True
136
137    def __ne__(self, other):
138        return not self.__eq__(other)
139
140    def __hash__(self):
141        return hash((self.max_depth, self.min_samples_split, self.min_samples_leaf, self.criterion, self.tree))
142
143    def __lt__(self, other):
144        if not isinstance(other, DecisionTree):
145            return False
146
147        if self.max_depth < other.max_depth:
148            return True
149
150        if self.min_samples_split < other.min_samples_split:
151            return True
152
153        if self.min_samples_leaf < other.min_samples_leaf:
154            return True
155
156        if self.criterion < other.criterion:
157            return True
158
159        if self.tree < other.tree:
160            return True
161
162        return False
163
164    def __gt__(self, other):
165        if not isinstance(other, DecisionTree):
166            return False
167
168        if self.max_depth > other.max_depth:
169            return True
170
171        if self.min_samples_split > other.min_samples_split:
172            return True
173
174        if self.min_samples_leaf > other.min_samples_leaf:
175            return True
176
177        if self.criterion > other.criterion:
178            return True
179
180        if self.tree > other.tree:
181            return True
182
183        return False
184
185    def __le__(self, other):
186        if not isinstance(other, DecisionTree):
187            return False
188
189        if self.max_depth < other.max_depth:
190            return True
191
192        if self.min_samples_split < other.min_samples_split:
193            return True
194
195        if self.min_samples_leaf < other.min_samples_leaf:
196            return True
197
198        if self.criterion < other.criterion:
199            return True
200
201        if self.tree < other.tree:
202            return True
203
204        return True
205
206    def __ge__(self, other):
207        if not isinstance(other, DecisionTree):
208            return False
209
210        if self.max_depth > other.max_depth:
211            return True
212
213        if self.min_samples_split > other.min_samples_split:
214            return True
215
216        if self.min_samples_leaf > other.min_samples_leaf:
217            return True
218
219        if self.criterion > other.criterion:
220            return True
221
222        if self.tree > other.tree:
223            return True
224
225        return True
226
227    def __int__(self):
228        return int(self.max_depth)
229
230    def __float__(self):
231        return float(self.max_depth)
232
233    def __bool__(self):
234        return bool(self.max_depth)
235
236    def __complex__(self):
237        return complex(self.max_depth)
238
239    def __str__(self):
240        return str(self.max_depth)
241
242    def __repr__(self):
243        return repr(self.max_depth)
244
245    def __hash__(self):
246        return hash(self.max_depth)
247
248    def __lt__(self, other):
249        if not isinstance(other, DecisionTree):
250            return False
251
252        if self.max_depth < other.max_depth:
253            return True
254
255        if self.min_samples_split < other.min_samples_split:
256            return True
257
258        if self.min_samples_leaf < other.min_samples_leaf:
259            return True
260
261        if self.criterion < other.criterion:
262            return True
263
264        if self.tree < other.tree:
265            return True
266
267        return False
268
269    def __gt__(self, other):
270        if not isinstance(other, DecisionTree):
271            return False
272
273        if self.max_depth > other.max_depth:
274            return True
275
276        if self.min_samples_split > other.min_samples_split:
277            return True
278
279        if self.min_samples_leaf > other.min_samples_leaf:
280            return True
281
282        if self.criterion > other.criterion:
283            return True
284
285        if self.tree > other.tree:
286            return True
287
288        return False
289
290    def __le__(self, other):
291        if not isinstance(other, DecisionTree):
292            return False
293
294        if self.max_depth < other.max_depth:
295            return True
296
297        if self.min_samples_split < other.min_samples_split:
298            return True
299
300        if self.min_samples_leaf < other.min_samples_leaf:
301            return True
302
303        if self.criterion < other.criterion:
304            return True
305
306        if self.tree < other.tree:
307            return True
308
309        return True
310
311    def __ge__(self, other):
312        if not isinstance(other, DecisionTree):
313            return False
314
315        if self.max_depth > other.max_depth:
316            return True
317
318        if self.min_samples_split > other.min_samples_split:
319            return True
320
321        if self.min_samples_leaf > other.min_samples_leaf:
322            return True
323
324        if self.criterion > other.criterion:
325            return True
326
327        if self.tree > other.tree:
328            return True
329
330        return True
331
332    def __int__(self):
333        return int(self.max_depth)
334
335    def __float__(self):
336        return float(self.max_depth)
337
338    def __bool__(self):
339        return bool(self.max_depth)
340
341    def __complex__(self):
342        return complex(self.max_depth)
343
344    def __str__(self):
345        return str(self.max_depth)
346
347    def __repr__(self):
348        return repr(self.max_depth)
349
350    def __hash__(self):
351        return hash(self.max_depth)
352
353    def __lt__(self, other):
354        if not isinstance(other, DecisionTree):
355            return False
356
357        if self.max_depth < other.max_depth:
358            return True
359
360        if self.min_samples_split < other.min_samples_split:
361            return True
362
363        if self.min_samples_leaf < other.min_samples_leaf:
364            return True
365
366        if self.criterion < other.criterion:
367            return True
368
369        if self.tree < other.tree:
370            return True
371
372        return False
373
374    def __gt__(self, other):
375        if not isinstance(other, DecisionTree):
376            return False
377
378        if self.max_depth > other.max_depth:
379            return True
380
381        if self.min_samples_split > other.min_samples_split:
382            return True
383
384        if self.min_samples_leaf > other.min_samples_leaf:
385            return True
386
387        if self.criterion > other.criterion:
388            return True
389
390        if self.tree > other.tree:
391            return True
392
393        return False
394
395    def __le__(self, other):
396        if not isinstance(other, DecisionTree):
397            return False
398
399        if self.max_depth < other.max_depth:
400            return True
401
402        if self.min_samples_split < other.min_samples_split:
403            return True
404
405        if self.min_samples_leaf < other.min_samples_leaf:
406            return True
407
408        if self.criterion < other.criterion:
409            return True
410
411        if self.tree < other.tree:
412            return True
413
414        return True
415
416    def __ge__(self, other):
417        if not isinstance(other, DecisionTree):
418            return False
419
420        if self.max_depth > other.max_depth:
421            return True
422
423        if self.min_samples_split > other.min_samples_split:
424            return True
425
426        if self.min_samples_leaf > other.min_samples_leaf:
427            return True
428
429        if self.criterion > other.criterion:
430            return True
431
432        if self.tree > other.tree:
433            return True
434
435        return True
436
437    def __int__(self):
438        return int(self.max_depth)
439
440    def __float__(self):
441        return float(self.max_depth)
442
443    def __bool__(self):
444        return bool(self.max_depth)
445
446    def __complex__(self):
447        return complex(self.max_depth)
448
449    def __str__(self):
450        return str(self.max_depth)
451
452    def __repr__(self):
453        return repr(self.max_depth)
454
455    def __hash__(self):
456        return hash(self.max_depth)
457
458    def __lt__(self, other):
459        if not isinstance(other, DecisionTree):
460            return False
461
462        if self.max_depth < other.max_depth:
463            return True
464
465        if self.min_samples_split < other.min_samples_split:
466            return True
467
468        if self.min_samples_leaf < other.min_samples_leaf:
469            return True
470
471        if self.criterion < other.criterion:
472            return True
473
474        if self.tree < other.tree:
475            return True
476
477        return False
478
479    def __gt__(self, other):
480        if not isinstance(other, DecisionTree):
481            return False
482
483        if self.max_depth > other.max_depth:
484            return True
485
486        if self.min_samples_split > other.min_samples_split:
487            return True
488
489        if self.min_samples_leaf > other.min_samples_leaf:
490            return True
491
492        if self.criterion > other.criterion:
493            return True
494
495        if self.tree > other.tree:
496            return True
497
498        return False
499
500    def __le__(self, other):
501        if not isinstance(other, DecisionTree):
502            return False
503
504        if self.max_depth < other.max_depth:
505            return True
506
507        if self.min_samples_split < other.min_samples_split:
508            return True
509
510        if self.min_samples_leaf < other.min_samples_leaf:
511            return True
512
513        if self.criterion < other.criterion:
514            return True
515
516        if self.tree < other.tree:
517            return True
518
519        return True
520
521    def __ge__(self, other):
522        if not isinstance(other, DecisionTree):
523            return False
524
525        if self.max_depth > other.max_depth:
526            return True
527
528        if self.min_samples_split > other.min_samples_split:
529            return True
530
531        if self.min_samples_leaf > other.min_samples_leaf:
532            return True
533
534        if self.criterion > other.criterion:
535            return True
536
537        if self.tree > other.tree:
538            return True
539
540        return True
541
542    def __int__(self):
543        return int(self.max_depth)
544
545    def __float__(self):
546        return float(self.max_depth)
547
548    def __bool__(self):
549        return bool(self.max_depth)
550
551    def __complex__(self):
552        return complex(self.max_depth)
553
554    def __str__(self):
555        return str(self.max_depth)
556
557    def __repr__(self):
558        return repr(self.max_depth)
559
560    def __hash__(self):
561        return hash(self.max_depth)
562
563    def __lt__(self, other):
564        if not isinstance(other, DecisionTree):
565            return False
566
567        if self.max_depth < other.max_depth:
568            return True
569
570        if self.min_samples_split < other.min_samples_split:
571            return True
572
573        if self.min_samples_leaf < other.min_samples_leaf:
574            return True
575
576        if self.criterion < other.criterion:
577            return True
578
579        if self.tree < other.tree:
580            return True
581
582        return False
583
584    def __gt__(self, other):
585        if not isinstance(other, DecisionTree):
586            return False
587
588        if self.max_depth > other.max_depth:
589            return True
590
591        if self.min_samples_split > other.min_samples_split:
592            return True
593
594        if self.min_samples_leaf > other.min_samples_leaf:
595            return True
596
597        if self.criterion > other.criterion:
598            return True
599
600        if self.tree > other.tree:
601            return True
602
603        return False
604
605    def __le__(self, other):
606        if not isinstance(other, DecisionTree):
607            return False
608
609        if self.max_depth < other.max_depth:
610            return True
611
612        if self.min_samples_split < other.min_samples_split:
613            return True
614
615        if self.min_samples_leaf < other.min_samples_leaf:
616            return True
617
618        if self.criterion < other.criterion:
619            return True
620
621        if self.tree < other.tree:
622            return True
623
624        return True
625
626    def __ge__(self, other):
627        if not isinstance(other, DecisionTree):
628            return False
629
630        if self.max_depth > other.max_depth:
631            return True
632
633        if self.min_samples_split > other.min_samples_split:
634            return True
635
636        if self.min_samples_leaf > other.min_samples_leaf:
637            return True
638
639        if self.criterion > other.criterion:
640            return True
641
642        if self.tree > other.tree:
643            return True
644
645        return True
646
647    def __int__(self):
648        return int(self.max_depth)
649
650    def __float__(self):
651        return float(self.max_depth)
652
653    def __bool__(self):
654        return bool(self.max_depth)
655
656    def __complex__(self):
657        return complex(self.max_depth)
658
659    def __str__(self):
660        return str(self.max_depth)
661
662    def __repr__(self):
663        return repr(self.max_depth)
664
665    def __hash__(self):
666        return hash(self.max_depth)
667
668    def __lt__(self, other):
669        if not isinstance(other, DecisionTree):
670            return False
671
672        if self.max_depth < other.max_depth:
673            return True
674
675        if self.min_samples_split < other.min_samples_split:
676            return True
677
678        if self.min_samples_leaf < other.min_samples_leaf:
679            return True
680
681        if self.criterion < other.criterion:
682            return True
683
684        if self.tree < other.tree:
685            return True
686
687        return False
688
689    def __gt__(self, other):
690        if not isinstance(other, DecisionTree):
691            return False
692
693        if self.max_depth > other.max_depth:
694            return True
695
696        if self.min_samples_split > other.min_samples_split:
697            return True
698
699        if self.min_samples_leaf > other.min_samples_leaf:
700            return True
701
702        if self.criterion > other.criterion:
703            return True
704
705        if self.tree > other.tree:
706            return True
707
708        return False
709
710    def __le__(self, other):
711        if not isinstance(other, DecisionTree):
712            return False
713
714        if self.max_depth < other.max_depth:
715            return True
716
717        if self.min_samples_split < other.min_samples_split:
718            return True
719
720        if self.min_samples_leaf < other.min_samples_leaf:
721            return True
722
723        if self.criterion < other.criterion:
724            return True
725
726        if self.tree < other.tree:
727            return True
728
729        return True
730
731    def __ge__(self, other):
732        if not isinstance(other, DecisionTree):
733            return False
734
735        if self.max_depth > other.max_depth:
736            return True
737
738        if self.min_samples_split > other.min_samples_split:
739            return True
740
741        if self.min_samples_leaf > other.min_samples_leaf:
742            return True
743
744        if self.criterion > other.criterion:
745            return True
746
747        if self.tree > other.tree:
748            return True
749
750        return True
751
752    def __int__(self):
753        return int(self.max_depth)
754
755    def __float__(self):
756        return float(self.max_depth)
757
758    def __bool__(self):
759        return bool(self.max_depth)
760
761    def __complex__(self):
762        return complex(self.max_depth)
763
764    def __str__(self):
765        return str(self.max_depth)
766
767    def __repr__(self):
768        return repr(self.max_depth)
769
770    def __hash__(self):
771        return hash(self.max_depth)
772
773    def __lt__(self, other):
774        if not isinstance(other, DecisionTree):
775            return False
776
777        if self.max_depth < other.max_depth:
778            return True
779
780        if self.min_samples_split < other.min_samples_split:
781            return True
782
783        if self.min_samples_leaf < other.min_samples_leaf:
784            return True
785
786        if self.criterion < other.criterion:
787            return True
788
789        if self.tree < other.tree:
790            return True
791
792        return False
793
794    def __gt__(self, other):
795        if not isinstance(other, DecisionTree):
796            return False
797
798        if self.max_depth > other.max_depth:
800            return True
801
802        if self.min_samples_split > other.min_samples_split:
803            return True
804
805        if self.min_samples_leaf > other.min_samples_leaf:
806            return True
807
808        if self.criterion > other.criterion:
809            return True
810
811        if self.tree > other.tree:
812            return True
813
814        return False
815
816    def __le__(self, other):
817        if not isinstance(other, DecisionTree):
818            return False
819
820        if self.max_depth < other.max_depth:
821            return True
822
823        if self.min_samples_split < other.min_samples_split:
824            return True
825
826        if self.min_samples_leaf < other.min_samples_leaf:
827            return True
828
829        if self.criterion < other.criterion:
830            return True
831
832        if self.tree < other.tree:
833            return True
834
835        return True
836
837    def __ge__(self, other):
838        if not isinstance(other, DecisionTree):
839            return False
840
841        if self.max_depth > other.max_depth:
842            return True
843
844        if self.min_samples_split > other.min_samples_split:
845            return True
846
847        if self.min_samples_leaf > other.min_samples_leaf:
848            return True
849
850        if self.criterion > other.criterion:
851            return True
852
853        if self.tree > other.tree:
854            return True
855
856        return True
857
858    def __int__(self):
859        return int(self.max_depth)
860
861    def __float__(self):
862        return float(self.max_depth)
863
864    def __bool__(self):
865        return bool(self.max_depth)
866
867    def __complex__(self):
868        return complex(self.max_depth)
869
870    def __str__(self):
871        return str(self.max_depth)
872
873    def __repr__(self):
874        return repr(self.max_depth)
875
876    def __hash__(self):
877        return hash(self.max_depth)
878
879    def __lt__(self, other):
880        if not isinstance(other, DecisionTree):
881            return False
882
883        if self.max_depth < other.max_depth:
884            return True
885
886        if self.min_samples_split < other.min_samples_split:
887            return True
888
889        if self.min_samples_leaf < other.min_samples_leaf:
890            return True
891
892        if self.criterion < other.criterion:
893            return True
894
895        if self.tree < other.tree:
896            return True
897
898        return False
899
900    def __gt__(self, other):
901        if not isinstance(other, DecisionTree):
902            return False
903
904        if self.max_depth > other.max_depth:
906            return True
907
908        if self.min_samples_split > other.min_samples_split:
909            return True
910
911        if self.min_samples_leaf > other.min_samples_leaf:
912            return True
913
914        if self.criterion > other.criterion:
915            return True
916
917        if self.tree > other.tree:
918            return True
919
920        return False
921
922    def __le__(self, other):
923        if not isinstance(other, DecisionTree):
924            return False
925
926        if self.max_depth < other.max_depth:
927            return True
928
929        if self.min_samples_split < other.min_samples_split:
930            return True
931
932        if self.min_samples_leaf < other.min_samples_leaf:
933            return True
934
935        if self.criterion < other.criterion:
936            return True
937
938        if self.tree < other.tree:
939            return True
940
941        return True
942
943    def __ge__(self, other):
944        if not isinstance(other, DecisionTree):
945            return False
946
947        if self.max_depth > other.max_depth:
948            return True
949
950        if self.min_samples_split > other.min_samples_split:
951            return True
952
953        if self.min_samples_leaf > other.min_samples_leaf:
954            return True
955
956        if self.criterion > other.criterion:
957            return True
958
959        if self.tree > other.tree:
960            return True
961
962        return True
963
964    def __int__(self):
965        return int(self.max_depth)
966
967    def __float__(self):
968        return float(self.max_depth)
969
970    def __bool__(self):
971        return bool(self.max_depth)
972
973    def __complex__(self):
974        return complex(self.max_depth)
975
976    def __str__(self):
977        return str(self.max_depth)
978
979    def __repr__(self):
980        return repr(self.max_depth)
981
982    def __hash__(self):
983        return hash(self.max_depth)
984
985    def __lt__(self, other):
986        if not isinstance(other, DecisionTree):
987            return False
988
989        if self.max_depth < other.max_depth:
990            return True
991
992        if self.min_samples_split < other.min_samples_split:
993            return True
994
995        if self.min_samples_leaf < other.min_samples_leaf:
996            return True
997
998        if self.criterion < other.criterion:
999            return True
1000
1001        if self.tree < other.tree:
1002            return True
1003
1004        return False
1005
1006    def __gt__(self, other):
1007        if not isinstance(other, DecisionTree):
1008            return False
1009
1010        if self.max_depth > other.max_depth:
1012            return True
1013
1014        if self.min_samples_split > other.min_samples_split:
1015            return True
1016
1017        if self.min_samples_leaf > other.min_samples_leaf:
1018            return True
1019
1020        if self.criterion > other.criterion:
1021            return True
1022
1023        if self.tree > other.tree:
1024            return True
1025
1026        return False
1027
1028    def __le__(self, other):
1029        if not isinstance(other, DecisionTree):
1030            return False
1031
1032        if self.max_depth < other.max_depth:
1033            return True
1034
1035        if self.min_samples_split < other.min_samples_split:
1036            return True
1037
1038        if self.min_samples_leaf < other.min_samples_leaf:
1039            return True
1040
1041        if self.criterion < other.criterion:
1042            return True
1043
1044        if self.tree < other.tree:
1045            return True
1046
1047        return True
1048
1049    def __ge__(self, other):
1050        if not isinstance(other, DecisionTree):
1051            return False
1052
1053        if self.max_depth > other.max_depth:
1054            return True
1055
1056        if self.min_samples_split > other.min_samples_split:
1057            return True
1058
1059        if self.min_samples_leaf > other.min_samples_leaf:
1060            return True
1061
1062        if self.criterion > other.criterion:
1063            return True
1064
1065        if self.tree > other.tree:
1066            return True
1067
1068        return True
1069
1070    def __int__(self):
1071        return int(self.max_depth)
1072
1073    def __float__(self):
1074        return float(self.max_depth)
1075
1076    def __bool__(self):
1077        return bool(self.max_depth)
1078
1079    def __complex__(self):
1080        return complex(self.max_depth)
1081
1082    def __str__(self):
1083        return str(self.max_depth)
1084
1085    def __repr__(self):
1086        return repr(self.max_depth)
1087
1088    def __hash__(self):
1089        return hash(self.max_depth)
1090
1091    def __lt__(self, other):
1092        if not isinstance(other, DecisionTree):
1093            return False
1094
1095        if self.max_depth < other.max_depth:
1096            return True
1097
1098        if self.min_samples_split < other.min_samples_split:
1099            return True
1100
1101        if self.min_samples_leaf < other.min_samples_leaf:
1102            return True
1103
1104        if self.criterion < other.criterion:
1105            return True
1106
1107        if self.tree < other.tree:
1108            return True
1109
1110        return False
1111
1112    def __gt__(self, other):
1113        if not isinstance(other, DecisionTree):
1114            return False
1115
1116        if self.max_depth > other.max_depth:
1118            return True
1119
1120        if self.min_samples_split > other.min_samples_split:
1121            return True
1122
1123        if self.min_samples_leaf > other.min_samples_leaf:
1124            return True
1125
1126        if self.criterion > other.criterion:
1127            return True
1128
1129        if self.tree > other.tree:
1130            return True
1131
1132        return False
1133
1134    def __le__(self, other):
1135        if not isinstance(other, DecisionTree):
1136            return False
1137
1138        if self.max_depth < other.max_depth:
1139            return True
1140
1141        if self.min_samples_split < other.min_samples_split:
1142            return True
1143
1144        if self.min_samples_leaf < other.min_samples_leaf:
1145            return True
1146
1147        if self.criterion < other.criterion:
1148            return True
1149
1150        if self.tree < other.tree:
1151            return True
1152
1153        return True
1154
1155    def __ge__(self, other):
1156        if not isinstance(other, DecisionTree):
1157            return False
1158
1159        if self.max_depth > other.max_depth:
1160            return True
1161
1162        if self.min_samples_split > other.min_samples_split:
1163            return True
1164
1165        if self.min_samples_leaf > other.min_samples_leaf:
1166            return True
1167
1168        if self.criterion > other.criterion:
1169            return True
1170
1171        if self.tree > other.tree:
1172            return True
1173
1174        return True
1175
1176    def __int__(self):
1177        return int(self.max_depth)
1178
1179    def __float__(self):
1180        return float(self.max_depth)
1181
1182    def __bool__(self):
1183        return bool(self.max_depth)
1184
1185    def __complex__(self):
1186        return complex(self.max_depth)
1187
1188    def __str__(self):
1189        return str(self.max_depth)
1190
1191    def __repr__(self):
1192        return repr(self.max_depth)
1193
1194    def __hash__(self):
1195        return hash(self.max_depth)
1196
1197    def __lt__(self, other):
1198        if not isinstance(other, DecisionTree):
1199            return False
1200
1201        if self.max_depth < other.max_depth:
1202            return True
1203
1204        if self.min_samples_split < other.min_samples_split:
1205            return True
1206
1207        if self.min_samples_leaf < other.min_samples_leaf:
1208            return True
1209
1210        if self.criterion < other.criterion:
1211            return True
1212
1213        if self.tree < other.tree:
1214            return True
1215
1216        return False
1217
1218    def __gt__(self, other):
1219        if not isinstance(other, DecisionTree):
1220            return False
1221
1222        if self.max_depth > other.max_depth:
1224            return True
1225
1226        if self.min_samples_split > other.min_samples_split:
1227            return True
1228
1229        if self.min_samples_leaf > other.min_samples_leaf:
1230            return True
1231
1232        if self.criterion > other.criterion:
1233            return True
1234
1235        if self.tree > other.tree:
1236            return True
1237
1238        return False
1239
1240    def __le__(self, other):
1241        if not isinstance(other, DecisionTree):
1242            return False
1243
1244        if self.max_depth < other.max_depth:
1245            return True
1246
1247        if self.min_samples_split < other.min_samples_split:
1248            return True
1249
1250        if self.min_samples_leaf < other.min_samples_leaf:
1251            return True
1252
1253        if self.criterion < other.criterion:
1254            return True
1255
1256        if self.tree < other.tree:
1257            return True
1258
1259        return True
1260
1261    def __ge__(
```

```

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 rursive 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) \
93             or \
94             n_samples < self.min_samples_split or \
95             len(np.unique(y)) == 1:
96             return {'class': Counter(y).most_common(1)[0][0], 'leaf': True}
97
98         # Meilleur split
99         feature, threshold, gain = self._best_split(X, y)
100
101        if gain == 0: # Pas d'am lioration
102            return {'class': Counter(y).most_common(1)[0][0], 'leaf': True}
103
104        # Division

```

```

104     left_mask = X[:, feature] <= threshold
105     right_mask = ~left_mask
106
107     if np.sum(left_mask) < self.min_samples_leaf or \
108         np.sum(right_mask) < self.min_samples_leaf:
109         return {'class': Counter(y).most_common(1)[0][0], 'leaf':
110             True}
111
112     # Construction recursive
113     node = {
114         'feature': feature,
115         'threshold': threshold,
116         'left': self._build_tree(X[left_mask], y[left_mask],
117             depth + 1),
118         'right': self._build_tree(X[right_mask], y[right_mask],
119             depth + 1),
120         'leaf': False
121     }
122
123     return node
124
125
126
127     def fit(self, X, y):
128         """Entrainement"""
129         self.tree = self._build_tree(X, y)
130         return self
131
132
133     def _predict_sample(self, x, node):
134         """Prédiction pour un chantillon """
135         if node['leaf']:
136             return node['class']
137
138         if x[node['feature']] <= node['threshold']:
139             return self._predict_sample(x, node['left'])
140         else:
141             return self._predict_sample(x, node['right'])
142
143     def predict(self, X):
144         """Prédiction"""
145         return np.array([self._predict_sample(x, self.tree) for x
146                         in X])
147
148
149     # Utilisation avec sklearn (version optimisée)
150     from sklearn.tree import DecisionTreeClassifier
151     from sklearn.datasets import load_iris
152
153     # Chargement des données
154     iris = load_iris()
155     X, y = iris.data, iris.target
156
157     # Division train/test
158     X_train, X_test, y_train, y_test = train_test_split(
159         X, y, test_size=0.2, random_state=42
160     )
161
162     # Test avec différents hyperparamètres
163     print("== Test des Hyperparamètres ==")

```

```

156 for max_depth in [None, 3, 5, 10]:
157     for min_samples_split in [2, 5, 10]:
158         model = DecisionTreeClassifier(
159             max_depth=max_depth,
160             min_samples_split=min_samples_split,
161             criterion='gini',
162             random_state=42
163         )
164         model.fit(X_train, y_train)
165         train_acc = accuracy_score(y_train, model.predict(X_train))
166         test_acc = accuracy_score(y_test, model.predict(X_test))
167         print(f"max_depth={max_depth}, min_samples_split={min_samples_split}: "
168               f"Train={train_acc:.4f}, Test={test_acc:.4f}")
169
170 # Meilleur modèle
171 best_model = DecisionTreeClassifier(
172     max_depth=3,
173     min_samples_split=5,
174     criterion='gini',
175     random_state=42
176 )
177 best_model.fit(X_train, y_train)
178
179 # Prédictions
180 y_pred = best_model.predict(X_test)
181
182 # Matrices de confusion
183 print("\n--- Matrices de confusion ---")
184 print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
185 print("\nMatrice de confusion:")
186 print(confusion_matrix(y_test, y_pred))
187 print("\nRapport de classification:")
188 print(classification_report(y_test, y_pred, target_names=iris.target_names))
189
190 # Visualisation de l'arbre
191 plt.figure(figsize=(20, 10))
192 plot_tree(best_model, feature_names=iris.feature_names,
193            class_names=iris.target_names, filled=True)
194 plt.title("Arbre de Classification")
195 plt.show()

```

Clustering : K-Means

Algorithm

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) :

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

Centroïde d'un cluster :

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

où :

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

Algorithm K-Means

Algorithm 2 K-Means

Require: Données \mathbf{X} , nombre de clusters k

Ensure: Centroïdes $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k$

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)

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

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.cluster import KMeans
4 from sklearn.preprocessing import StandardScaler
5 from sklearn.metrics import silhouette_score, davies_bouldin_score
6
7 class KMeansClustering:
8     def __init__(self, k=3, max_iter=300, init='k-means++', n_init=10):
9         self.k = k
10        self.max_iter = max_iter
11        self.init = init
12        self.n_init = n_init
13        self.centroids = None
14        self.labels = None
15        self.inertia = None
16
17    def _initialize_centroids(self, X):
18        """Initialisation des centro des"""
19        if self.init == 'random':
20            indices = np.random.choice(len(X), self.k, replace=False)
21            return X[indices]
22        elif self.init == 'k-means++':
23            # K-means++ : choisir le premier centro de
24            # al atoirement,
25            # puis choisir les suivants avec probabilit
26            # proportionnelle
27            # la distance au centro de le plus proche
28            centroids = [X[np.random.randint(len(X))]]
29
30            for _ in range(self.k - 1):
31                distances = np.array([
32                    min([np.linalg.norm(x - c)**2 for c in
33                        centroids])
34                    for x in X
35                ])
36                probabilities = distances / distances.sum()
37                cumulative_probs = probabilities.cumsum()
38                r = np.random.rand()
39                idx = np.searchsorted(cumulative_probs, r)
40                centroids.append(X[idx])
41
42            return np.array(centroids)
43
44    def _assign_clusters(self, X, centroids):
45        pass
```

```

42     """Assigner chaque point au cluster le plus proche"""
43     distances = np.sqrt(((X - centroids[:, np.newaxis])**2).sum(
44         axis=2))
45     return np.argmin(distances, axis=0)
46
47     def _update_centroids(self, X, labels):
48         """Mettre jour les centro des"""
49         centroids = np.array([X[labels == i].mean(axis=0)
50                               for i in range(self.k)])
51     return centroids
52
53     def _compute_inertia(self, X, labels, centroids):
54         """Calculer l'inertia"""
55         inertia = 0
56         for i in range(self.k):
57             cluster_points = X[labels == i]
58             if len(cluster_points) > 0:
59                 inertia += np.sum((cluster_points - centroids[i])
60                                   **2)
61     return inertia
62
63     def fit(self, X):
64         """Entrainement"""
65         best_inertia = float('inf')
66         best_centroids = None
67         best_labels = None
68
69         # Essayer plusieurs initialisations
70         for init_idx in range(self.n_init):
71             # Initialisation
72             centroids = self._initialize_centroids(X)
73
74             for iteration in range(self.max_iter):
75                 # Assignment
76                 labels = self._assign_clusters(X, centroids)
77
78                 # Nouveaux centro des
79                 new_centroids = self._update_centroids(X, labels)
80
81                 # V rifier la convergence
82                 if np.allclose(centroids, new_centroids):
83                     break
84
85                 centroids = new_centroids
86
87                 # Calculer l'inertia
88                 inertia = self._compute_inertia(X, labels, centroids)
89
90                 # Garder le meilleur r sultat
91                 if inertia < best_inertia:
92                     best_inertia = inertia
93                     best_centroids = centroids
94                     best_labels = labels
95
96         self.centroids = best_centroids
97         self.labels = best_labels

```

```

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

```

```

150 # Entrainement avec le meilleur k
151 model = KMeans(n_clusters=best_k, random_state=42, n_init=10)
152 model.fit(X_scaled)
153 labels = model.predict(X_scaled)
154
155 # M triques
156 print(f"\n==== M triques d' valuation ===")
157 print(f"Nombre de clusters: {best_k}")
158 print(f"Inertia: {model.inertia_:.2f}")
159 print(f"Score de Silhouette: {silhouette_score(X_scaled, labels):.4f}")
160 print(f"Score Davies-Bouldin: {davies_bouldin_score(X_scaled, labels):.4f}")
161
162 # Visualisation (si 2D ou avec PCA)
163 from sklearn.decomposition import PCA
164
165 if X_scaled.shape[1] > 2:
166     pca = PCA(n_components=2)
167     X_2d = pca.fit_transform(X_scaled)
168 else:
169     X_2d = X_scaled
170
171 plt.figure(figsize=(10, 8))
172 scatter = plt.scatter(X_2d[:, 0], X_2d[:, 1], c=labels, cmap='viridis', alpha=0.6)
173 plt.scatter(model.cluster_centers_[:, 0] if X_scaled.shape[1] == 2
174             else pca.transform(model.cluster_centers_)[:, 0],
175             model.cluster_centers_[:, 1] if X_scaled.shape[1] == 2
176             else pca.transform(model.cluster_centers_)[:, 1],
177             c='red', marker='x', s=200, linewidths=3, label='Centre des')
178 plt.colorbar(scatter)
179 plt.xlabel('Première composante principale' if X_scaled.shape[1] > 2
180             else 'Feature 1')
180 plt.ylabel('Deuxième composante principale' if X_scaled.shape[1] >
181             2 else 'Feature 2')
181 plt.title(f'Clustering K-Means (k={best_k})')
182 plt.legend()
183 plt.grid(True)
184 plt.show()

```

Métriques d'Évaluation

Régression

Formule

Mean Squared Error (MSE) :

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

Root Mean Squared Error (RMSE) :

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

Mean Absolute Error (MAE) :

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

Coefficient de Détermination (R^2) :

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

où :

- SS_{res} (*Somme des carrés des résidus*) : Erreur du modèle
- SS_{tot} (*Somme des carrés totaux*) : Variation totale où $= 1_{n \sum_{i=1}^n y_i}$ 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) :

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

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

Precision (Précision) :

$$PRECISION = \frac{TP}{TP + FP} \quad (32)$$

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) :

$$RECALL = \frac{TP}{TP + FN} \quad (33)$$

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 :

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (34)$$

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) :

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

Score de Silhouette :

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

où :

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

Score Davies-Bouldin :

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

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

Script Python : Calcul des Métriques

Exemple

```

1 import numpy as np
2 from sklearn.metrics import (
3     mean_squared_error, mean_absolute_error, r2_score,
4     accuracy_score, precision_score, recall_score, f1_score,
5     confusion_matrix, classification_report,
6     silhouette_score, davies_bouldin_score
7 )
8
9 # ===== MÉTRIQUES DE RÉGRESSION =====
10 def regression_metrics(y_true, y_pred):
11     """Calculer toutes les métriques de régression"""
12     mse = mean_squared_error(y_true, y_pred)
13     rmse = np.sqrt(mse)
14     mae = mean_absolute_error(y_true, y_pred)
15     r2 = r2_score(y_true, y_pred)
16
17     print("==== Métriques de Régression ===")
18     print(f"MSE: {mse:.4f}")
19     print(f"RMSE: {rmse:.4f}")
20     print(f"MAE: {mae:.4f}")
21     print(f"R² : {r2:.4f}")
22
23     return {'MSE': mse, 'RMSE': rmse, 'MAE': mae, 'R²': r2}
24
25 # ===== MÉTRIQUES DE CLASSIFICATION =====
26 def classification_metrics(y_true, y_pred, average='weighted'):

```

```

27     """Calculer toutes les métriques de classification"""
28     accuracy = accuracy_score(y_true, y_pred)
29     precision = precision_score(y_true, y_pred, average=average,
30         zero_division=0)
31     recall = recall_score(y_true, y_pred, average=average,
32         zero_division=0)
33     f1 = f1_score(y_true, y_pred, average=average, zero_division=0)
34     cm = confusion_matrix(y_true, y_pred)
35
36     print("==== Métriques de Classification ====")
37     print(f"Accuracy: {accuracy:.4f}")
38     print(f"Precision: {precision:.4f}")
39     print(f"Recall: {recall:.4f}")
40     print(f"F1-Score: {f1:.4f}")
41     print("\nMatrice de confusion:")
42     print(cm)
43
44     return {
45         'Accuracy': accuracy,
46         'Precision': precision,
47         'Recall': recall,
48         'F1-Score': f1,
49         'Confusion Matrix': cm
50     }
51
52 # ===== METRIQUES DE CLUSTERING =====
53 def clustering_metrics(X, labels, centroids=None):
54     """Calculer les métriques de clustering"""
55     silhouette = silhouette_score(X, labels)
56     davies_bouldin = davies_bouldin_score(X, labels)
57
58     # Calcul de l'inertia si centroïdes fournis
59     if centroids is not None:
60         inertia = 0
61         for i, centroid in enumerate(centroids):
62             cluster_points = X[labels == i]
63             if len(cluster_points) > 0:
64                 inertia += np.sum((cluster_points - centroid)**2)
65     else:
66         inertia = None
67
68     print("==== Métriques de Clustering ====")
69     print(f"Score de Silhouette: {silhouette:.4f}")
70     print(f"Score Davies-Bouldin: {davies_bouldin:.4f}")
71     if inertia is not None:
72         print(f"Inertia: {inertia:.2f}")
73
74     return {
75         'Silhouette': silhouette,
76         'Davies-Bouldin': davies_bouldin,
77         'Inertia': inertia
78     }
79
80 # Exemple d'utilisation
81 # Rgression
82 y_true_reg = np.array([3, -0.5, 2, 7])

```

```
81 y_pred_reg = np.array([2.5, 0.0, 2, 8])
82 regression_metrics(y_true_reg, y_pred_reg)
83
84 # Classification
85 y_true_clf = np.array([0, 1, 2, 2, 0, 1])
86 y_pred_clf = np.array([0, 2, 2, 2, 0, 1])
87 classification_metrics(y_true_clf, y_pred_clf)
88
89 # Clustering
90 from sklearn.datasets import make_blobs
91 X_cluster, _ = make_blobs(n_samples=300, centers=4, random_state
   =42)
92 labels = KMeans(n_clusters=4, random_state=42).fit_predict(
   X_cluster)
93 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	<code>learning_rate (α)</code> <code>n_iterations</code>	Taux d'apprentissage pour gradient descent Nombre d'itérations
Régression Polynomiale	<code>degree</code>	Degré du polynôme
KNN	<code>k</code> <code>distance_metric</code> <code>weights</code>	Nombre de voisins Métrique de distance (euclidienne, Manhattan) Poids uniformes ou basés sur distance
Arbre de Décision	<code>max_depth</code> <code>min_samples_split</code> <code>min_samples_leaf</code> <code>criterion</code> <code>max_features</code>	Profondeur maximale Min échantillons pour diviser Min échantillons dans feuille Critère (gini, entropy) Max features à considérer
K-Means	<code>k (n_clusters)</code> <code>max_iter</code> <code>init</code> <code>n_init</code> <code>tol</code>	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

```

1 from sklearn.model_selection import GridSearchCV,
2     RandomizedSearchCV
3 from sklearn.linear_model import LinearRegression
4 from sklearn.neighbors import KNeighborsClassifier
5 from sklearn.tree import DecisionTreeClassifier
6 from sklearn.cluster import KMeans
7
8 # ===== Grid Search pour KNN =====
9 param_grid_knn = {
10     'n_neighbors': [3, 5, 7, 9, 11],
11     'weights': ['uniform', 'distance'],
12     'metric': ['euclidean', 'manhattan']
13 }
14 knn = KNeighborsClassifier()
15 grid_search_knn = GridSearchCV(

```

```

16     knn, param_grid_knn, cv=5, scoring='accuracy', n_jobs=-1
17 )
18 grid_search_knn.fit(X_train, y_train)
19
20 print("Meilleurs hyperparam tres KNN:", grid_search_knn.
21       best_params_)
22 print("Meilleur score:", grid_search_knn.best_score_)
23
24 # ===== Grid Search pour Arbre de D cision =====
25 param_grid_tree = {
26     'max_depth': [3, 5, 7, 10, None],
27     'min_samples_split': [2, 5, 10],
28     'min_samples_leaf': [1, 2, 4],
29     'criterion': ['gini', 'entropy']
30 }
31
32 tree = DecisionTreeClassifier()
33 grid_search_tree = GridSearchCV(
34     tree, param_grid_tree, cv=5, scoring='accuracy', n_jobs=-1
35 )
36 grid_search_tree.fit(X_train, y_train)
37
38 print("Meilleurs hyperparam tres Arbre:", grid_search_tree.
39       best_params_)
40 print("Meilleur score:", grid_search_tree.best_score_)
41
42 # ===== Validation Crois e pour K-Means =====
43 from sklearn.model_selection import cross_val_score
44
45 k_range = range(2, 11)
46 silhouette_scores = []
47
48 for k in k_range:
49     kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
50     labels = kmeans.fit_predict(X)
51     score = silhouette_score(X, labels)
52     silhouette_scores.append(score)
53
54 best_k = k_range[np.argmax(silhouette_scores)]
55 print(f"Meilleur k pour K-Means: {best_k}")

```

Résumé des Scripts Python Complets

Template Général d'Entraînement

Exemple

```

1 # Template général pour l'entraînement d'un modèle ML
2
3 import numpy as np
4 import pandas as pd
5 from sklearn.model_selection import train_test_split
6 from sklearn.preprocessing import StandardScaler
7 from sklearn.metrics import *
8
9 # 1. Chargement des données
10 data = pd.read_csv('dataset.csv')
11 X = data.drop('target', axis=1).values
12 y = data['target'].values
13
14 # 2. Processing
15 scaler = StandardScaler()
16 X_scaled = scaler.fit_transform(X)
17
18 # 3. Division train/test
19 X_train, X_test, y_train, y_test = train_test_split(
20     X_scaled, y, test_size=0.2, random_state=42
21 )
22
23 # 4. Cr ation et entra nement du mod le
24 from sklearn.linear_model import LinearRegression
25 from sklearn.neighbors import KNeighborsClassifier
26 from sklearn.tree import DecisionTreeClassifier
27 from sklearn.cluster import KMeans
28
29 # Exemple pour r gression
30 model = LinearRegression()
31 model.fit(X_train, y_train)
32
33 # 5. Pr dictions
34 y_train_pred = model.predict(X_train)
35 y_test_pred = model.predict(X_test)
36
37 # 6. valuation
38 print("Train MSE:", mean_squared_error(y_train, y_train_pred))
39 print("Test MSE:", mean_squared_error(y_test, y_test_pred))
40 print("Train R :", r2_score(y_train, y_train_pred))
41 print("Test R :", r2_score(y_test, y_test_pred))
42
43 # 7. Visualisation (optionnel)
44 import matplotlib.pyplot as plt
45 plt.scatter(y_test, y_test_pred)
46 plt.xlabel('Valeurs réelles')
47 plt.ylabel('Pr dictions')
48 plt.title('Pr dictions vs Réalit e')
49 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