

Résumé Complet - Machine Learning
Formules, Algorithmes et Scripts Python

AmineGR03

26 janvier 2026

Table des matières

Introduction

Paramètres vs Hyperparamètres

Important

Définitions :

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

Types d'Apprentissage

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

Régression Linéaire Simple

Modèle Mathématique

Formule

Équation de la régression linéaire simple :

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

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("== Mètres 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 trique 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 : Support Vector Machine (SVM)

Introduction

Important

Qu'est-ce qu'un SVM ?

Le SVM est un algorithme d'apprentissage supervisé utilisé principalement pour :

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

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

Pourquoi les SVM sont populaires ?

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

SVM Linéaire : Principe

Formule

Hyperplan de séparation :

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

Frontières de marge :

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

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

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

Règle de décision :

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

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

Calcul de la Marge

Formule

Calcul de la distance entre les deux hyperplans :

Soit deux points support vectors :

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

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

Calcul de r :

$$\mathbf{W}^T \cdot (\mathbf{x}^- + r\mathbf{w}) + B = +1 \quad (26)$$

$$r\|\mathbf{w}\|^2 + \mathbf{W}^T \cdot \mathbf{x}^- + B = +1 \quad (27)$$

$$r\|\mathbf{w}\|^2 - 1 = +1 \quad (28)$$

$$r\|\mathbf{w}\|^2 = 2 \quad (29)$$

$$r = \frac{2}{\|\mathbf{w}\|^2} \quad (30)$$

Marge : La distance entre les deux hyperplans est $r = \frac{2}{\|\mathbf{w}\|^2}$

Objectif : Maximiser la marge \Rightarrow Minimiser $\|\mathbf{w}\|^2$

SVM Hard Margin (Marge Rigide)

Formule

Problème primal :

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

Conditions :

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

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

Optimisation avec Multiplicateurs de Lagrange

Formule

Lagrangien :

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

où $\varphi_i \geq 0$ sont les multiplicateurs de Lagrange.

Conditions KKT (Karush-Kuhn-Tucker) :

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_i \varphi_i y_i \mathbf{x}_i \quad (33)$$

$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_i \varphi_i y_i = 0 \quad (34)$$

$$\varphi_i[y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1] = 0 \quad (\text{condition de complémentarité}) \quad (35)$$

Interprétation :

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

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

Parcimonie du modèle SVM :

- Dataset : 10,000 points d'entraînement
- Support vectors : 50-200 points seulement (1-2%)
- Points avec $\varphi_i = 0$: 9,800 points (98%)
- Impact : Modèle final très compact, prédiction rapide

Problème Dual

Formule

Formulation duale :

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

Sous les contraintes :

$$\alpha_i \geq 0 \quad \forall i \quad (37)$$

$$\sum_i \alpha_i y_i = 0 \quad (38)$$

Avantages du dual :

- Plus facile à résoudre numériquement
- Les données apparaissent uniquement via des produits scalaires $\langle \mathbf{x}_i, \mathbf{x}_j \rangle$
- Permet l'utilisation du kernel trick pour les cas non-linéaires

SVM Soft Margin (Marge Souple)

Important

Pourquoi le Hard Margin ne suffit pas ?

Le SVM Hard Margin suppose que les données sont parfaitement séparables :

- Aucun point ne doit être dans la marge
- Aucun point ne doit être mal classé

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

Formule

Slack variables (Variables de relâchement) :

On introduit $\xi_i \geq 0$ pour permettre à un point de violer la contrainte de marge.

Contrainte modifiée :

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

Interprétation de ξ_i :

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

- $0 < \xi_i < 1$: Point dans la marge mais bien classé
- $\xi_i \geq 1$: Point mal classé

Problème d'optimisation :

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

Sous les contraintes :

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

$$\xi_i \geq 0 \quad (42)$$

Paramètre C :

- C grand : Les erreurs coûtent cher \Rightarrow modèle très strict (marge étroite)
- C petit : Plus de flexibilité \Rightarrow modèle plus tolérant (marge large)
- C contrôle le compromis entre maximiser la marge et pénaliser les erreurs

Problème dual (Soft Margin) :

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

Sous les contraintes :

$$0 \leq \alpha_i \leq C \quad \forall i \quad (44)$$

$$\sum_i \alpha_i y_i = 0 \quad (45)$$

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

SVM Non-Linéaire : Kernel Trick

Important

Limitation du SVM linéaire :

Le SVM linéaire fonctionne parfaitement lorsque les classes sont séparables par une ligne droite (ou un hyperplan). Mais que faire lorsque les données ont une structure plus complexe (ex : cercles concentriques) ?

La solution : SVM non-linéaire

Grâce au kernel trick, nous transformons les données dans un espace de dimension supérieure où elles deviennent linéairement séparables, sans calculer explicitement cette transformation.

Formule

Qu'est-ce qu'un noyau (kernel) ?

Un noyau est une fonction de similarité entre deux points. Dans un SVM, cette similarité est calculée sous forme de produit scalaire.

Kernel linéaire :

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

Transformation dans un espace de dimension supérieure :

$$\mathbf{x} \rightarrow \phi(\mathbf{x}) \quad (47)$$

Dans ce nouvel espace, la similarité devient :

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

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

Formulation duale avec kernel :

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

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

Kernels Courants

Formule

1. Kernel Linéaire :

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

Quand l'utiliser :

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

2. Kernel Polynomial :

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

où :

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

Quand l'utiliser :

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

Exemple (degré 2) :

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

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

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

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

Quand l'utiliser :

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

Interprétation de γ :

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

Exemple de calcul RBF :

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

Script Python : SVM Linéaire (TP)

Exemple

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from sklearn.svm import SVC
5 from sklearn.model_selection import train_test_split
6 from sklearn.preprocessing import StandardScaler
7 from sklearn.metrics import accuracy_score, confusion_matrix,
   classification_report
8
9 # ===== PARTIE 1 : Chargement des données =====
10 # Fichier : iris_svm_lineaire_pret.csv
11 data = pd.read_csv('iris_svm_lineaire_pret.csv')
12
13 # Aperçu
14 print("Aperçu des données :")
15 print(data.head())
16 print(f"\nNombre d'observations : {len(data)}")
17
18 # ===== PARTIE 2 : Préparation des données =====
19 X = data[['feature1', 'feature2']].values # Features
20 y = data['class'].values # Classe cible
21
22 # Séparation train/test

```

```

23 X_train, X_test, y_train, y_test = train_test_split(
24     X, y, test_size=0.2, random_state=42
25 )
26
27 # Normalisation (recommand pour SVM)
28 scaler = StandardScaler()
29 X_train_scaled = scaler.fit_transform(X_train)
30 X_test_scaled = scaler.transform(X_test)
31
32 # ===== PARTIE 3 : Entrainement d'un SVM Lin aire =====
33 # Creer un mod le SVM lin aire avec C = 1
34 svm_linear = SVC(kernel='linear', C=1.0, random_state=42)
35 svm_linear.fit(X_train_scaled, y_train)
36
37 # Predictions
38 y_pred_train = svm_linear.predict(X_train_scaled)
39 y_pred_test = svm_linear.predict(X_test_scaled)
40
41 # M triques
42 print("\n==== M triques SVM Lin aire ===")
43 print(f"Accuracy Train: {accuracy_score(y_train, y_pred_train):.4f}")
44 print(f"Accuracy Test: {accuracy_score(y_test, y_pred_test):.4f}")
45
46 # Informations sur les support vectors
47 print(f"\nNombre de vecteurs de support: {len(svm_linear.support_)}")
48 print(f"Support vectors indices: {svm_linear.support_}")
49 print(f"Support vectors: \n{svm_linear.support_vectors_}")
50
51 # Coefficients
52 print(f"\nCoefficients (w): {svm_linear.coef_[0]}")
53 print(f"Biais (b): {svm_linear.intercept_[0]}")
54
55 # ===== PARTIE 4 : Visualisation =====
56 def plot_svm_decision_boundary(X, y, model, title):
57     """Visualiser l'hyperplan, les marges et les support vectors"""
58     plt.figure(figsize=(10, 8))
59
60     # Nuage de points
61     plt.scatter(X[y == 0, 0], X[y == 0, 1], c='red', marker='o',
62                 label='Classe 0', s=50, alpha=0.6)
63     plt.scatter(X[y == 1, 0], X[y == 1, 1], c='blue', marker='s',
64                 label='Classe 1', s=50, alpha=0.6)
65
66     # Support vectors
67     plt.scatter(model.support_vectors_[:, 0],
68                 model.support_vectors_[:, 1],
69                 s=200, facecolors='none', edgecolors='black',
70                 linewidth=2, label='Support Vectors')
71
72     # Frontiere de d cision
73     ax = plt.gca()
74     xlim = ax.get_xlim()
75     ylim = ax.get_ylim()
76

```

```

77 # Cr er une grille pour la fronti re
78 xx = np.linspace(xlim[0], xlim[1], 100)
79 yy = np.linspace(ylim[0], ylim[1], 100)
80 YY, XX = np.meshgrid(yy, xx)
81 xy = np.vstack([XX.ravel(), YY.ravel()]).T
82 Z = model.decision_function(xy).reshape(XX.shape)

83 # Contour de la fronti re de d cision
84 ax.contour(XX, YY, Z, colors='black', levels=[-1, 0, 1],
85             alpha=0.5, linestyles=['--', '--', '--'], linewidths
86             =2)

87 plt.xlabel('Feature 1')
88 plt.ylabel('Feature 2')
89 plt.title(title)
90 plt.legend()
91 plt.grid(True, alpha=0.3)
92 plt.show()

93 # Visualisation
94 plot_svm_decision_boundary(X_train_scaled, y_train, svm_linear,
95                             'SVM Lin aire - Hyperplan et Marges')

96 # ===== PARTIE 5 : Test avec diff rentes valeurs de C =====
97 C_values = [0.1, 1, 10, 100]
98 results = []

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

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

105     results.append({
106         'C': C,
107         'Accuracy': acc,
108         'N_Support_Vectors': n_sv
109     })

110     print(f"C={C:6.1f}: Accuracy={acc:.4f}, Support Vectors={n_sv}")
111

112 df_results = pd.DataFrame(results)
113 print("\n==== R sultats selon C ===")
114 print(df_results.to_string(index=False))

115 # Questions d'analyse :
116 # 1. Quand C est grand, la marge devient-elle large ou stricte ?
117 #      C grand : marge stricte ( troite ), moins d'erreurs
118 #      tol r es
119 # 2. Combien de support vectors y a-t-il pour chaque C ?
120 #      C grand : plus de support vectors (mod le plus strict)

```

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

Exemple

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from sklearn.svm import SVC
5 from sklearn.model_selection import train_test_split
6 from sklearn.preprocessing import StandardScaler
7 from sklearn.metrics import accuracy_score, confusion_matrix
8
9 # ===== PARTIE 1 : Chargement des données =====
10 # Fichier : breast_cancer_svm_rbf_2d_ready.csv
11 data = pd.read_csv('breast_cancer_svm_rbf_2d_ready.csv')
12
13 print("Aperçu des données :")
14 print(data.head())
15 print(f"\nNombre d'observations : {len(data)}")
16
17 # ===== PARTIE 2 : Préparation des données =====
18 X = data[['feature1', 'feature2']].values
19 y = data['target'].values
20
21 # Visualisation initiale
22 plt.figure(figsize=(8, 6))
23 plt.scatter(X[y == 0, 0], X[y == 0, 1], c='red', marker='o',
24             label='Classe 0', s=50, alpha=0.6)
25 plt.scatter(X[y == 1, 0], X[y == 1, 1], c='blue', marker='s',
26             label='Classe 1', s=50, alpha=0.6)
27 plt.xlabel('Feature 1')
28 plt.ylabel('Feature 2')
29 plt.title('Données - Structure Non-Linéaire')
30 plt.legend()
31 plt.grid(True, alpha=0.3)
32 plt.show()
33
34 # Séparation train/test
35 X_train, X_test, y_train, y_test = train_test_split(
36     X, y, test_size=0.2, random_state=42
37 )
38
39 # Normalisation
40 scaler = StandardScaler()
41 X_train_scaled = scaler.fit_transform(X_train)
42 X_test_scaled = scaler.transform(X_test)
43
44 # ===== PARTIE 3 : SVM Linéaire (modèle de référence) =====
45 svm_linear = SVC(kernel='linear', C=1.0, random_state=42)
46 svm_linear.fit(X_train_scaled, y_train)
47
48 y_pred_linear = svm_linear.predict(X_test_scaled)
49 acc_linear = accuracy_score(y_test, y_pred_linear)
50
51 print(f"\n==== SVM Linéaire ===")
52 print(f"Accuracy: {acc_linear:.4f}")
53 print(f"Support Vectors: {len(svm_linear.support_)}")
```

```

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

```

```

109         label='Classe 1', s=50, alpha=0.6)
110
111     # Support vectors
112     plt.scatter(model.support_vectors_[:, 0],
113                  model.support_vectors_[:, 1],
114                  s=200, facecolors='none', edgecolors='black',
115                  linewidth=2, label='Support Vectors')
116
117     # Frontière de décision
118     ax = plt.gca()
119     xlim = ax.get_xlim()
120     ylim = ax.get_ylim()
121
122     xx = np.linspace(xlim[0], xlim[1], 100)
123     yy = np.linspace(ylim[0], ylim[1], 100)
124     YY, XX = np.meshgrid(yy, xx)
125     xy = np.vstack([XX.ravel(), YY.ravel()]).T
126     Z = model.decision_function(xy).reshape(XX.shape)
127
128     # Contour de la frontière
129     ax.contour(XX, YY, Z, colors='black', levels=[0],
130                alpha=0.8, linestyles=['-'], linewidths=2)
131     ax.contourf(XX, YY, Z, levels=[-np.inf, 0, np.inf],
132                 alpha=0.3, colors=['red', 'blue'])
133
134     plt.xlabel('Feature 1')
135     plt.ylabel('Feature 2')
136     plt.title(title)
137     plt.legend()
138     plt.grid(True, alpha=0.3)
139     plt.show()
140
141 # Visualisation SVM RBF
142 svm_rbf_best = SVC(kernel='rbf', C=best_params['C'],
143                      gamma=best_params['gamma'], random_state=42)
144 svm_rbf_best.fit(X_train_scaled, y_train)
145
146 plot_svm_rbf_decision_boundary(X_train_scaled, y_train,
147                                 f'SVM RBF (C={best_params["C"]},'
148                                 f'gamma={best_params["gamma"]})')
149
150 # ===== PARTIE 7 : Comparaison Lin aire vs RBF =====
151 fig, axes = plt.subplots(1, 2, figsize=(16, 6))
152
153 # SVM Lin aire
154 ax = axes[0]
155 ax.scatter(X_train_scaled[y_train == 0, 0], X_train_scaled[y_train
156 == 0, 1],
157             c='red', marker='o', label='Classe 0', s=50, alpha=0.6)
158 ax.scatter(X_train_scaled[y_train == 1, 0], X_train_scaled[y_train
159 == 1, 1],
160             c='blue', marker='s', label='Classe 1', s=50, alpha=0.6)
161 ax.scatter(svm_linear.support_vectors_[:, 0], svm_linear.
162             support_vectors_[:, 1],
163             s=200, facecolors='none', edgecolors='black', linewidth

```

```

        =2)
160 xlim = ax.get_xlim()
161 ylim = ax.get_ylim()
162 xx = np.linspace(xlim[0], xlim[1], 100)
163 yy = np.linspace(ylim[0], ylim[1], 100)
164 YY, XX = np.meshgrid(yy, xx)
165 xy = np.vstack([XX.ravel(), YY.ravel()]).T
166 Z = svm_linear.decision_function(xy).reshape(XX.shape)
167 ax.contour(XX, YY, Z, colors='black', levels=[0], linewidths=2)
168 ax.set_title(f'SVM Lin aire (Accuracy: {acc_linear:.4f})')
169 ax.legend()
170 ax.grid(True, alpha=0.3)

171
172 # SVM RBF
173 ax = axes[1]
174 ax.scatter(X_train_scaled[y_train == 0, 0], X_train_scaled[y_train
    == 0, 1],
            c='red', marker='o', label='Classe 0', s=50, alpha=0.6)
175 ax.scatter(X_train_scaled[y_train == 1, 0], X_train_scaled[y_train
    == 1, 1],
            c='blue', marker='s', label='Classe 1', s=50, alpha=0.6)
176 ax.scatter(svm_rbf_best.support_vectors_[:, 0], svm_rbf_best.
    support_vectors_[:, 1],
            s=200, facecolors='none', edgecolors='black', linewidth
                =2)
177 xlim = ax.get_xlim()
178 ylim = ax.get_ylim()
179 xx = np.linspace(xlim[0], xlim[1], 100)
180 yy = np.linspace(ylim[0], ylim[1], 100)
181 YY, XX = np.meshgrid(yy, xx)
182 xy = np.vstack([XX.ravel(), YY.ravel()]).T
183 Z = svm_rbf_best.decision_function(xy).reshape(XX.shape)
184 ax.contour(XX, YY, Z, colors='black', levels=[0], linewidths=2)
185 ax.contourf(XX, YY, Z, levels=[-np.inf, 0, np.inf],
    alpha=0.3, colors=['red', 'blue'])
186 ax.set_title(f'SVM RBF (Accuracy: {acc_rbf:.4f})')
187 ax.legend()
188 ax.grid(True, alpha=0.3)

189
190 plt.tight_layout()
191 plt.show()

192
193
194 # Questions d'analyse :
195 # 1. Pourquoi le SVM lin aire choue -t-il sur ces donn es ?
196 #       Les donn es ne sont pas lin airement s parables
197 # 2. Comment le kernel RBF capture-t-il la structure non-lin aire
198 #       ?
199 #       Transformation implicite dans un espace de dimension
200 #       sup rieure
201 # 3. Quel est l'effet de gamma sur la fronti re ?
202 #       gamma petit : fronti re lisse, gamma grand : fronti re
203 #       complexe (risque overfitting)

```

Hyperparamètres et Paramètres

Important

Hyperparamètres (fixés avant l'entraînement) :

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

Paramètres (appris pendant l'entraînement) :

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

Classification : Arbre de Décision

Concept

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

Mesures d'Impureté

Formule

Entropie :

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

Gini Impurity :

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

Information Gain (Gain d'information) :

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

où :

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

Algorithme ID3 (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,
5     classification_report
6 import matplotlib.pyplot as plt
7 from sklearn.tree import plot_tree
8
9 class DecisionTree:
10     def __init__(self, max_depth=None, min_samples_split=2,
11                  min_samples_leaf=1, criterion='gini'):
12         self.max_depth = max_depth
13         self.min_samples_split = min_samples_split
14         self.min_samples_leaf = min_samples_leaf
15         self.criterion = criterion
16         self.tree = None
17
18     def _gini(self, y):
19         """Calcul de l'impureté Gini"""
20         if len(y) == 0:
21             return 0
22         counts = Counter(y)
23         proportions = [count/len(y) for count in counts.values()]
24         return 1 - sum(p**2 for p in proportions)
25
26     def _entropy(self, y):
27         """Calcul de l'entropie"""
28         if len(y) == 0:
29             return 0
30         counts = Counter(y)
31         proportions = [count/len(y) for count in counts.values()]
32         return -sum(p * np.log2(p) if p > 0 else 0 for p in
33                     proportions)
34
35     def _impurity(self, y):
36         """Calcul de l'impureté selon le critère"""
37         if self.criterion == 'gini':
38             return self._gini(y)
39         else: # entropy
40             return self._entropy(y)
41
42     def _information_gain(self, y_parent, y_left, y_right):
43         """Calcul du gain d'information"""
44         parent_impurity = self._impurity(y_parent)
45         n = len(y_parent)
46         n_left, n_right = len(y_left), len(y_right)
47
48         if n == 0:
49             return 0
50
51         child_impurity = (n_left/n) * self._impurity(y_left) + \
52                           (n_right/n) * self._impurity(y_right)

```

```

52     return parent_impurity - child_impurity
53
54     def _best_split(self, X, y):
55         """Trouve le meilleur split"""
56         best_gain = -1
57         best_feature = None
58         best_threshold = None
59
60         n_features = X.shape[1]
61
62         for feature_idx in range(n_features):
63             # Trier les valeurs uniques
64             thresholds = np.unique(X[:, feature_idx])
65
66             for threshold in thresholds:
67                 # Division
68                 left_mask = X[:, feature_idx] <= threshold
69                 right_mask = ~left_mask
70
71                 if np.sum(left_mask) == 0 or np.sum(right_mask) == 0:
72                     continue
73
74                 y_left = y[left_mask]
75                 y_right = y[right_mask]
76
77                 # Gain d'information
78                 gain = self._information_gain(y, y_left, y_right)
79
80                 if gain > best_gain:
81                     best_gain = gain
82                     best_feature = feature_idx
83                     best_threshold = threshold
84
85         return best_feature, best_threshold, best_gain
86
87     def _build_tree(self, X, y, depth=0):
88         """Construction 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
150     # Utilisation avec sklearn (version optimisée)
151     from sklearn.tree import DecisionTreeClassifier
152     from sklearn.datasets import load_iris
153
154     # Chargement des données
155     iris = load_iris()
156     X, y = iris.data, iris.target
157
158
159     # Division train/test
160     X_train, X_test, y_train, y_test = train_test_split(
161         X, y, test_size=0.2, random_state=42
162     )
163
164     # Test avec différents hyperparamètres
165     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={  
168             min_samples_split}: "  
169             f"Train={train_acc:.4f}, Test={test_acc:.4f}")  
170  
171 # Meilleur modèle  
172 best_model = DecisionTreeClassifier(  
173     max_depth=3,  
174     min_samples_split=5,  
175     criterion='gini',  
176     random_state=42  
177 )  
178 best_model.fit(X_train, y_train)  
179  
180 # Prédictions  
181 y_pred = best_model.predict(X_test)  
182  
183 # Matrices de confusion  
184 print("\n--- Matrices de confusion ---")  
185 print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")  
186 print("\nMatrice de confusion:")  
187 print(confusion_matrix(y_test, y_pred))  
188 print("\nRapport de classification:")  
189 print(classification_report(y_test, y_pred, target_names=iris.  
190     target_names))  
191  
192 # Visualisation de l'arbre  
193 plt.figure(figsize=(20, 10))  
194 plot_tree(best_model, feature_names=iris.feature_names,  
195             class_names=iris.target_names, filled=True)  
196 plt.title("Arbre de Classification")  
197 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) :

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

Centroïde d'un cluster :

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

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)

Important

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

Script Python : Entraînement et Prédiction

Exemple

```

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 centroïdes"""
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 centroïde aléatoirement,
24            # puis choisir les suivants avec probabilité proportionnelle
25            # la distance au centroïde le plus proche
26            centroids = [X[np.random.randint(len(X))]]
27
28            for _ in range(self.k - 1):
29                distances = np.array([
30                    min([np.linalg.norm(x - c)**2 for c in
31                         centroids])
32                    for x in X
33                ])
34                probabilities = distances / distances.sum()
35                cumulative_probs = probabilities.cumsum()
36                r = np.random.rand()
37                idx = np.searchsorted(cumulative_probs, r)
38                centroids.append(X[idx])
39
40            return np.array(centroids)
41
42    def _assign_clusters(self, X, centroids):

```

```

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('Inertia')
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) :

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

Root Mean Squared Error (RMSE) :

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

Mean Absolute Error (MAE) :

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

Coefficient de Détermination (R^2) :

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

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) :

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

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

Precision (Précision) :

$$PRECISION = \frac{TP}{TP + FP} \quad (64)$$

Exemple : $94/(94 + 2) = 94/96 = 0.979$ (97.9%)

Interprétation : Parmi les plantes prédites comme "Toxiques", 97.9% l'étaient vraiment. La précision indique la fiabilité des prédictions positives.

Recall (Rappel) :

$$RECALL = \frac{TP}{TP + FN} \quad (65)$$

Exemple : $94/(94 + 6) = 94/100 = 0.94$ (94%)

Interprétation : Sur toutes les plantes réellement toxiques, on en a trouvé 94%. Le modèle a manqué 6% des plantes toxiques (les FN).

F1-Score :

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (66)$$

Choix de la métrique selon le contexte :

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

Clustering

Formule

Inertia (Within-cluster sum of squares) :

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

Score de Silhouette :

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

où :

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

Score Davies-Bouldin :

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

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