

SUPPORT DE FORMATION

Science des Données et Machine Learning

FORMATEUR

JOUMAIL Hessam

ORGANISME

HCP - Direction Régionale de Dakhla-Oued Ed-Dahab

LIEU

Siège de HCP - Direction Régionale de Dakhla-Oued Ed-Dahab

MODALITÉ

Formation en Présentiel

DATES DE FORMATION

6 jours - Du 26 Mai 2025 au 31 Mai 2025

1 Support de Formation : Science des Données et Machine Learning

1.1 Table des Matières

1. Introduction Python de base
 2. Analyse de données avec Python
 3. Visualisation de données
 4. Projet pratique
 5. Introduction au Machine Learning
 6. Algorithmes ML avancés
 7. Ressources complémentaires
-

1.2 Introduction Python de base

1.2.1 Introduction

La science des données est un domaine interdisciplinaire qui utilise des méthodes scientifiques, des processus, des algorithmes et des systèmes pour extraire des connaissances et des insights à partir de données structurées et non structurées. Elle combine les statistiques, l'informatique et l'expertise métier pour résoudre des problèmes complexes.

Objectifs d'apprentissage : - Comprendre l'écosystème de la science des données - Maîtriser l'environnement de développement Python - Acquérir les bases de programmation Python nécessaires

1.2.2 Environment Set-Up (Anaconda, Jupyter)

Installation d'Anaconda Anaconda est une distribution Python qui inclut plus de 1500 packages pré-installés pour la science des données.

Étapes d'installation : 1. Télécharger Anaconda depuis anaconda.com 2. Exécuter l'installateur selon votre système d'exploitation 3. Vérifier l'installation avec la commande `conda --version`

Configuration de l'environnement :

Créer un nouvel environnement conda create -n datascience python=3.9

Activer l'environnement conda activate datascience

Installer les packages essentiels conda install numpy pandas matplotlib seaborn scikit-learn jupyter

Jupyter Notebook Jupyter Notebook est un environnement interactif qui permet de créer et partager des documents contenant du code, des équations, des visualisations et du texte narratif.

Lancement de Jupyter :

Démarrer Jupyter Notebook jupyter notebook

Ou Jupyter Lab (interface plus moderne) jupyter lab

1.2.3 Jupyter Overview (cellules, markdown, exécution)

Types de cellules

1. **Cellules de code** : Contiennent du code Python exécutable
2. **Cellules Markdown** : Contiennent du texte formaté, des équations LaTeX
3. **Cellules Raw** : Contiennent du texte brut non formaté

Raccourcis clavier essentiels

- Shift + Enter : Exécuter la cellule et passer à la suivante
- Ctrl + Enter : Exécuter la cellule sans changer de sélection
- A : Insérer une cellule au-dessus
- B : Insérer une cellule en-dessous
- M : Convertir en cellule Markdown
- Y : Convertir en cellule de code

Exemple d'utilisation Markdown Usage de # pour : Titre principal Usage de ## pour : Sous-titre

Texte en gras et *texte en italique*

- Liste à puces
 - Élément 2
1. Liste numérotée
 2. Élément 2

Bloc de code print("Hello World")

Équation LaTeX : $E = mc^2$

Python Crash Course

Variables et types de données

Désactiver tous les warnings Python (de sklearn, matplotlib, numpy, etc.) pendant l'exécution.

```
[9]: import warnings
warnings.filterwarnings('ignore')
```

```
[10]: # Types de données de base
nom = "Data Science"          # String
age = 25                      # Integer
taille = 1.75                 # Float
actif = True                   # Boolean

# Listes
donnees = [1, 2, 3, 4, 5]
noms = ["Alice", "Bob", "Charlie"]

# Dictionnaires
personne = {
    "nom": "Alice",
    "age": 30,
    "ville": "Paris"
}

# Tuples (immutables)
coordonnees = (48.8566, 2.3522)

print(f"Type de 'nom': {type(nom)}")
print(f"Longueur de la liste: {len(donnees)}")
```

```
Type de 'nom': <class 'str'>
Longueur de la liste: 5
```

Conditions et structures de contrôle

```
[12]: # Conditions
score = 85

if score >= 90:
    mention = "Excellent"
elif score >= 80:
    mention = "Bien"
elif score >= 70:
    mention = "Assez bien"
else:
    mention = "Peut mieux faire"

print(f"Score: {score}, Mention: {mention}")
```

```
# Opérateurs logiques
age = 25
permis = True

peut_conduire = age >= 18 and permis
print(f"Peut conduire: {peut_conduire}")
```

Score: 85, Mention: Bien
Peut conduire: True

Boucles

```
[14]: # Boucle for
nombres = [1, 2, 3, 4, 5]

# Parcourir une liste
for nombre in nombres:
    print(f"Carré de {nombre}: {nombre**2}")

# Utiliser range()
for i in range(5):
    print(f"Itération {i}")

# Boucle while
compteur = 0
while compteur < 3:
    print(f"Compteur: {compteur}")
    compteur += 1

# List comprehension (compréhension de liste)
carres = [x**2 for x in range(10)]
print(f"Carrés: {carres}")

# Avec condition
pairs = [x for x in range(10) if x % 2 == 0]
print(f"Nombre pairs: {pairs}")
```

Carré de 1: 1
Carré de 2: 4
Carré de 3: 9
Carré de 4: 16
Carré de 5: 25
Itération 0
Itération 1
Itération 2
Itération 3
Itération 4
Compteur: 0
Compteur: 1

```
Compteur: 2
Carrés: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
Nombres pairs: [0, 2, 4, 6, 8]
```

Fonctions

```
[16]: # Fonction simple
def saluer(nom):
    return f"Bonjour {nom}!"

# Fonction avec paramètres par défaut
def calculer_moyenne(notes, precision=2):
    if not notes:
        return 0
    moyenne = sum(notes) / len(notes)
    return round(moyenne, precision)

# Fonction avec arguments variables
def calculer_somme(*args):
    return sum(args)

# Exemples d'utilisation
print(saluer("Alice"))
print(calculer_moyenne([15, 17, 12, 18]))
print(calculer_somme(1, 2, 3, 4, 5))

# Fonction lambda (anonyme)
carre = lambda x: x**2
print(f"Carré de 5: {carre(5)}")

# Utilisation avec map et filter
nombres = [1, 2, 3, 4, 5]
carres = list(map(lambda x: x**2, nombres))
pairs = list(filter(lambda x: x % 2 == 0, nombres))

print(f"Carrés: {carres}")
print(f"Pairs: {pairs}")
```

```
Bonjour Alice!
15.5
15
Carré de 5: 25
Carrés: [1, 4, 9, 16, 25]
Pairs: [2, 4]
```

Gestion des erreurs

```
[18]: # Try-except pour gérer les erreurs
def diviser(a, b):
    try:
```

```

        résultat = a / b
        return résultat
    except ZeroDivisionError:
        print("Erreur: Division par zéro!")
        return None
    except TypeError:
        print("Erreur: Types de données invalides!")
        return None
    finally:
        print("Opération terminée")

# Tests
print(diviser(10, 2))
print(diviser(10, 0))
print(diviser("10", 2))

```

```

Opération terminée
5.0
Erreur: Division par zéro!
Opération terminée
None
Erreur: Types de données invalides!
Opération terminée
None

```

1.3 Analyse de données avec Python

1.3.1 NumPy : Tableaux, opérations, indexation

NumPy (Numerical Python) est la bibliothèque fondamentale pour le calcul scientifique en Python. Elle fournit un objet tableau multidimensionnel de haute performance et des outils pour travailler avec ces tableaux.

Pourquoi NumPy ? - Performance : Les opérations sont implémentées en C - Facilité d'utilisation : Syntaxe intuitive pour les opérations mathématiques - Écosystème : Base de nombreuses autres bibliothèques (Pandas, Scikit-learn)

Création de tableaux

```
[20]: import numpy as np

# Création de tableaux de base
arr1d = np.array([1, 2, 3, 4, 5])
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
arr3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

print(f"Tableau 1D: {arr1d}")
print(f"Forme: {arr1d.shape}, Type: {arr1d.dtype}")
```

```

# Fonctions de création
zeros = np.zeros((3, 4))           # Matrice de zéros
ones = np.ones((2, 3))            # Matrice de uns
eye = np.eye(3)                  # Matrice identité
arange = np.arange(0, 10, 2)       # Séquence avec pas
linspace = np.linspace(0, 1, 5)    # Séquence avec nombre d'éléments
random_arr = np.random.random((3, 3)) # Nombres aléatoires

print(f"Zéros: {zeros}")
print(f"Arange: {arange}")
print(f"Linspace: {linspace}")

```

Tableau 1D: [1 2 3 4 5]
 Forme: (5,), Type: int32
 Zéros:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
 Arange: [0 2 4 6 8]
 Linspace: [0. 0.25 0.5 0.75 1.]

Opérations sur les tableaux

```

[22]: # Opérations arithmétiques élémentaires
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])

print(f"Addition: {a + b}")
print(f"Multiplication: {a * b}")
print(f"Puissance: {a ** 2}")
print(f"Racine carrée: {np.sqrt(a)}")

# Opérations avec des scalaires
print(f"a + 10: {a + 10}")
print(f"a * 2: {a * 2}")

# Opérations matricielles
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print(f"Produit matriciel:\n{np.dot(A, B)}")
# Ou avec l'opérateur @
print(f"Produit avec @:\n{A @ B}")

# Transposée
print(f"Transposée de A:\n{A.T}")

```

```

# Fonctions statistiques
data = np.random.normal(0, 1, (5, 4))  # Distribution normale
print(f"Données: \n{data}")
print(f"Moyenne: {np.mean(data)}")
print(f" Médiane: {np.median(data)}")
print(f" Écart-type: {np.std(data)}")
print(f" Variance: {np.var(data)}")

# Par axe
print(f" Moyenne par colonne: {np.mean(data, axis=0)}")
print(f" Moyenne par ligne: {np.mean(data, axis=1)}")

```

```

Addition: [ 6  8 10 12]
Multiplication: [ 5 12 21 32]
Puissance: [ 1  4  9 16]
Racine carrée: [1.           1.41421356 1.73205081 2.          ]
a + 10: [11 12 13 14]
a * 2: [2 4 6 8]
Produit matriciel:
[[19 22]
 [43 50]]
Produit avec @:
[[19 22]
 [43 50]]
Transposée de A:
[[1 3]
 [2 4]]
Données:
[[ 1.13645008  1.85192221 -0.77259191 -0.00257937]
 [-0.3863576  -1.36586528  0.13906119  0.85238271]
 [ 0.05597439 -0.56909163 -1.18825164  1.07201684]
 [ 0.01587642 -0.40057904 -0.08444815 -1.53470365]
 [ 0.53640714  0.85784596 -0.66594766 -0.00975523]]
Moyenne: -0.023111710147244578
Médiane: -0.006167296260576015
Écart-type: 0.8644838822904776
Variance: 0.7473323827400162
Moyenne par colonne: [ 0.27167009  0.07484645 -0.51443563  0.07547226]
Moyenne par ligne: [ 0.55330025 -0.19019474 -0.15733801 -0.50096361  0.17963755]

```

Indexation et slicing

```

[24]: # Indexation 1D
arr = np.array([10, 20, 30, 40, 50])
print(f"Premier élément: {arr[0]}")
print(f"Dernier élément: {arr[-1]}")
print(f"Slice [1:4]: {arr[1:4]}")
print(f"Éléments pairs: {arr[::2]}")

```

```

# Indexation 2D
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(f"Élément [1, 2]: {matrix[1, 2]}")
print(f"Première ligne: {matrix[0, :]}")
print(f"Première colonne: {matrix[:, 0]}")
print(f"Sous-matrice:\n{matrix[0:2, 1:3]}")

# Indexation booléenne
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
mask = data > 5
print(f"Masque: {mask}")
print(f"Éléments > 5: {data[mask]}")

# Indexation avancée
indices = [0, 2, 4]
print(f"Éléments aux indices {indices}: {data[indices]}")

# Modification avec indexation
arr_copy = arr.copy()
arr_copy[arr_copy > 30] = 0
print(f"Après modification: {arr_copy}")

```

```

Premier élément: 10
Dernier élément: 50
Slice [1:4]: [20 30 40]
Éléments pairs: [10 30 50]
Élément [1, 2]: 6
Première ligne: [1 2 3]
Première colonne: [1 4 7]
Sous-matrice:
[[2 3]
 [5 6]]
Masque: [False False False False False  True  True  True  True  True]
Éléments > 5: [ 6  7  8  9 10]
Éléments aux indices [0, 2, 4]: [1 3 5]
Après modification: [10 20 30  0  0]

```

Reshape et manipulation de forme

```

[26]: # Reshape
arr = np.arange(12)
print(f"Tableau original: {arr}")

# Différentes formes
matrix_3x4 = arr.reshape(3, 4)
matrix_2x6 = arr.reshape(2, 6)
matrix_auto = arr.reshape(-1, 3)  # -1 pour calcul automatique

```

```

print(f"3x4:\n{matrix_3x4}")
print(f"2x6:\n{matrix_2x6}")
print(f"Auto x 3:\n{matrix_auto}")

# Aplatissement
flat = matrix_3x4.flatten()
ravel = matrix_3x4.ravel() # Vue, pas de copie
print(f"Aplati: {flat}")

# Concaténation
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

# Verticalement (axis=0)
v_concat = np.concatenate([a, b], axis=0)
# Horizontalement (axis=1)
h_concat = np.concatenate([a, b], axis=1)

print(f"Concaténation verticale:\n{v_concat}")
print(f"Concaténation horizontale:\n{h_concat}")

# Alternatives
v_stack = np.vstack([a, b])
h_stack = np.hstack([a, b])
print(f"VStack:\n{v_stack}")
print(f"HStack:\n{h_stack}")

```

Tableau original: [0 1 2 3 4 5 6 7 8 9 10 11]

3x4:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

2x6:

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
```

Auto x 3:

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

Aplati: [0 1 2 3 4 5 6 7 8 9 10 11]

Concaténation verticale:

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

Concaténation horizontale:

```
[[1 2 5 6]
 [3 4 7 8]]
VStack:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
HStack:
[[1 2 5 6]
 [3 4 7 8]]
```

1.3.2 Pandas : DataFrames, nettoyage, exploration

Pandas est la bibliothèque de référence pour la manipulation et l'analyse de données en Python. Elle fournit des structures de données puissantes et flexibles pour travailler avec des données "relationnelles" ou "étiquetées".

Structures de données principales : - **Series** : Tableau unidimensionnel avec index - **DataFrame** : Structure bidimensionnelle avec lignes et colonnes étiquetées

Création de DataFrames

```
[28]: import pandas as pd
import numpy as np

# À partir d'un dictionnaire
data_dict = {
    'nom': ['Alice', 'Bob', 'Charlie', 'Diana'],
    'age': [25, 30, 35, 28],
    'ville': ['Paris', 'Lyon', 'Marseille', 'Paris'],
    'salaire': [50000, 60000, 55000, 52000]
}

df = pd.DataFrame(data_dict)
print("DataFrame créé à partir d'un dictionnaire:")
print(df)
print(f"\nInformations sur le DataFrame:")
print(df.info())

# À partir d'un tableau NumPy
np_data = np.random.randn(5, 3)
df_numpy = pd.DataFrame(
    np_data,
    columns=['A', 'B', 'C'],
    index=['Ligne1', 'Ligne2', 'Ligne3', 'Ligne4', 'Ligne5']
)
print(f"\nDataFrame à partir de NumPy:\n{df_numpy}")

# Series (colonne unique)
```

```
series = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
print(f"\nSeries: {series}")
```

DataFrame créé à partir d'un dictionnaire:

```
nom    age      ville   salaire
0   Alice    25      Paris    50000
1     Bob    30      Lyon    60000
2 Charlie   35  Marseille  55000
3   Diana    28      Paris    52000
```

Informations sur le DataFrame:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype  
 ---  --      --      --      --    
 0   nom      4 non-null    object  
 1   age      4 non-null    int64  
 2   ville     4 non-null    object  
 3   salaire   4 non-null    int64  
dtypes: int64(2), object(2)
memory usage: 260.0+ bytes
None
```

DataFrame à partir de NumPy:

```
          A         B         C
Ligne1 -0.347494  0.075913 -1.104132
Ligne2 -0.264915  0.866100  0.114864
Ligne3 -1.219740 -0.874575  0.505481
Ligne4 -0.959125  0.936282 -0.689046
Ligne5 -1.095980 -0.528223 -0.920902
```

Series:

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
```

Lecture et écriture de fichiers

```
[30]: # Créer des données d'exemple
sample_data = {
    'produit': ['A', 'B', 'C', 'D', 'E'] * 20,
    'prix': np.random.uniform(10, 100, 100),
    'quantite': np.random.randint(1, 50, 100),
    'date': pd.date_range('2023-01-01', periods=100, freq='D')}
```

```

}

df_sample = pd.DataFrame(sample_data)

# Sauvegarder en CSV
df_sample.to_csv('donnees_sample.csv', index=False)

# Lire un fichier CSV
df_loaded = pd.read_csv('donnees_sample.csv')
print("Données chargées depuis CSV:")
print(df_loaded.head())

# Autres formats
# df.to_excel('data.xlsx', index=False) # Excel
# df.to_json('data.json') # JSON
# df.to_parquet('data.parquet') # Parquet (efficace)

# Paramètres de lecture utiles
df_custom = pd.read_csv(
    'donnees_sample.csv',
    parse_dates=['date'], # Parser les dates
    index_col='date', # Utiliser date comme index
    nrows=10 # Lire seulement 10 lignes
)
print(f"\nDataFrame personnalisé:\n{df_custom.head()}")

```

Données chargées depuis CSV:

	produit	prix	quantite	date
0	A	40.711894	2	2023-01-01
1	B	97.300172	28	2023-01-02
2	C	42.365943	24	2023-01-03
3	D	77.378830	45	2023-01-04
4	E	12.307798	21	2023-01-05

DataFrame personnalisé:

	produit	prix	quantite
date			
2023-01-01	A	40.711894	2
2023-01-02	B	97.300172	28
2023-01-03	C	42.365943	24
2023-01-04	D	77.378830	45
2023-01-05	E	12.307798	21

Exploration de données

[32]:

```

# Informations générales
print("Aperçu des données:")
print(df.head()) # Premiers 5 éléments
print(df.tail(3)) # Derniers 3 éléments

```

```

print(f"Shape: {df.shape}")
print(f"Colonnes: {df.columns.tolist()}")
print(f"Index: {df.index}")

# Statistiques descriptives
print("\nStatistiques descriptives:")
print(df.describe()) # Pour colonnes numériques
print(df.describe(include='object')) # Pour colonnes textuelles

# Informations sur les types et valeurs manquantes
print(f"\nTypes de données:\n{df.dtypes}")
print(f"\nValeurs manquantes:\n{df.isnull().sum()}")
print(f"\nValeurs uniques par colonne:")
for col in df.columns:
    print(f"{col}: {df[col].nunique()} valeurs uniques")

# Valeurs uniques
print(f"\nVilles uniques: {df['ville'].unique()}")
print(f"\nCompte par ville:\n{df['ville'].value_counts()}")

```

Aperçu des données:

```

      nom  age      ville  salaire
0    Alice  25      Paris    50000
1      Bob  30      Lyon    60000
2  Charlie  35  Marseille    55000
3   Diana  28      Paris    52000
      nom  age      ville  salaire
1      Bob  30      Lyon    60000
2  Charlie  35  Marseille    55000
3   Diana  28      Paris    52000
Shape: (4, 4)
Colonnes: ['nom', 'age', 'ville', 'salaire']
Index: RangeIndex(start=0, stop=4, step=1)

```

Statistiques descriptives:

```

      age      salaire
count  4.000000  4.00000
mean  29.500000 54250.00000
std   4.203173 4349.32945
min   25.000000 50000.00000
25%  27.250000 51500.00000
50%  29.000000 53500.00000
75%  31.250000 56250.00000
max  35.000000 60000.00000
      nom  ville
count      4      4
unique     4      3
top      Alice  Paris

```

```

freq      1      2

Types de données:
nom      object
age      int64
ville    object
salaire  int64
dtype: object

Valeurs manquantes:
nom      0
age      0
ville    0
salaire  0
dtype: int64

Valeurs uniques par colonne:
nom: 4 valeurs uniques
age: 4 valeurs uniques
ville: 3 valeurs uniques
salaire: 4 valeurs uniques

Villes uniques: ['Paris' 'Lyon' 'Marseille']

Compte par ville:
ville
Paris      2
Lyon       1
Marseille  1
Name: count, dtype: int64

```

Sélection et filtrage

```

[34]: # Sélection de colonnes
print("Sélection de colonnes:")
print(df['nom'])                                # Une colonne (Series)
print(df[['nom', 'age']])                         # Plusieurs colonnes (DataFrame)

# Sélection par position
print(f"\nPremière ligne:{df.iloc[0]}")
print(f"\nPremières 2 lignes, colonnes 1-2:{df.iloc[0:2, 1:3]}")

# Sélection par label
df_indexed = df.set_index('nom')
print(f"\nSélection par label:{df_indexed.loc['Alice']}")

# Filtrage
print("\nFiltrage:")

```

```

# Condition simple
df_paris = df[df['ville'] == 'Paris']
print(f"Personnes à Paris:\n{df_paris}")

# Conditions multiples
df_filtered = df[(df['age'] > 25) & (df['salaire'] > 50000)]
print(f"\nPersonnes > 25 ans et salaire > 50k:\n{df_filtered}")

# Utilisation de isin()
villes_cibles = ['Paris', 'Lyon']
df_villes = df[df['ville'].isin(villes_cibles)]
print(f"\nPersonnes à Paris ou Lyon:\n{df_villes}")

# Filtrage avec query (syntaxe plus lisible)
df_query = df.query("age > 25 and ville == 'Paris'")
print(f"\nAvec query():\n{df_query}")

```

Sélection de colonnes:

```

0      Alice
1      Bob
2    Charlie
3    Diana
Name: nom, dtype: object
      nom  age
0    Alice  25
1      Bob  30
2  Charlie  35
3    Diana  28

```

Première ligne:

```

nom      Alice
age      25
ville    Paris
salaire  50000
Name: 0, dtype: object

```

Premières 2 lignes, colonnes 1-2:

```

  age  ville
0  25  Paris
1  30  Lyon

```

Sélection par label:

```

age      25
ville    Paris
salaire  50000
Name: Alice, dtype: object

```

Filtrage:

```

Personnes à Paris:
    nom  age  ville  salaire
0  Alice  25  Paris    50000
3  Diana  28  Paris    52000

Personnes > 25 ans et salaire > 50k:
    nom  age      ville  salaire
1    Bob  30      Lyon    60000
2  Charlie  35  Marseille  55000
3  Diana  28      Paris    52000

Personnes à Paris ou Lyon:
    nom  age  ville  salaire
0  Alice  25  Paris    50000
1    Bob  30  Lyon    60000
3  Diana  28  Paris    52000

Avec query():
    nom  age  ville  salaire
3  Diana  28  Paris    52000

```

Manipulation de données

```

[36]: # Ajout de colonnes
df['salaire_mensuel'] = df['salaire'] / 12
df['senior'] = df['age'] > 30

print(f"Après ajout de colonnes:\n{df}")

# Modification de colonnes
df['age_groupe'] = pd.cut(df['age'], bins=[0, 25, 30, 35, 100],
                           labels=['<25', '25-30', '30-35', '>35'])
print(f"\nGroupes d'âge:\n{df['age_groupe'].value_counts()}\n")

# Suppression
df_clean = df.drop(['salaire_mensuel'], axis=1)  # Supprimer colonne
df_clean = df_clean.drop(0)                         # Supprimer ligne par index

# Renommage
df_renamed = df.rename(columns={'nom': 'nom_complet', 'age': 'age_annees'})
print(f"\nColonnes renommées: {df_renamed.columns.tolist()}\n")

# Tri
df_sorted = df.sort_values('salaire', ascending=False)
print(f"\nTrié par salaire (décroissant):\n{df_sorted}\n")

# Tri par plusieurs colonnes
df_multi_sort = df.sort_values(['ville', 'age'])

```

```
print(f"\nTrié par ville puis âge:\n{df_multi_sort}")
```

Après ajout de colonnes:

```
    nom  age      ville  salaire  salaire_mensuel  senior
0   Alice  25      Paris    50000      4166.666667  False
1     Bob  30      Lyon    60000      5000.000000  False
2  Charlie  35  Marseille    55000      4583.333333  True
3   Diana  28      Paris    52000      4333.333333  False
```

Groupes d'âge:

```
age_groupe
25-30    2
<25      1
30-35    1
>35      0
Name: count, dtype: int64
```

Colonnes renommées: ['nom_complet', 'age_annees', 'ville', 'salaire', 'salaire_mensuel', 'senior', 'age_groupe']

Trié par salaire (décroissant):

```
    nom  age      ville  salaire  salaire_mensuel  senior  age_groupe
1   Bob  30      Lyon    60000      5000.000000  False    25-30
2  Charlie  35  Marseille    55000      4583.333333  True    30-35
3   Diana  28      Paris    52000      4333.333333  False    25-30
0   Alice  25      Paris    50000      4166.666667  False    <25
```

Trié par ville puis âge:

```
    nom  age      ville  salaire  salaire_mensuel  senior  age_groupe
1   Bob  30      Lyon    60000      5000.000000  False    25-30
2  Charlie  35  Marseille    55000      4583.333333  True    30-35
0   Alice  25      Paris    50000      4166.666667  False    <25
3   Diana  28      Paris    52000      4333.333333  False    25-30
```

Gestion des valeurs manquantes

```
[38]: # Créer des données avec valeurs manquantes
data_missing = {
    'A': [1, 2, np.nan, 4, 5],
    'B': [1, np.nan, 3, 4, np.nan],
    'C': [1, 2, 3, 4, 5]
}
df_missing = pd.DataFrame(data_missing)
print(f"Données avec valeurs manquantes:\n{df_missing}")

# Détection
print(f"\nValeurs manquantes:\n{df_missing.isnull()}")
print(f"\nNombre par colonne:\n{df_missing.isnull().sum()}"
```

```

# Suppression
df_dropna = df_missing.dropna()                      # Supprimer lignes avec NaN
df_dropna_col = df_missing.dropna(axis=1)             # Supprimer colonnes avec NaN
print(f"\nAprès suppression des lignes:\n{df_dropna}")

# Remplacement
df_filled = df_missing.fillna(0)                      # Remplacer par 0
df_filled_mean = df_missing.fillna(df_missing.mean()) # Par la moyenne
df_forward = df_missing.fillna(method='ffill')        # Propagation avant
df_backward = df_missing.fillna(method='bfill')        # Propagation arrière

print(f"\nRempli avec la moyenne:\n{df_filled_mean}")

# Interpolation
df_interp = df_missing.interpolate()
print(f"\nInterpolation linéaire:\n{df_interp}")

```

Données avec valeurs manquantes:

	A	B	C
0	1.0	1.0	1
1	2.0	NaN	2
2	NaN	3.0	3
3	4.0	4.0	4
4	5.0	NaN	5

Valeurs manquantes:

	A	B	C
0	False	False	False
1	False	True	False
2	True	False	False
3	False	False	False
4	False	True	False

Nombre par colonne:

	A	B	C
0	1		
1		2	
2		0	
3			

dtype: int64

Après suppression des lignes:

	A	B	C
0	1.0	1.0	1
3	4.0	4.0	4

Rempli avec la moyenne:

	A	B	C
0	1.0	1.000000	1

```

1 2.0 2.666667 2
2 3.0 3.000000 3
3 4.0 4.000000 4
4 5.0 2.666667 5

```

Interpolation linéaire:

	A	B	C
0	1.0	1.0	1
1	2.0	2.0	2
2	3.0	3.0	3
3	4.0	4.0	4
4	5.0	4.0	5

Groupement et agrégation

```

[40]: # Données d'exemple pour groupement
ventes_data = {
    'region': ['Nord', 'Sud', 'Est', 'Ouest', 'Nord', 'Sud'] * 10,
    'produit': ['A', 'B', 'A', 'B', 'C', 'A'] * 10,
    'ventes': np.random.uniform(100, 1000, 60),
    'mois': np.random.choice(['Jan', 'Fev', 'Mar'], 60)
}
df_ventes = pd.DataFrame(ventes_data)

# Groupement simple
print("Ventes par région:")
ventes_par_region = df_ventes.groupby('region')['ventes'].sum()
print(ventes_par_region)

# Groupement multiple
print("\nVentes par région et produit:")
ventes_multi = df_ventes.groupby(['region', 'produit'])['ventes'].sum()
print(ventes_multi)

# Multiples agrégations
print("\nStatistiques par région:")
stats_region = df_ventes.groupby('region')['ventes'].agg(['sum', 'mean', 'count'])
print(stats_region)

# Agrégation personnalisée
def coefficient_variation(x):
    return x.std() / x.mean()

print("\nCoefficient de variation par région:")
cv_region = df_ventes.groupby('region')['ventes'].agg(coefficient_variation)
print(cv_region)

```

```

# Transform et apply
df_ventes['ventes_norm'] = df_ventes.groupby('region')['ventes'].transform(
    lambda x: (x - x.mean()) / x.std()
)
print(f"\nVentes normalisées par région:\n{df_ventes.head()}")

```

Ventes par région:

```

region
Est      5434.603479
Nord    12023.509172
Ouest    4725.445424
Sud      10455.893493
Name: ventes, dtype: float64

```

Ventes par région et produit:

```

region  produit
Est      A      5434.603479
Nord    A      5465.643385
        C      6557.865787
Ouest    B      4725.445424
Sud      A      6128.459714
        B      4327.433779
Name: ventes, dtype: float64

```

Statistiques par région:

		sum	mean	count
region				
Est	5434.603479	543.460348	10	
Nord	12023.509172	601.175459	20	
Ouest	4725.445424	472.544542	10	
Sud	10455.893493	522.794675	20	

Coefficient de variation par région:

```

region
Est      0.477611
Nord    0.390860
Ouest    0.582429
Sud      0.541026
Name: ventes, dtype: float64

```

Ventes normalisées par région:

	region	produit	ventes	mois	ventes_norm
0	Nord	A	602.102127	Jan	0.003944
1	Sud	B	312.875470	Fev	-0.742169
2	Est	A	238.625912	Fev	-1.174415
3	Ouest	B	991.007803	Mar	1.883788
4	Nord	C	382.986745	Fev	-0.928560

1.4 Visualisation de données

1.4.1 Matplotlib : Courbes, titres, axes

Matplotlib est la bibliothèque de visualisation fondamentale en Python. Elle offre un contrôle précis sur tous les aspects d'un graphique et sert de base à de nombreuses autres bibliothèques de visualisation.

Architecture de Matplotlib : - **Figure** : La fenêtre ou page complète - **Axes** : Les systèmes de coordonnées (un ou plusieurs par figure) - **Artist** : Tous les éléments visibles (lignes, texte, etc.)

Graphiques de base

```
[42]: import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Configuration pour de meilleurs graphiques
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['font.size'] = 12

# Données d'exemple
x = np.linspace(0, 2*np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.sin(2*x)

# Graphique simple
plt.figure(figsize=(12, 8))

# Subplot 1: Graphique linéaire de base
plt.subplot(2, 2, 1)
plt.plot(x, y1)
plt.title('Sinus simple')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.grid(True)

# Subplot 2: Plusieurs courbes
plt.subplot(2, 2, 2)
plt.plot(x, y1, label='sin(x)', color='blue', linewidth=2)
plt.plot(x, y2, label='cos(x)', color='red', linestyle='--')
plt.plot(x, y3, label='sin(2x)', color='green', marker='o', markersize=3)
plt.title('Fonctions trigonométriques')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

```

plt.grid(True, alpha=0.3)

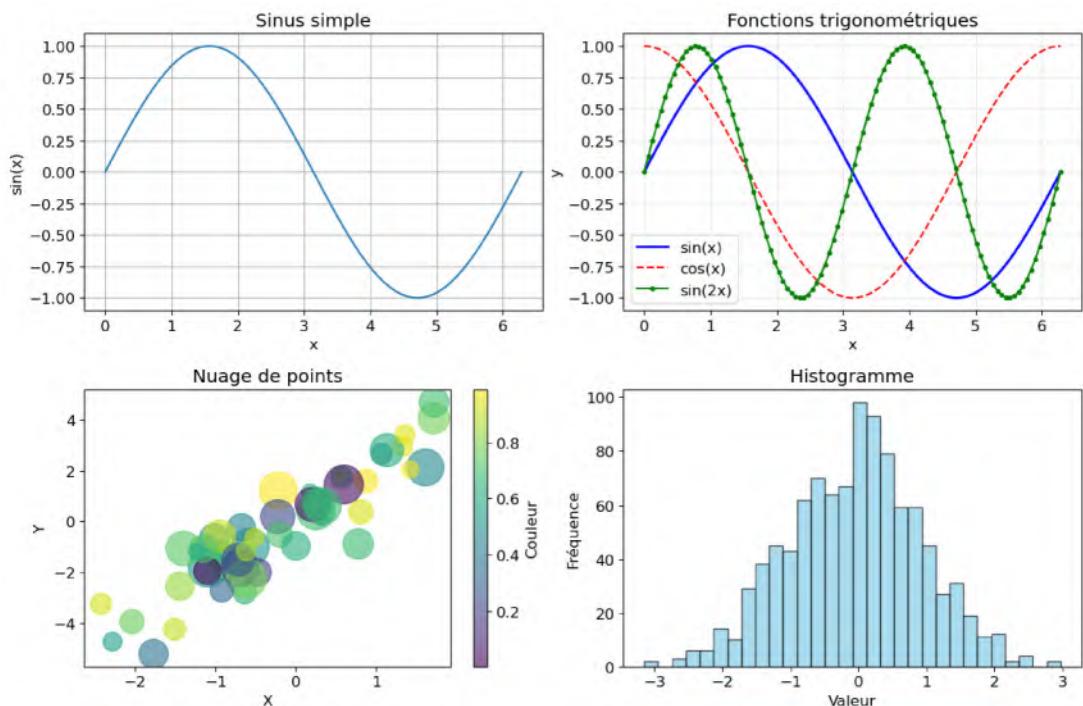
# Subplot 3: Nuage de points
plt.subplot(2, 2, 3)
n_points = 50
x_scatter = np.random.randn(n_points)
y_scatter = 2 * x_scatter + np.random.randn(n_points)
colors = np.random.rand(n_points)
sizes = 1000 * np.random.rand(n_points)

plt.scatter(x_scatter, y_scatter, c=colors, s=sizes, alpha=0.6, cmap='viridis')
plt.colorbar(label='Couleur')
plt.title('Nuage de points')
plt.xlabel('X')
plt.ylabel('Y')

# Subplot 4: Histogramme
plt.subplot(2, 2, 4)
data = np.random.normal(0, 1, 1000)
plt.hist(data, bins=30, alpha=0.7, color='skyblue', edgecolor='black')
plt.title('Histogramme')
plt.xlabel('Valeur')
plt.ylabel('Fréquence')

plt.tight_layout()
plt.show()

```



Personnalisation avancée

```
[44]: # Création d'une figure personnalisée
fig, ax = plt.subplots(figsize=(12, 8))

# Données
x = np.linspace(0, 10, 100)
y = np.exp(-x/3) * np.cos(2*x)

# Graphique principal
line = ax.plot(x, y, color='darkblue', linewidth=3, label='f(x) = e^(-x/3) * cos(2x)')

# Personnalisation des axes
ax.set_xlim(0, 10)
ax.set_ylim(-0.5, 1.1)
ax.set_xlabel('Temps (s)', fontsize=14, fontweight='bold')
ax.set_ylabel('Amplitude', fontsize=14, fontweight='bold')
ax.set_title('Oscillation amortie', fontsize=16, fontweight='bold', pad=20)

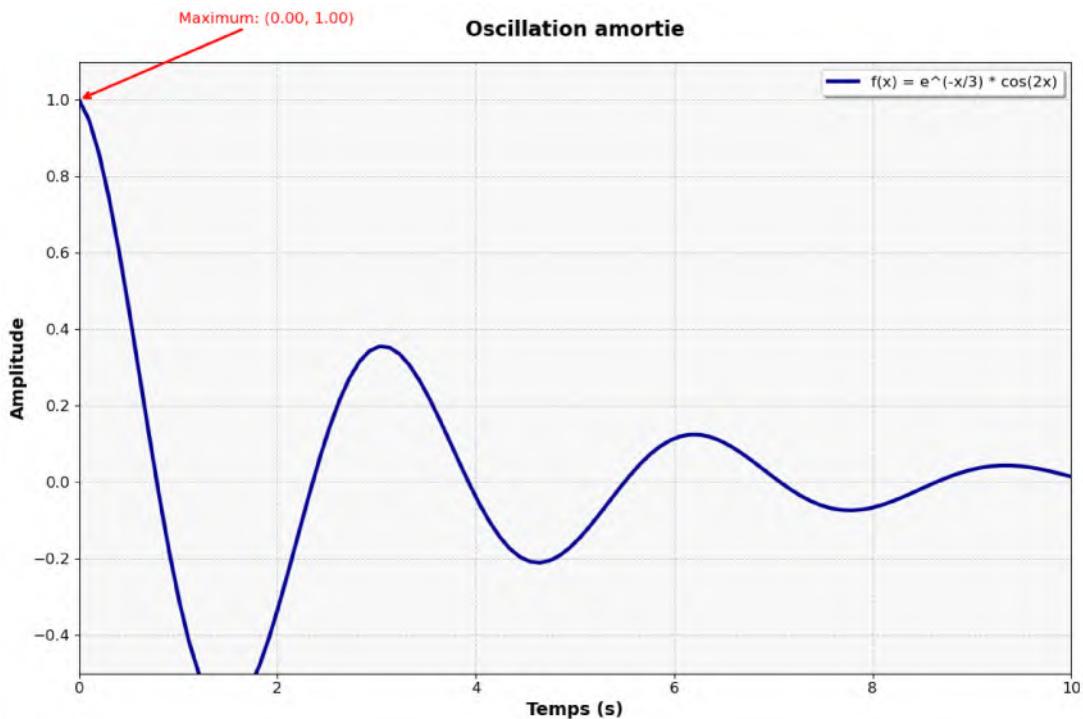
# Grille personnalisée
ax.grid(True, linestyle='--', alpha=0.7)
ax.set_axisbelow(True)

# Annotations
max_idx = np.argmax(y)
ax.annotate(f'Maximum: ({x[max_idx]:.2f}, {y[max_idx]:.2f})',
            xy=(x[max_idx], y[max_idx]), xytext=(x[max_idx]+1, y[max_idx]+0.2),
            arrowprops=dict(arrowstyle='->', color='red', lw=2),
            fontsize=12, color='red')

# Légende personnalisée
ax.legend(loc='upper right', frameon=True, fancybox=True, shadow=True)

# Couleur de fond
ax.set_facecolor('#f8f8f8')

plt.tight_layout()
plt.show()
```



Types de graphiques spécialisés

```
[46]: # Création de différents types de graphiques
fig, axes = plt.subplots(2, 3, figsize=(18, 12))

# 1. Graphique en barres
categories = ['A', 'B', 'C', 'D', 'E']
values = [23, 17, 35, 29, 12]
axes[0, 0].bar(categories, values, color=['red', 'green', 'blue', 'orange', ↴'purple'])
axes[0, 0].set_title('Graphique en barres')
axes[0, 0].set_ylabel('Valeurs')

# 2. Graphique en secteurs
axes[0, 1].pie(values, labels=categories, autopct='%.1f%%', startangle=90)
axes[0, 1].set_title('Graphique en secteurs')

# 3. Graphique en aires empilées
x = np.arange(1, 6)
y1 = [1, 4, 6, 8, 9]
y2 = [2, 2, 7, 10, 12]
y3 = [2, 3, 4, 11, 15]

axes[0, 2].stackplot(x, y1, y2, y3, labels=['Série 1', 'Série 2', 'Série 3'],
```

```

        alpha=0.8)
axes[0, 2].set_title('Aires empilées')
axes[0, 2].legend(loc='upper left')

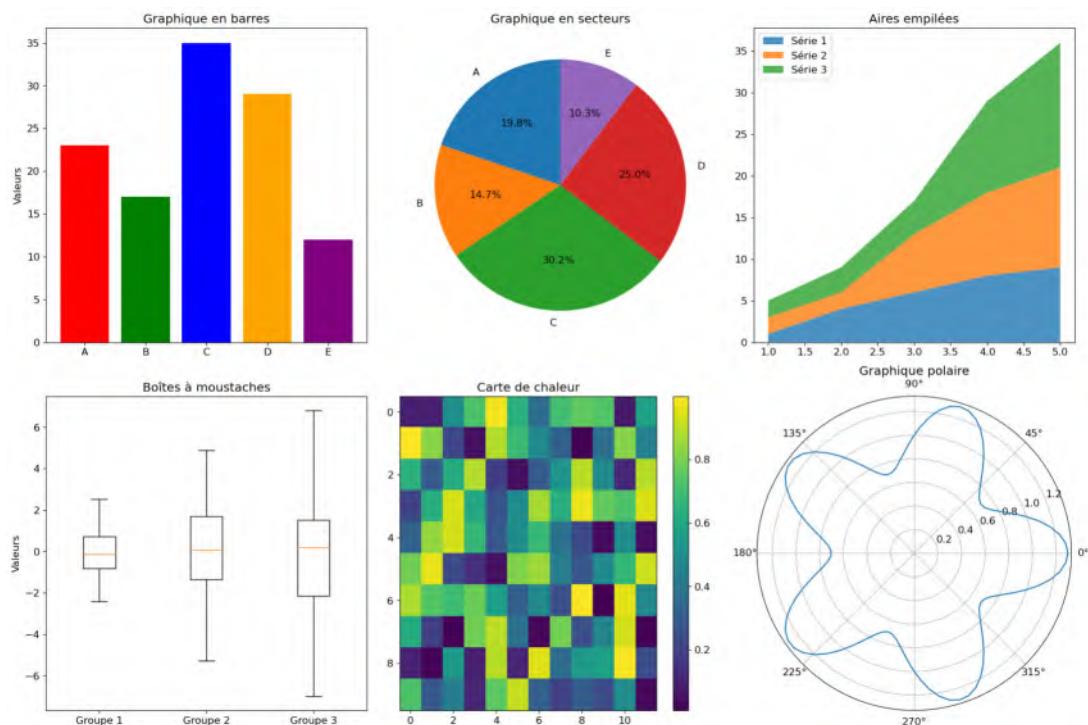
# 4. Box plot
data_box = [np.random.normal(0, std, 100) for std in range(1, 4)]
axes[1, 0].boxplot(data_box, labels=['Groupe 1', 'Groupe 2', 'Groupe 3'])
axes[1, 0].set_title('Boîtes à moustaches')
axes[1, 0].set_ylabel('Valeurs')

# 5. Heatmap
data_heat = np.random.rand(10, 12)
im = axes[1, 1].imshow(data_heat, cmap='viridis', aspect='auto')
axes[1, 1].set_title('Carte de chaleur')
fig.colorbar(im, ax=axes[1, 1])

# 6. Graphique polaire
theta = np.linspace(0, 2*np.pi, 100)
r = 1 + 0.3*np.cos(5*theta)
axes[1, 2].remove() # Supprimer l'axe cartésien
ax_polar = fig.add_subplot(2, 3, 6, projection='polar')
ax_polar.plot(theta, r)
ax_polar.set_title('Graphique polaire')

plt.tight_layout()
plt.show()

```



1.4.2 Seaborn : Heatmaps, distribution, régressions

Seaborn est une bibliothèque de visualisation statistique construite sur matplotlib. Elle offre une interface de haut niveau pour créer des graphiques statistiques attrayants et informatifs.

Avantages de Seaborn : - Styles esthétiques par défaut - Intégration native avec Pandas - Graphiques statistiques spécialisés - Gestion automatique des couleurs et légendes

Configuration et styles

```
[48]: import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

# Configuration de Seaborn
sns.set_style("whitegrid")
sns.set_palette("husl")
plt.rcParams['figure.figsize'] = (10, 6)

# Chargement du dataset Titanic pour les exemples
titanic = sns.load_dataset('titanic')
tips = sns.load_dataset('tips')
iris = sns.load_dataset('iris')

print("Aperçu des datasets:")
print(f"Titanic: {titanic.shape}")
print(f"Tips: {tips.shape}")
print(f"Iris: {iris.shape}")
```

Aperçu des datasets:

Titanic: (891, 15)

Tips: (244, 7)

Iris: (150, 5)

Visualisations de distribution

```
[50]: # Création de graphiques de distribution
fig, axes = plt.subplots(2, 3, figsize=(18, 12))

# 1. Histogramme avec courbe de densité
sns.histplot(data=tips, x='total_bill', kde=True, ax=axes[0, 0])
axes[0, 0].set_title('Distribution des montants')

# 2. Distribution par catégorie
sns.histplot(data=tips, x='total_bill', hue='sex', kde=True, ax=axes[0, 1])
axes[0, 1].set_title('Distribution par sexe')
```

```

# 3. Box plot
sns.boxplot(data=tips, x='day', y='total_bill', ax=axes[0, 2])
axes[0, 2].set_title('Montants par jour')

# 4. Violin plot
sns.violinplot(data=tips, x='day', y='total_bill', ax=axes[1, 0])
axes[1, 0].set_title('Violin plot par jour')

# 5. Strip plot
sns.stripplot(data=tips, x='day', y='total_bill', jitter=True, ax=axes[1, 1])
axes[1, 1].set_title('Strip plot par jour')

# 6. Swarm plot
sns.swarmplot(data=tips, x='day', y='total_bill', ax=axes[1, 2])
axes[1, 2].set_title('Swarm plot par jour')

plt.tight_layout()
plt.show()

# Distribution bivariée
plt.figure(figsize=(15, 5))

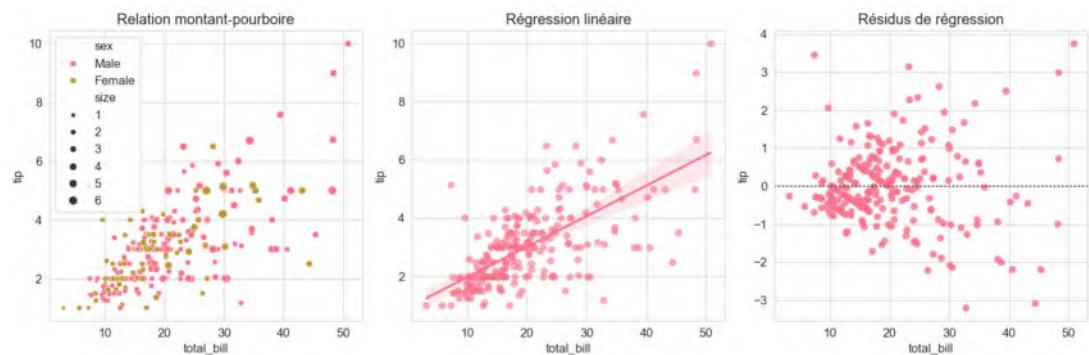
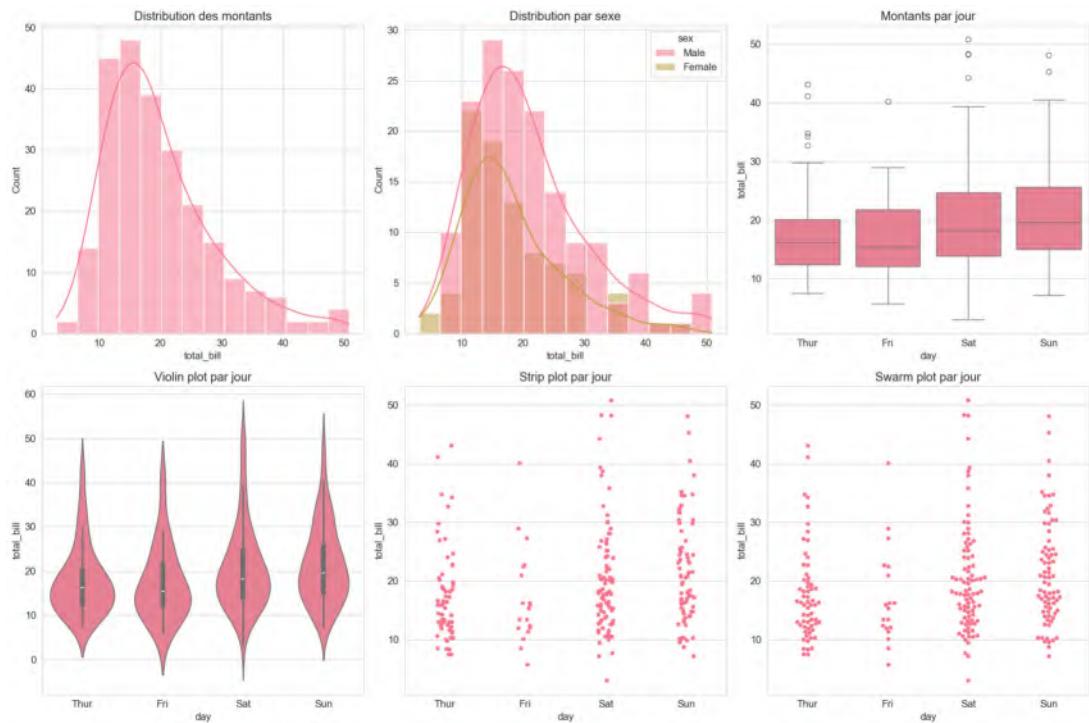
plt.subplot(1, 3, 1)
sns.scatterplot(data=tips, x='total_bill', y='tip', hue='sex', size='size')
plt.title('Relation montant-pourboire')

plt.subplot(1, 3, 2)
sns.regplot(data=tips, x='total_bill', y='tip', scatter_kws={'alpha':0.6})
plt.title('Régression linéaire')

plt.subplot(1, 3, 3)
sns.residplot(data=tips, x='total_bill', y='tip')
plt.title('Résidus de régression')

plt.tight_layout()
plt.show()

```



Matrices de corrélation et heatmaps

```
[52]: # Préparation des données numériques
iris_numeric = iris.select_dtypes(include=[np.number])
tips_numeric = tips.select_dtypes(include=[np.number])

# Calcul des matrices de corrélation
corr_iris = iris_numeric.corr()
corr_tips = tips_numeric.corr()

# Visualisation des matrices de corrélation
```

```

fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Heatmap simple
sns.heatmap(corr_iris, annot=True, cmap='coolwarm', center=0, ax=axes[0])
axes[0].set_title('Corrélations - Dataset Iris')

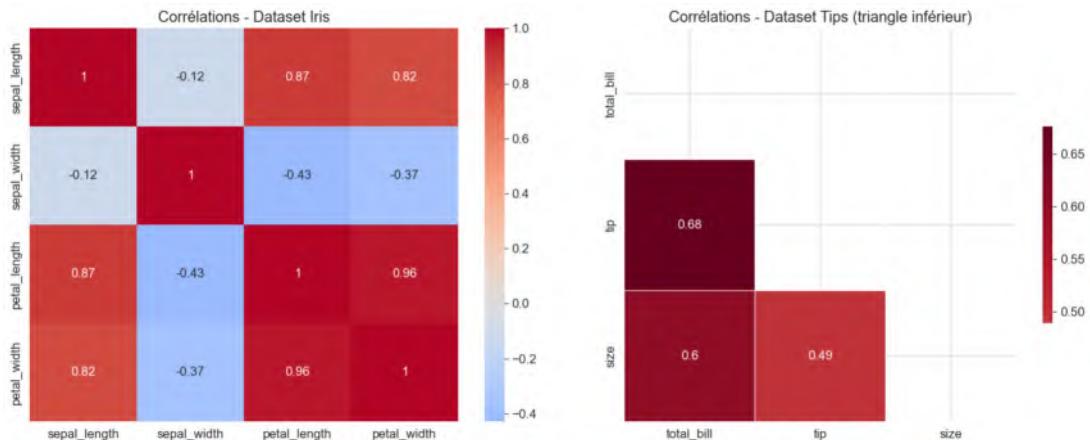
# Heatmap personnalisée
mask = np.triu(np.ones_like(corr_tips, dtype=bool))
sns.heatmap(corr_tips, mask=mask, annot=True, cmap='RdBu_r', center=0,
            square=True, linewidths=0.5, cbar_kws={"shrink": .5}, ax=axes[1])
axes[1].set_title('Corrélations - Dataset Tips (triangle inférieur)')

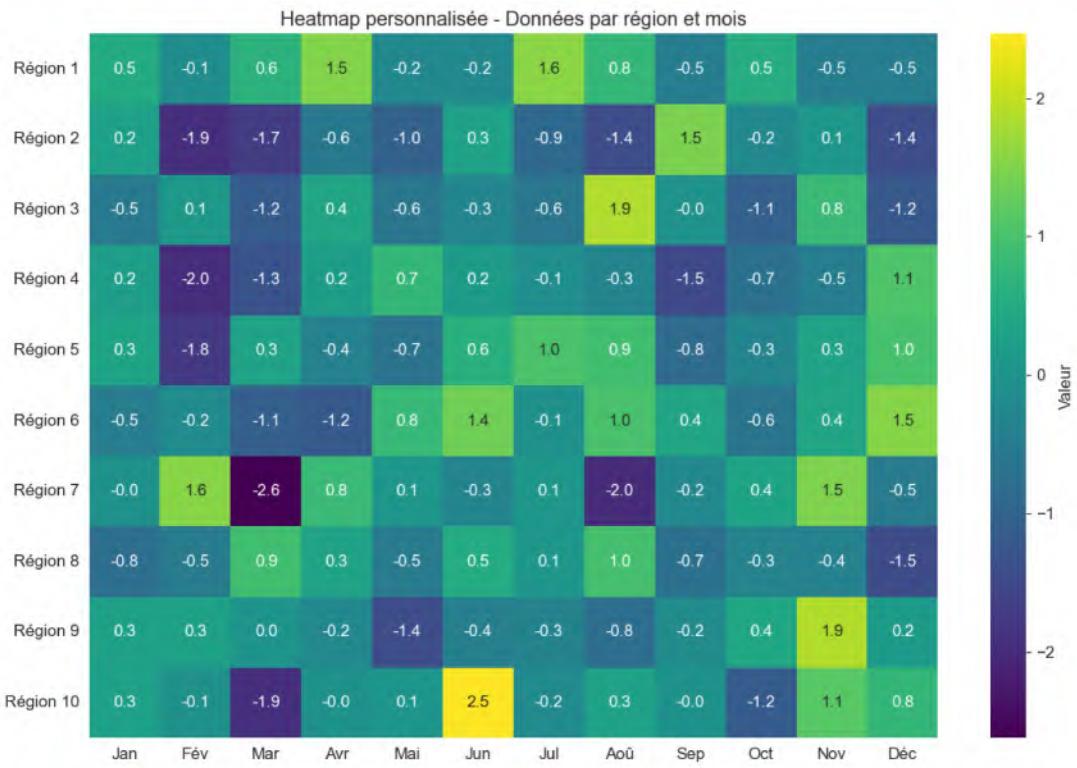
plt.tight_layout()
plt.show()

# Heatmap avec données personnalisées
# Créer une matrice de données
np.random.seed(42)
data_matrix = np.random.randn(10, 12)
months = ['Jan', 'Fév', 'Mar', 'Avr', 'Mai', 'Jun',
          'Jul', 'Aoû', 'Sep', 'Oct', 'Nov', 'Déc']
regions = [f'Région {i+1}' for i in range(10)]

plt.figure(figsize=(12, 8))
sns.heatmap(data_matrix,
            xticklabels=months,
            yticklabels=regions,
            annot=True,
            fmt='.1f',
            cmap='viridis',
            cbar_kws={'label': 'Valeur'})
plt.title('Heatmap personnalisée - Données par région et mois')
plt.tight_layout()
plt.show()

```





Graphiques de régression

```
[54]: # Différents types de régressions
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# 1. Régression linéaire simple
sns.regplot(data=tips, x='total_bill', y='tip', ax=axes[0, 0])
axes[0, 0].set_title('Régression linéaire simple')

# 2. Régression par groupe
sns.lmplot(data=tips, x='total_bill', y='tip', hue='sex', height=6, aspect=1.2)
plt.title('Régression par sexe')
plt.show()

# 3. Régression polynomiale
sns.regplot(data=tips, x='total_bill', y='tip', order=2, ax=axes[0, 1])
axes[0, 1].set_title('Régression polynomiale (ordre 2)')

# 4. Régression robuste
sns.regplot(data=tips, x='total_bill', y='tip', robust=True, ax=axes[1, 0])
```

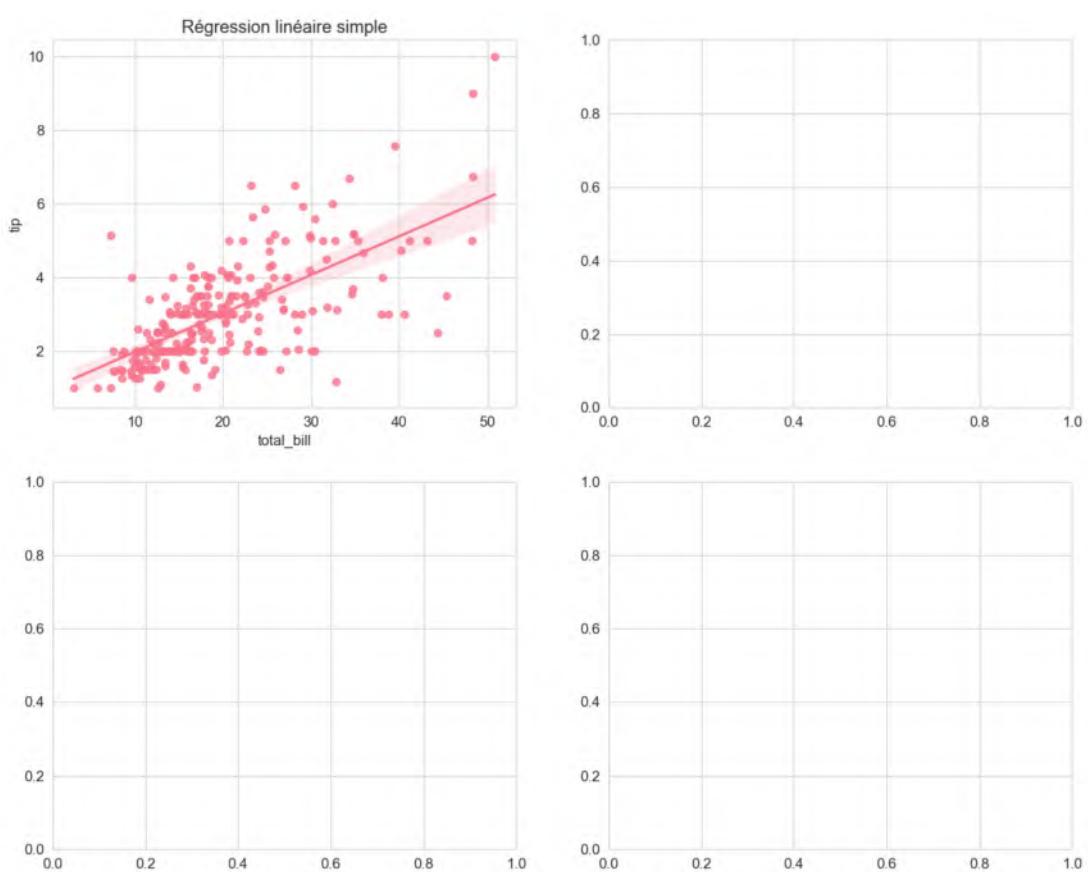
```

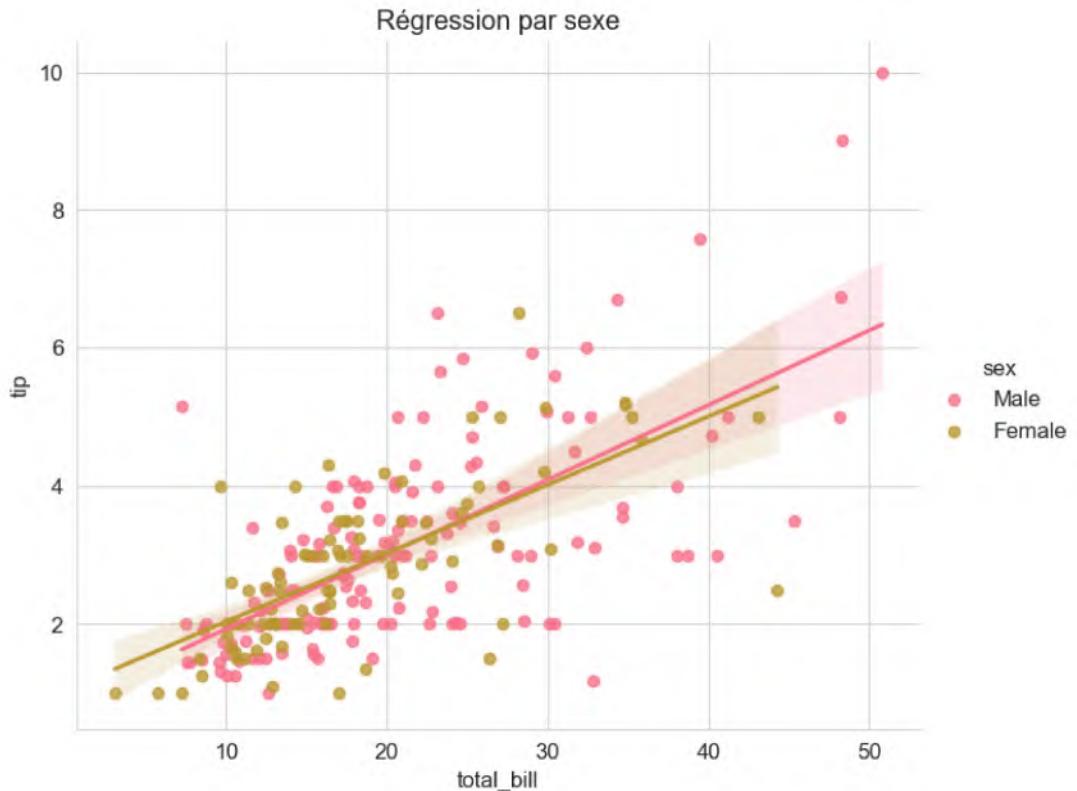
axes[1, 0].set_title('Régression robuste')

# 5. Régression logistique (pour données binaires)
# Créer une variable binaire
tips['high_tip'] = (tips['tip'] > tips['tip'].median()).astype(int)
sns.regplot(data=tips, x='total_bill', y='high_tip', logistic=True, ax=axes[1, 0])
axes[1, 1].set_title('Régression logistique')

plt.tight_layout()
plt.show()

```





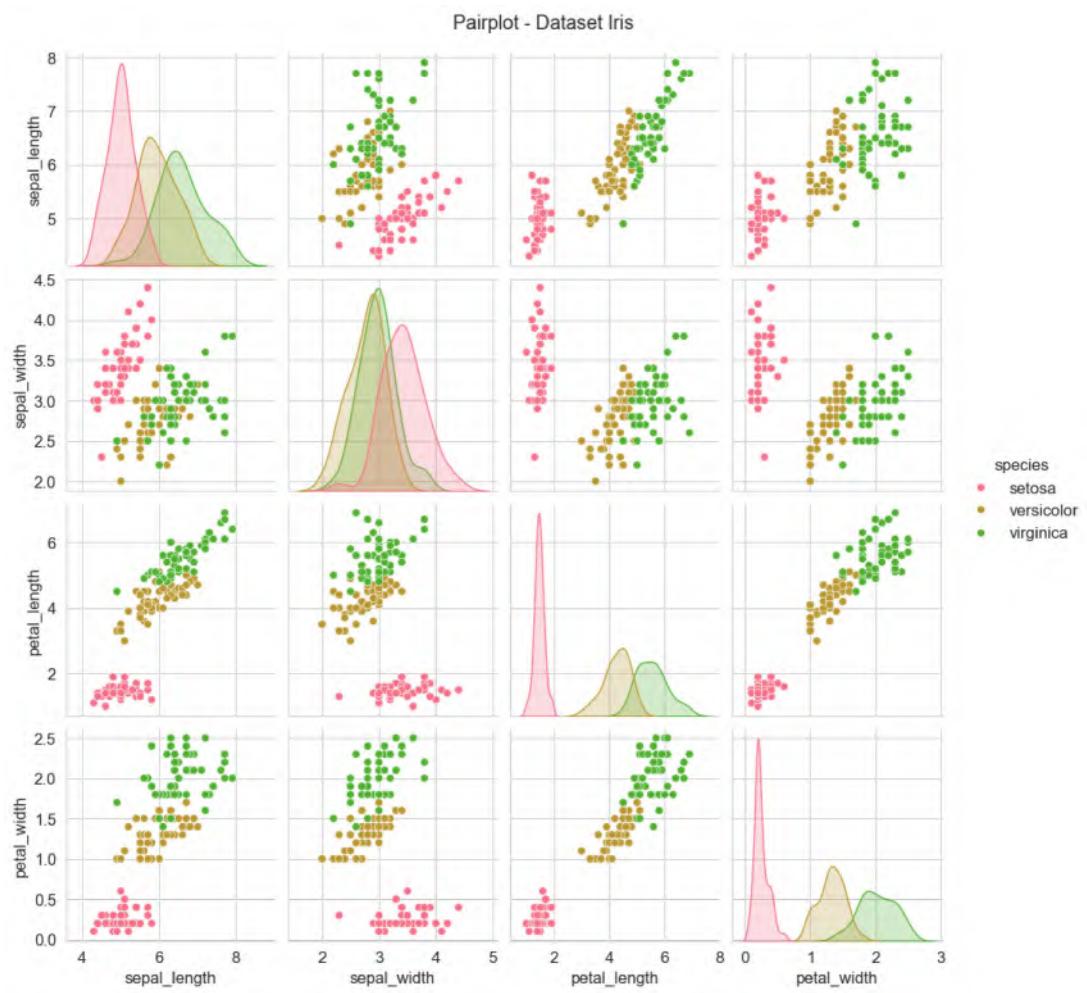
<Figure size 1000x600 with 0 Axes>

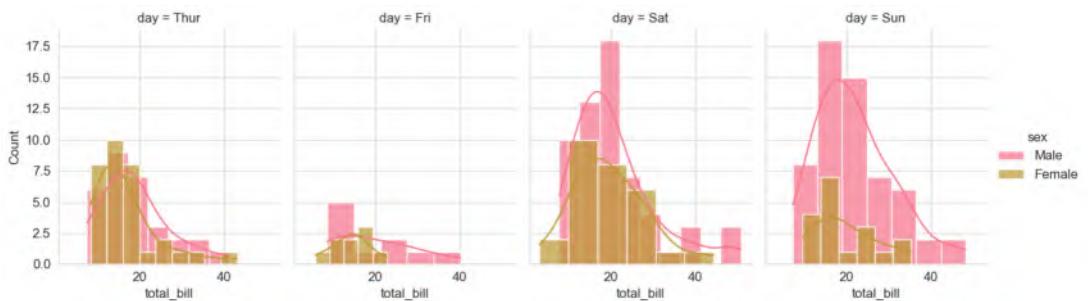
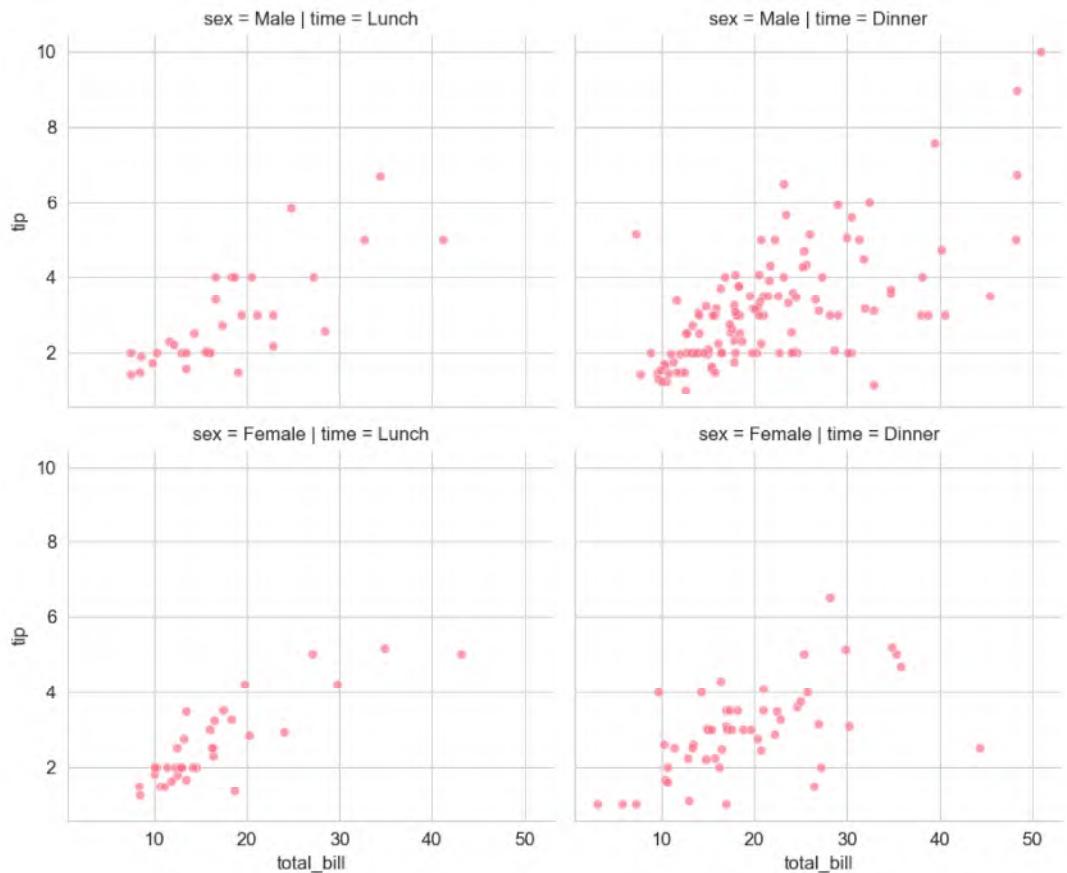
Graphiques multivariés

```
[56]: # Pairplot pour explorer toutes les relations
g = sns.pairplot(iris, hue='species', height=2.5)
g.fig.suptitle('Pairplot - Dataset Iris', y=1.02)
plt.show()

# FacetGrid pour graphiques conditionnels
g = sns.FacetGrid(tips, col='time', row='sex', height=4, aspect=1.2)
g.map(sns.scatterplot, 'total_bill', 'tip', alpha=0.7)
g.add_legend()
plt.show()

# Graphique à facettes avec histogrammes
g = sns.FacetGrid(tips, col='day', hue='sex', height=4, aspect=0.8)
g.map(sns.histplot, 'total_bill', alpha=0.7, kde=True)
g.add_legend()
plt.show()
```





1.4.3 Visualisation intégrée de Pandas

Pandas offre des méthodes de visualisation intégrées qui utilisent matplotlib en arrière-plan, permettant de créer rapidement des graphiques directement à partir des DataFrames.

Graphiques de base avec Pandas

```
[58]: # Données d'exemple
np.random.seed(42)
dates = pd.date_range('2023-01-01', periods=100)
df_time = pd.DataFrame({
    'ventes': np.random.randn(100).cumsum() + 100,
    'profits': np.random.randn(100).cumsum() + 50,
    'couts': np.random.randn(100).cumsum() + 30
}, index=dates)

# Graphiques temporels
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# 1. Graphique linéaire
df_time.plot(ax=axes[0, 0], title='Évolution temporelle')

# 2. Graphique en aires
df_time.plot.area(ax=axes[0, 1], alpha=0.7, title='Aires empilées')

# 3. Graphique en barres
df_monthly = df_time.resample('M').mean()
df_monthly.plot.bar(ax=axes[1, 0], title='Moyennes mensuelles')

# 4. Histogrammes
df_time.plot.hist(bins=20, alpha=0.7, ax=axes[1, 1], title='Distributions')

plt.tight_layout()
plt.show()

# Graphiques spécialisés
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# 1. Box plot
df_time.plot.box(ax=axes[0, 0], title='Boîtes à moustaches')

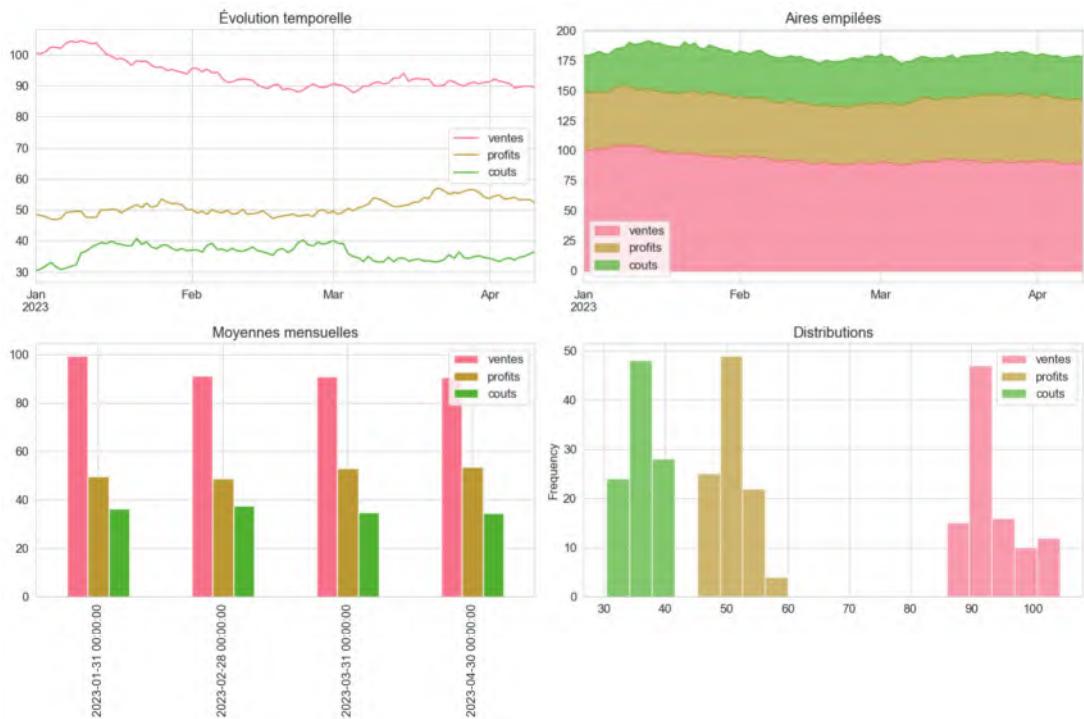
# 2. Nuage de points
df_time.plot.scatter(x='ventes', y='profits', ax=axes[0, 1],
                     title='Ventes vs Profits', alpha=0.6)

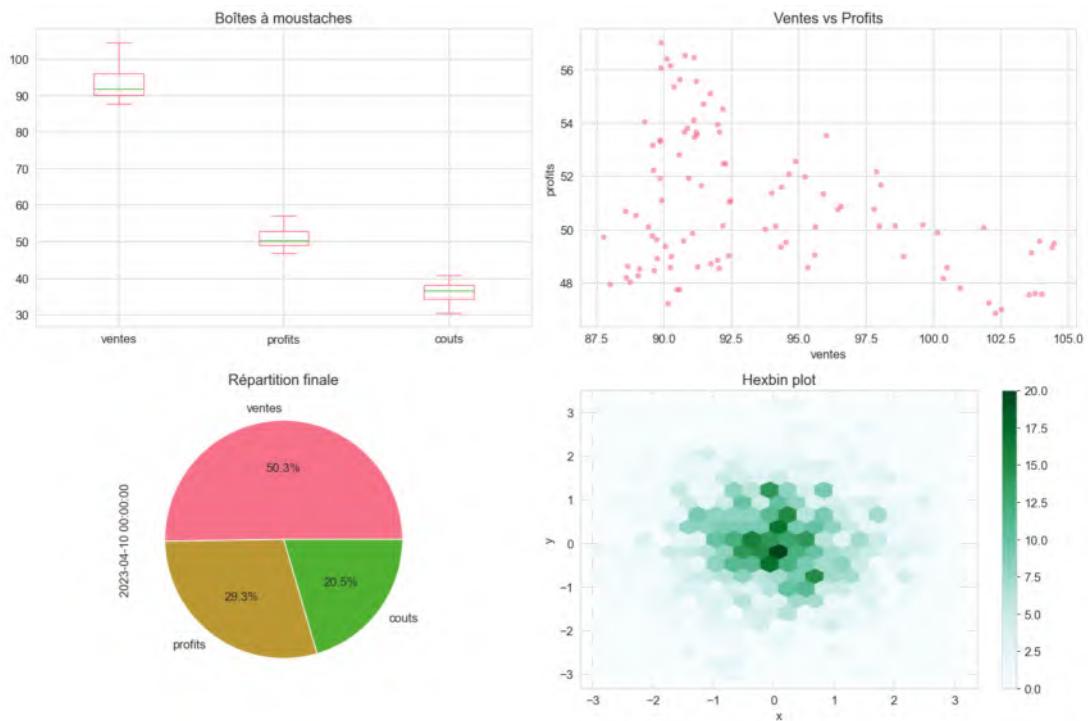
# 3. Graphique en secteurs (dernière valeur)
df_time.iloc[-1].plot.pie(ax=axes[1, 0], title='Répartition finale',
                          autopct='%.1f%%')

# 4. Graphique hexagonal (pour grandes quantités de données)
df_large = pd.DataFrame({
    'x': np.random.randn(1000),
    'y': np.random.randn(1000)
})
```

```
df_large.plot.hexbin(x='x', y='y', gridsize=20, ax=axes[1, 1],
                     title='Hexbin plot')

plt.tight_layout()
plt.show()
```





Graphiques avancés avec Pandas

```
[60]: # Données de ventes par région et produit
np.random.seed(42)
regions = ['Nord', 'Sud', 'Est', 'Ouest']
produits = ['Produit A', 'Produit B', 'Produit C']
mois = pd.date_range('2023-01', periods=12, freq='M')

# Création du DataFrame
data_sales = []
for region in regions:
    for produit in produits:
        for mois_val in mois:
            data_sales.append({
                'region': region,
                'produit': produit,
                'mois': mois_val,
                'ventes': np.random.uniform(100, 1000),
                'unites': np.random.randint(10, 100)
            })

df_sales = pd.DataFrame(data_sales)

# Graphiques groupés
```

```

fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# 1. Ventes par région (graphique en barres groupées)
pivot_region = df_sales.groupby(['region', 'produit'])['ventes'].sum().unstack()
pivot_region.plot.bar(ax=axes[0, 0], title='Ventes par région et produit')
axes[0, 0].legend(title='Produit')

# 2. Évolution temporelle par produit
pivot_temps = df_sales.groupby(['mois', 'produit'])['ventes'].sum().unstack()
pivot_temps.plot(ax=axes[0, 1], title='Évolution des ventes par produit')
axes[0, 1].legend(title='Produit')

# 3. Barres empilées par région
pivot_region.plot.bar(stacked=True, ax=axes[1, 0],
                      title='Ventes empilées par région')
axes[1, 0].legend(title='Produit')

# 4. Graphique en aires par produit
pivot_temps.plot.area(ax=axes[1, 1], alpha=0.7,
                      title='Aires empilées par produit')
axes[1, 1].legend(title='Produit')

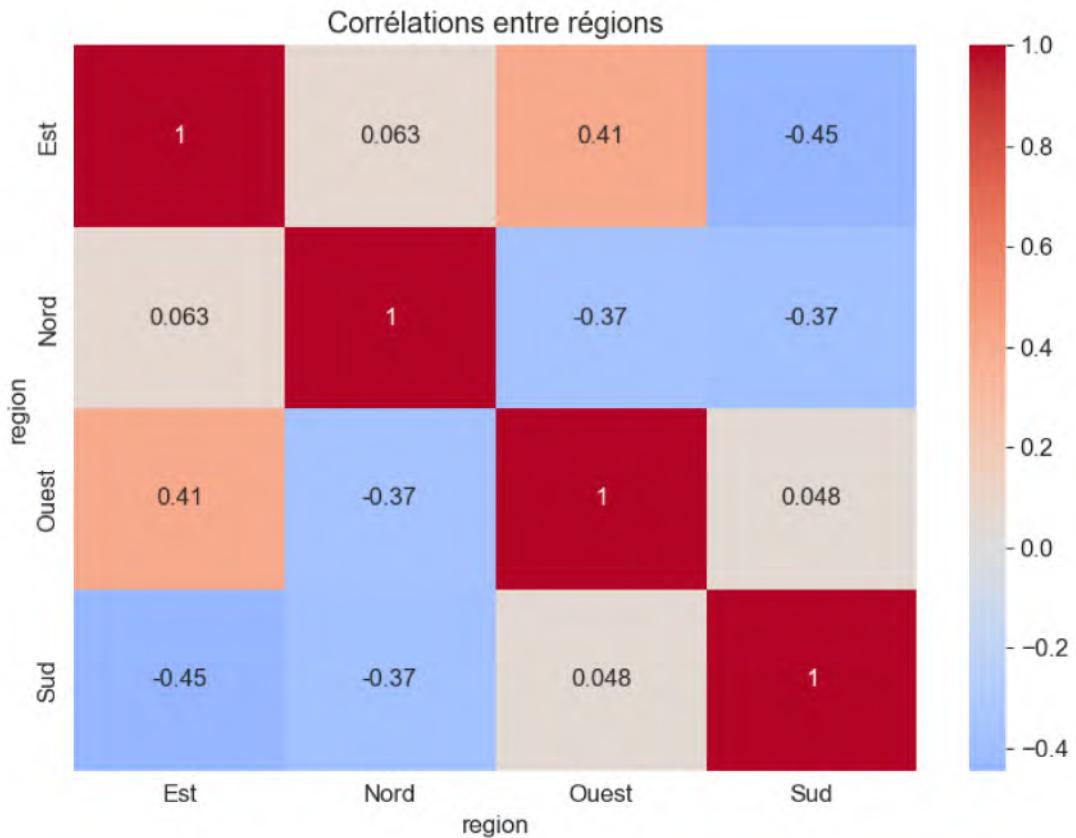
plt.tight_layout()
plt.show()

# Analyse de corrélation avec Pandas
correlation_data = df_sales.pivot_table(
    values='ventes',
    index='mois',
    columns='region',
    aggfunc='sum'
)

# Matrice de corrélation
plt.figure(figsize=(8, 6))
correlation_matrix = correlation_data.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Corrélations entre régions')
plt.tight_layout()
plt.show()

```





1.5 Projet pratique

1.5.1 Cas pratique : Analyse exploratoire complète

Ce projet pratique vous permettra d'appliquer toutes les compétences acquises dans les modules précédents. Nous allons analyser un dataset complet de données de ventes d'une entreprise fictive.

Objectifs du projet : - Charger et explorer un dataset réel - Nettoyer et préparer les données - Effectuer une analyse exploratoire complète - Créer des visualisations informatives - Tirer des conclusions business

Création du dataset d'exemple

```
[62]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta

# Configuration
```

```

np.random.seed(42)
sns.set_style("whitegrid")
plt.rcParams['figure.figsize'] = (12, 8)

# Génération d'un dataset de ventes réaliste
def generate_sales_data(n_records=5000):
    # Paramètres
    regions = ['Nord', 'Sud', 'Est', 'Ouest', 'Centre']
    produits = ['Laptop', 'Smartphone', 'Tablette', 'Casque', 'Souris', 'Clavier']
    categories = ['Électronique', 'Électronique', 'Électronique', 'Accessoires', 'Accessoires', 'Accessoires']
    vendeurs = [f'Vendeur_{i:02d}' for i in range(1, 21)]

    # Prix moyens par produit
    prix_moyens = {'Laptop': 800, 'Smartphone': 500, 'Tablette': 300, 'Casque': 100, 'Souris': 25, 'Clavier': 50}

    data = []
    start_date = datetime(2023, 1, 1)

    for i in range(n_records):
        # Date aléatoire sur l'année
        date = start_date + timedelta(days=np.random.randint(0, 365))

        # Sélection aléatoire des autres variables
        region = np.random.choice(regions)
        produit = np.random.choice(produits)
        vendeur = np.random.choice(vendeurs)

        # Prix avec variation aléatoire
        prix_base = prix_moyens[produit]
        prix = prix_base + np.random.normal(0, prix_base * 0.1)
        prix = max(prix, prix_base * 0.5) # Prix minimum

        # Quantité (plus élevée pour produits moins chers)
        if produit in ['Casque', 'Souris', 'Clavier']:
            quantite = np.random.poisson(3) + 1
        else:
            quantite = np.random.poisson(1) + 1

        # Coût (70-80% du prix)
        cout = prix * np.random.uniform(0.7, 0.8)

        # Satisfaction client (biais selon le produit)
        satisfaction_base = {'Laptop': 4.2, 'Smartphone': 4.0, 'Tablette': 3.8, 'Casque': 4.1, 'Souris': 3.9, 'Clavier': 4.0}

```

```

satisfaction = satisfaction_base[produit] + np.random.normal(0, 0.5)
satisfaction = np.clip(satisfaction, 1, 5)

# Remise (parfois)
remise = np.random.choice([0, 0, 0, 0.05, 0.1, 0.15], p=[0.6, 0.2, 0.1, 0.05, 0.03, 0.02])

data.append({
    'date': date,
    'region': region,
    'produit': produit,
    'categorie': categories[produits.index(produit)],
    'vendeur': vendeur,
    'quantite': quantite,
    'prix_unitaire': round(prix, 2),
    'cout_unitaire': round(cout, 2),
    'remise': remise,
    'satisfaction_client': round(satisfaction, 1)
})

df = pd.DataFrame(data)

# Calculs dérivés
df['prix_total'] = df['prix_unitaire'] * df['quantite']
df['cout_total'] = df['cout_unitaire'] * df['quantite']
df['prix_apres_remise'] = df['prix_total'] * (1 - df['remise'])
df['profit'] = df['prix_apres_remise'] - df['cout_total']
df['marge'] = df['profit'] / df['prix_apres_remise']

# Ajout de quelques valeurs manquantes de façon réaliste
missing_indices = np.random.choice(df.index, size=int(len(df) * 0.02), replace=False)
df.loc[missing_indices, 'satisfaction_client'] = np.nan

return df

# Génération du dataset
df_sales = generate_sales_data(5000)
print("Dataset générée avec succès!")
print(f"Forme du dataset: {df_sales.shape}")
print(f"Colonnes: {df_sales.columns.tolist()}")

```

Dataset générée avec succès!
 Forme du dataset: (5000, 15)
 Colonnes: ['date', 'region', 'produit', 'categorie', 'vendeur', 'quantite',
 'prix_unitaire', 'cout_unitaire', 'remise', 'satisfaction_client', 'prix_total',
 'cout_total', 'prix_apres_remise', 'profit', 'marge']

Phase 1 : Exploration initiale

```
[64]: # 1. Aperçu général des données
print("== APERÇU GÉNÉRAL ==")
print(f"Taille du dataset: {df_sales.shape}")
print(f"Période couverte: {df_sales['date'].min()} à {df_sales['date'].max()}")
print("\nPremiers enregistrements:")
print(df_sales.head())

print("\n== INFORMATIONS SUR LES COLONNES ==")
print(df_sales.info())

print("\n== STATISTIQUES DESCRIPTIVES ==")
print(df_sales.describe())

# 2. Valeurs manquantes
print("\n== VALEURS MANQUANTES ==")
missing_counts = df_sales.isnull().sum()
missing_percentages = (missing_counts / len(df_sales)) * 100
missing_df = pd.DataFrame({
    'Nombre': missing_counts,
    'Pourcentage': missing_percentages
})
print(missing_df[missing_df['Nombre'] > 0])

# 3. Valeurs uniques par colonne
print("\n== VALEURS UNIQUES ==")
for col in df_sales.columns:
    if df_sales[col].dtype == 'object' or col == 'date':
        unique_count = df_sales[col].nunique()
        print(f"{col}: {unique_count} valeurs uniques")
        if unique_count < 20:
            print(f"  Valeurs: {df_sales[col].unique()}")

# 4. Détection des valeurs aberrantes
print("\n== DÉTECTION DES VALEURS ABERRANTES ==")
numeric_cols = ['quantite', 'prix_unitaire', 'profit', 'marge', 'satisfaction_client']

for col in numeric_cols:
    Q1 = df_sales[col].quantile(0.25)
    Q3 = df_sales[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR

    outliers = df_sales[(df_sales[col] < lower_bound) | (df_sales[col] > upper_bound)]
```

```
print(f"col: {len(outliers)} valeurs aberrantes ({len(outliers)}/
len(df_sales)*100:.1f}%)")
```

==== APERÇU GÉNÉRAL ===

Taille du dataset: (5000, 15)

Période couverte: 2023-01-01 00:00:00 à 2023-12-31 00:00:00

Premiers enregistrements:

	date	region	produit	categorie	vendeur	quantite	\
0	2023-04-13	Ouest	Souris	Accessoires	Vendeur_15	2	
1	2023-04-10	Est	Clavier	Accessoires	Vendeur_02	1	
2	2023-11-10	Centre	Casque	Accessoires	Vendeur_17	1	
3	2023-06-24	Est	Casque	Accessoires	Vendeur_20	3	
4	2023-08-30	Sud	Laptop	Électronique	Vendeur_02	1	

	prix_unitaire	cout_unitaire	remise	satisfaction_client	prix_total	\
0	26.62	18.79	0.0	4.7	53.24	
1	51.17	40.89	0.0	4.1	51.17	
2	75.61	60.29	0.0	4.4	75.61	
3	94.56	75.16	0.1	4.2	283.68	
4	751.86	563.53	0.0	5.0	751.86	

	cout_total	prix_apres_remise	profit	marge
0	37.58	53.240	15.660	0.294140
1	40.89	51.170	10.280	0.200899
2	60.29	75.610	15.320	0.202619
3	225.48	255.312	29.832	0.116845
4	563.53	751.860	188.330	0.250485

==== INFORMATIONS SUR LES COLONNES ===

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 5000 entries, 0 to 4999

Data columns (total 15 columns):

#	Column	Non-Null Count	Dtype
---	---	-----	-----
0	date	5000 non-null	datetime64[ns]
1	region	5000 non-null	object
2	produit	5000 non-null	object
3	categorie	5000 non-null	object
4	vendeur	5000 non-null	object
5	quantite	5000 non-null	int64
6	prix_unitaire	5000 non-null	float64
7	cout_unitaire	5000 non-null	float64
8	remise	5000 non-null	float64
9	satisfaction_client	4900 non-null	float64
10	prix_total	5000 non-null	float64
11	cout_total	5000 non-null	float64
12	prix_apres_remise	5000 non-null	float64

```

13 profit           5000 non-null   float64
14 marge            5000 non-null   float64
dtypes: datetime64[ns](1), float64(9), int64(1), object(4)
memory usage: 586.1+ KB
None

==== STATISTIQUES DESCRIPTIVES ====
      date      quantite  prix_unitaire  cout_unitaire \
count      5000  5000.000000  5000.000000  5000.000000
mean  2023-07-03 13:34:27.840000    2.992800  293.522114  220.199510
min    2023-01-01 00:00:00    1.000000  16.200000  12.260000
25%   2023-04-02 00:00:00    2.000000  49.837500  37.217500
50%   2023-07-05 00:00:00    3.000000 118.935000  91.135000
75%   2023-10-04 00:00:00    4.000000 496.050000 373.100000
max   2023-12-31 00:00:00   11.000000 1018.150000 791.690000
std      NaN      1.708839  282.763083  212.468355

      remise  satisfaction_client  prix_total  cout_total \
count  5000.000000      4900.000000  5000.000000  5000.000000
mean    0.008630        3.99451    632.405892  474.506550
min    0.000000        2.10000    20.020000  14.870000
25%    0.000000        3.60000   176.575000 132.950000
50%    0.000000        4.00000   426.580000 317.505000
75%    0.000000        4.40000   844.957500 631.810000
max    0.150000        5.00000  5502.700000 4086.320000
std    0.028481        0.51439   638.213087  479.943044

      prix_apres_remise      profit      marge
count      5000.000000  5000.000000  5000.000000
mean    626.835611  152.329060  0.242630
min    20.020000  2.139500  0.058918
25%   174.562500  41.902500  0.218956
50%   423.145000  99.970000  0.245040
75%   839.645000 202.195000  0.271616
max   5502.700000 1416.380000  0.299997
std   633.292801  157.396608  0.037450

==== VALEURS MANQUANTES ====
      Nombre  Pourcentage
satisfaction_client      100        2.0

==== VALEURS UNIQUES ====
date: 365 valeurs uniques
region: 5 valeurs uniques
  Valeurs: ['Ouest' 'Est' 'Centre' 'Sud' 'Nord']
produit: 6 valeurs uniques
  Valeurs: ['Souris' 'Clavier' 'Casque' 'Laptop' 'Tablette' 'Smartphone']
categorie: 2 valeurs uniques

```

```

Valeurs: ['Accessoires' 'Électronique']
vendeur: 20 valeurs uniques

==== DÉTECTION DES VALEURS ABERRANTES ====
quantite: 75 valeurs aberrantes (1.5%)
prix_unitaire: 0 valeurs aberrantes (0.0%)
profit: 302 valeurs aberrantes (6.0%)
marge: 104 valeurs aberrantes (2.1%)
satisfaction_client: 2 valeurs aberrantes (0.0%)

```

Phase 2 : Nettoyage des données

```

[66]: # Copie de travail
df_clean = df_sales.copy()

print("==== NETTOYAGE DES DONNÉES ===")

# 1. Gestion des valeurs manquantes
print(f"Valeurs manquantes avant nettoyage: {df_clean.isnull().sum().sum()}")

# Pour la satisfaction client, utiliser la médiane par produit
satisfaction_medians = df_clean.groupby('produit')['satisfaction_client'].
    median()
for produit in satisfaction_medians.index:
    mask = (df_clean['produit'] == produit) & (df_clean['satisfaction_client'].
        isnull())
    df_clean.loc[mask, 'satisfaction_client'] = satisfaction_medians[produit]

print(f"Valeurs manquantes après nettoyage: {df_clean.isnull().sum().sum()}")

# 2. Gestion des valeurs aberrantes
print("\n==== GESTION DES VALEURS ABERRANTES ===")

# Suppression des marges négatives extrêmes (erreurs de saisie)
extreme_negative_margin = df_clean['marge'] < -0.5
print(f"Marges négatives extrêmes supprimées: {extreme_negative_margin.sum()}")
df_clean = df_clean[~extreme_negative_margin]

# Plafonnement des quantités très élevées
quantite_99 = df_clean['quantite'].quantile(0.99)
df_clean['quantite'] = df_clean['quantite'].clip(upper=quantite_99)

# 3. Création de nouvelles variables
print("\n==== CRÉATION DE NOUVELLES VARIABLES ===")

# Variables temporelles
df_clean['annee'] = df_clean['date'].dt.year
df_clean['mois'] = df_clean['date'].dt.month

```

```

df_clean['jour_semaine'] = df_clean['date'].dt.day_name()
df_clean['trimestre'] = df_clean['date'].dt.quarter

# Variables métier
df_clean['categorie_prix'] = pd.cut(df_clean['prix_unitaire'],
                                      bins=[0, 50, 200, 500, float('inf')],
                                      labels=['Bas', 'Moyen', 'Élevé', 'Premium'])

df_clean['performance_vendeur'] = df_clean.groupby('vendeur')['profit'].
    transform('mean')
df_clean['top_vendeur'] = df_clean['performance_vendeur'] >_
    df_clean['performance_vendeur'].median()

print(f"Dataset final: {df_clean.shape}")
print("Nouvelles colonnes créées:")
nouvelles_cols = ['annee', 'mois', 'jour_semaine', 'trimestre',_
    'categorie_prix', 'performance_vendeur', 'top_vendeur']
for col in nouvelles_cols:
    print(f" - {col}")

```

==== NETTOYAGE DES DONNÉES ===

Valeurs manquantes avant nettoyage: 100

Valeurs manquantes après nettoyage: 0

==== GESTION DES VALEURS ABERRANTES ===

Marges négatives extrêmes supprimées: 0

==== CRÉATION DE NOUVELLES VARIABLES ===

Dataset final: (5000, 22)

Nouvelles colonnes créées:

- annee
- mois
- jour_semaine
- trimestre
- categorie_prix
- performance_vendeur
- top_vendeur

Phase 3 : Analyse exploratoire approfondie

```

[68]: # 1. Analyse des ventes globales
print("==== ANALYSE DES VENTES GLOBALES ===")

# Métriques clés
total_ventes = df_clean['prix_apres_remise'].sum()
total_profit = df_clean['profit'].sum()
marge_moyenne = df_clean['marge'].mean()
satisfaction_moyenne = df_clean['satisfaction_client'].mean()

```

```

print(f"Chiffre d'affaires total: {total_ventes:.2f} €")
print(f"Profit total: {total_profit:.2f} €")
print(f"Marge moyenne: {marge_moyenne:.1%}")
print(f"Satisfaction moyenne: {satisfaction_moyenne:.1f}/5")

# 2. Analyse par région
print("\n==== ANALYSE PAR RÉGION ====")
region_analysis = df_clean.groupby('region').agg({
    'prix_apres_remise': 'sum',
    'profit': 'sum',
    'marge': 'mean',
    'satisfaction_client': 'mean',
    'quantite': 'sum'
}).round(2)

region_analysis.columns = ['CA', 'Profit', 'Marge_moy', 'Satisfaction_moy', ↴
    'Unités_vendues']
region_analysis = region_analysis.sort_values('CA', ascending=False)
print(region_analysis)

# 3. Analyse par produit
print("\n==== ANALYSE PAR PRODUIT ====")
product_analysis = df_clean.groupby('produit').agg({
    'prix_apres_remise': 'sum',
    'profit': 'sum',
    'marge': 'mean',
    'satisfaction_client': 'mean',
    'quantite': 'sum'
}).round(2)

product_analysis.columns = ['CA', 'Profit', 'Marge_moy', 'Satisfaction_moy', ↴
    'Unités_vendues']
product_analysis = product_analysis.sort_values('CA', ascending=False)
print(product_analysis)

# 4. Analyse temporelle
print("\n==== ANALYSE TEMPORELLE ====")
monthly_sales = df_clean.groupby('mois').agg({
    'prix_apres_remise': 'sum',
    'profit': 'sum',
    'quantite': 'sum'
}).round(2)

print("Ventes mensuelles:")
print(monthly_sales)

```

```

# 5. Analyse des vendeurs
print("\n==== TOP 5 VENDEURS ===")
top_vendeurs = df_clean.groupby('vendeur').agg({
    'prix_apres_remise': 'sum',
    'profit': 'sum',
    'satisfaction_client': 'mean'
}).sort_values('prix_apres_remise', ascending=False).head()

print(top_vendeurs.round(2))

```

==== ANALYSE DES VENTES GLOBALES ===

Chiffre d'affaires total: 3,134,178.05 €

Profit total: 761,645.30 €

Marge moyenne: 24.3%

Satisfaction moyenne: 4.0/5

==== ANALYSE PAR RÉGION ===

	CA	Profit	Marge_moy	Satisfaction_moy	Unités_vendues
region					
Ouest	667929.01	161547.38	0.24	4.00	3205
Est	646615.41	156977.02	0.24	3.99	3069
Nord	613720.64	149371.22	0.24	4.01	2853
Centre	609591.03	148020.68	0.24	4.00	2899
Sud	596321.97	145729.01	0.24	3.97	2911

==== ANALYSE PAR PRODUIT ===

	CA	Profit	Marge_moy	Satisfaction_moy	Unités_vendues
produit					
Laptop	1243622.25	300577.88	0.24	4.20	1559
Smartphone	794461.97	193715.40	0.24	4.01	1601
Tablette	512074.04	124995.24	0.24	3.77	1731
Casque	335544.33	81770.66	0.24	4.09	3379
Clavier	165509.86	40510.90	0.24	3.97	3326
Souris	82965.60	20075.22	0.24	3.93	3341

==== ANALYSE TEMPORELLE ===

Ventes mensuelles:

	prix_apres_remise	profit	quantite
mois			
1	248947.75	60634.75	1362
2	247404.58	60154.58	1187
3	286732.72	68966.72	1244
4	247553.51	60769.53	1161
5	269696.84	66121.17	1285
6	250245.41	60318.84	1169
7	231300.17	55783.32	1178
8	278442.74	67554.67	1316
9	274534.22	66986.58	1179

```

10          281515.43  68249.39      1350
11          242085.54  59108.13      1152
12          275719.13  66997.61      1354

```

==== TOP 5 VENDEURS ===

vendeur	prix_apres_remise	profit	satisfaction_client
Vendeur_09	192463.23	46728.97	3.98
Vendeur_19	183003.05	44904.82	4.03
Vendeur_03	178096.31	43064.94	4.02
Vendeur_12	171718.22	42477.96	4.01
Vendeur_15	166678.96	39981.85	3.99

Phase 4 : Visualisations avancées

```

[70]: # Configuration pour les graphiques
plt.style.use('default')
colors = plt.cm.Set3(np.linspace(0, 1, 12))

# 1. Dashboard de ventes par région
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# Chiffre d'affaires par région
region_ca = df_clean.groupby('region')['prix_apres_remise'].sum().
    ↪sort_values(ascending=False)
axes[0, 0].bar(region_ca.index, region_ca.values, color=colors[:len(region_ca)])
axes[0, 0].set_title('Chiffre d\'affaires par région')
axes[0, 0].set_ylabel('CA (€)')
for i, v in enumerate(region_ca.values):
    axes[0, 0].text(i, v + max(region_ca.values)*0.01, f'{v:.0f}', ha='center')

# Profit par région
region_profit = df_clean.groupby('region')['profit'].sum().
    ↪sort_values(ascending=False)
axes[0, 1].bar(region_profit.index, region_profit.values, color=colors[:len(region_profit)])
axes[0, 1].set_title('Profit par région')
axes[0, 1].set_ylabel('Profit (€)')

# Satisfaction par région
region_satisfaction = df_clean.groupby('region')['satisfaction_client'].mean().
    ↪sort_values(ascending=False)
axes[1, 0].bar(region_satisfaction.index, region_satisfaction.values, ↴
    ↪color=colors[:len(region_satisfaction)])
axes[1, 0].set_title('Satisfaction moyenne par région')
axes[1, 0].set_ylabel('Satisfaction (/5)')
axes[1, 0].set_ylim(0, 5)

```

```

# Unités vendues par région
region_units = df_clean.groupby('region')['quantite'].sum().
    ↪sort_values(ascending=False)
axes[1, 1].bar(region_units.index, region_units.values, color=colors[::
    ↪len(region_units)])
axes[1, 1].set_title('Unités vendues par région')
axes[1, 1].set_ylabel('Unités')

plt.tight_layout()
plt.show()

# 2. Analyse temporelle détaillée
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# Évolution mensuelle des ventes
monthly_data = df_clean.groupby('mois').agg({
    'prix_apres_remise': 'sum',
    'profit': 'sum'
})
months = ['Jan', 'Fév', 'Mar', 'Avr', 'Mai', 'Jun',
          'Jul', 'Aoû', 'Sep', 'Oct', 'Nov', 'Déc']

axes[0, 0].plot(monthly_data.index, monthly_data['prix_apres_remise'],
                 marker='o', linewidth=2, color='blue')
axes[0, 0].set_title('Évolution mensuelle du CA')
axes[0, 0].set_xlabel('Mois')
axes[0, 0].set_ylabel('CA (€)')
axes[0, 0].set_xticks(range(1, 13))
axes[0, 0].set_xticklabels([months[i][:3] for i in range(12)])
axes[0, 0].grid(True, alpha=0.3)

# Ventes par jour de la semaine
weekly_sales = df_clean.groupby('jour_semaine')['prix_apres_remise'].sum()
day_order = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', ↪
    ↪'Saturday', 'Sunday']
day_names = ['Lun', 'Mar', 'Mer', 'Jeu', 'Ven', 'Sam', 'Dim']
weekly_sales = weekly_sales.reindex(day_order)

axes[0, 1].bar(day_names, weekly_sales.values, color='orange', alpha=0.7)
axes[0, 1].set_title('Ventes par jour de la semaine')
axes[0, 1].set_ylabel('CA (€)')
axes[0, 1].tick_params(axis='x', rotation=45)

# Évolution du profit mensuel
axes[1, 0].plot(monthly_data.index, monthly_data['profit'],
                 marker='s', linewidth=2, color='green')
axes[1, 0].set_title('Évolution mensuelle du profit')

```

```

axes[1, 0].set_xlabel('Mois')
axes[1, 0].set_ylabel('Profit (€)')
axes[1, 0].set_xticks(range(1, 13))
axes[1, 0].set_xticklabels([months[i][:3] for i in range(12)])
axes[1, 0].grid(True, alpha=0.3)

# Heatmap des ventes par mois et région
pivot_month_region = df_clean.groupby(['mois', 'region'])['prix_apres_remise'].
    sum().unstack()
sns.heatmap(pivot_month_region, annot=True, fmt=',.0f', cmap='YlOrRd', □
    ax=axes[1, 1])
axes[1, 1].set_title('Heatmap: Ventes par mois et région')
axes[1, 1].set_xlabel('Région')
axes[1, 1].set_ylabel('Mois')

plt.tight_layout()
plt.show()

# 3. Analyse des produits
fig, axes = plt.subplots(2, 2, figsize=(16, 12))

# Performance des produits (CA vs Profit)
product_perf = df_clean.groupby('produit').agg({
    'prix_apres_remise': 'sum',
    'profit': 'sum',
    'quantite': 'sum'
})

scatter = axes[0, 0].scatter(product_perf['prix_apres_remise'],
    product_perf['profit'],
    s=product_perf['quantite']/5,
    alpha=0.7, c=range(len(product_perf)), □
    cmap='viridis')
axes[0, 0].set_xlabel('Chiffre d'affaires (€)')
axes[0, 0].set_ylabel('Profit (€)')
axes[0, 0].set_title('Performance des produits (taille = unités vendues)')

# Ajouter les labels des produits
for i, (product, row) in enumerate(product_perf.iterrows()):
    axes[0, 0].annotate(product, (row['prix_apres_remise'], row['profit']),
        xytext=(5, 5), textcoords='offset points', fontsize=9)

# Distribution des marges par produit
df_clean.boxplot(column='marge', by='produit', ax=axes[0, 1])
axes[0, 1].set_title('Distribution des marges par produit')
axes[0, 1].set_xlabel('Produit')
axes[0, 1].set_ylabel('Marge')

```

```

axes[0, 1].tick_params(axis='x', rotation=45)

# Satisfaction par produit
product_satisfaction = df_clean.groupby('produit')['satisfaction_client'].
    mean().sort_values(ascending=False)
bars = axes[1, 0].bar(product_satisfaction.index, product_satisfaction.values,
                      color='lightcoral', alpha=0.7)
axes[1, 0].set_title('Satisfaction moyenne par produit')
axes[1, 0].set_ylabel('Satisfaction (/5)')
axes[1, 0].tick_params(axis='x', rotation=45)
axes[1, 0].set_ylim(0, 5)

# Ajouter les valeurs sur les barres
for bar, value in zip(bars, product_satisfaction.values):
    axes[1, 0].text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.05,
                    f'{value:.1f}', ha='center', va='bottom')

# Parts de marché (volume)
market_share = df_clean.groupby('produit')['quantite'].sum()
axes[1, 1].pie(market_share.values, labels=market_share.index, autopct='%.1f%%',
               startangle=90, colors=plt.cm.Pastel1.colors)
axes[1, 1].set_title('Parts de marché (unités vendues)')

plt.tight_layout()
plt.show()

# 4. Analyse des corrélations
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Matrice de corrélation des variables numériques
numeric_vars = ['prix_unitaire', 'quantite', 'profit', 'marge', ↴
    'satisfaction_client']
correlation_matrix = df_clean[numeric_vars].corr()

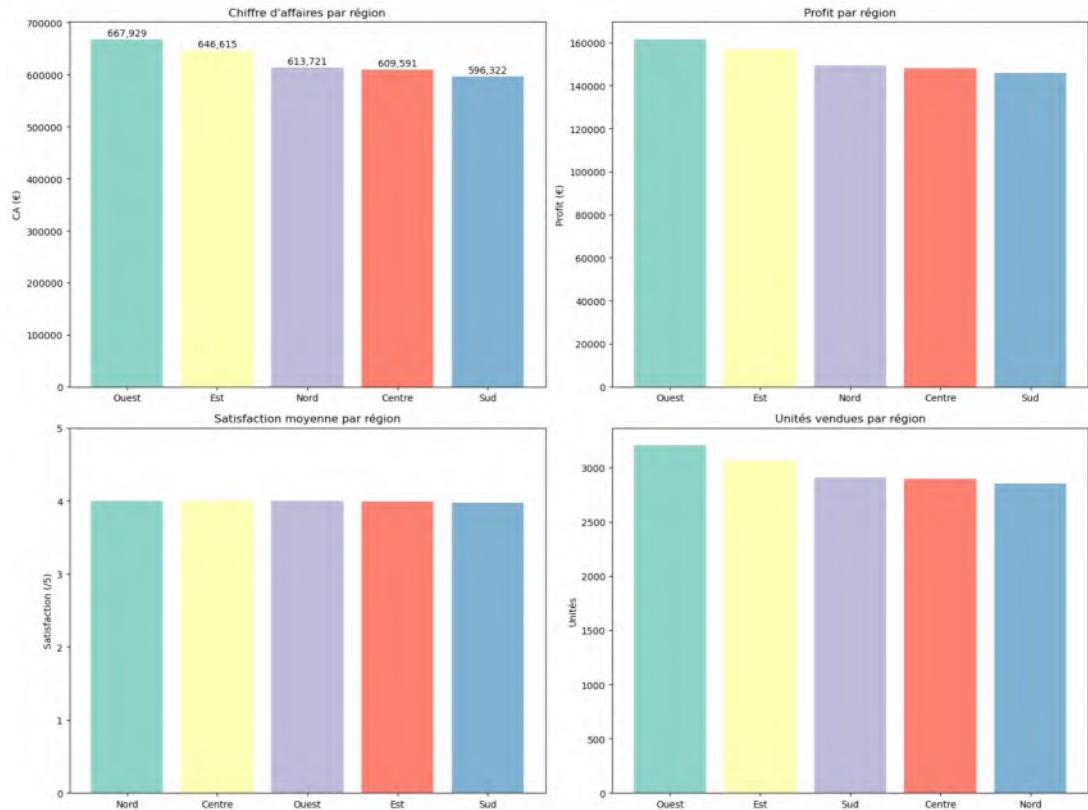
sns.heatmap(correlation_matrix, annot=True, cmap='RdBu_r', center=0,
            square=True, ax=axes[0])
axes[0].set_title('Matrice de corrélation')

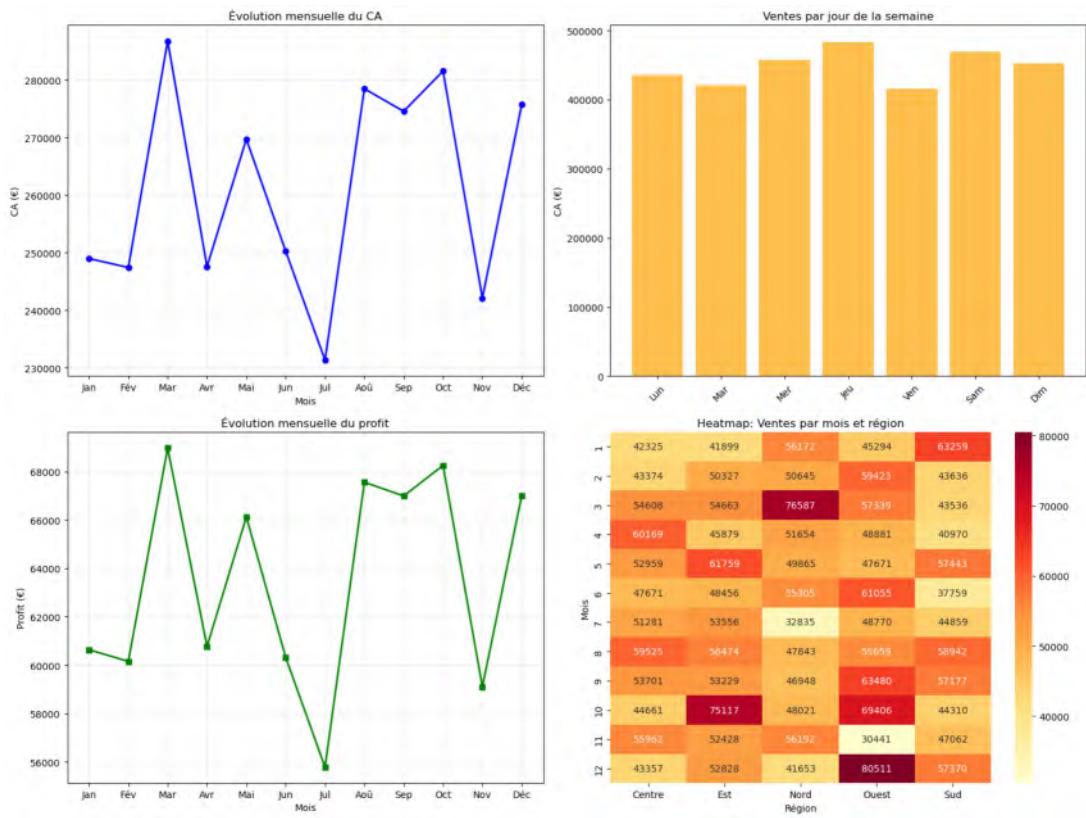
# Analyse de la relation prix-satisfaction
sns.scatterplot(data=df_clean, x='prix_unitaire', y='satisfaction_client',
                 hue='categorie', alpha=0.6, ax=axes[1])
axes[1].set_title('Relation Prix-Satisfaction par catégorie')
axes[1].set_xlabel('Prix unitaire (€)')
axes[1].set_ylabel('Satisfaction client (/5)')

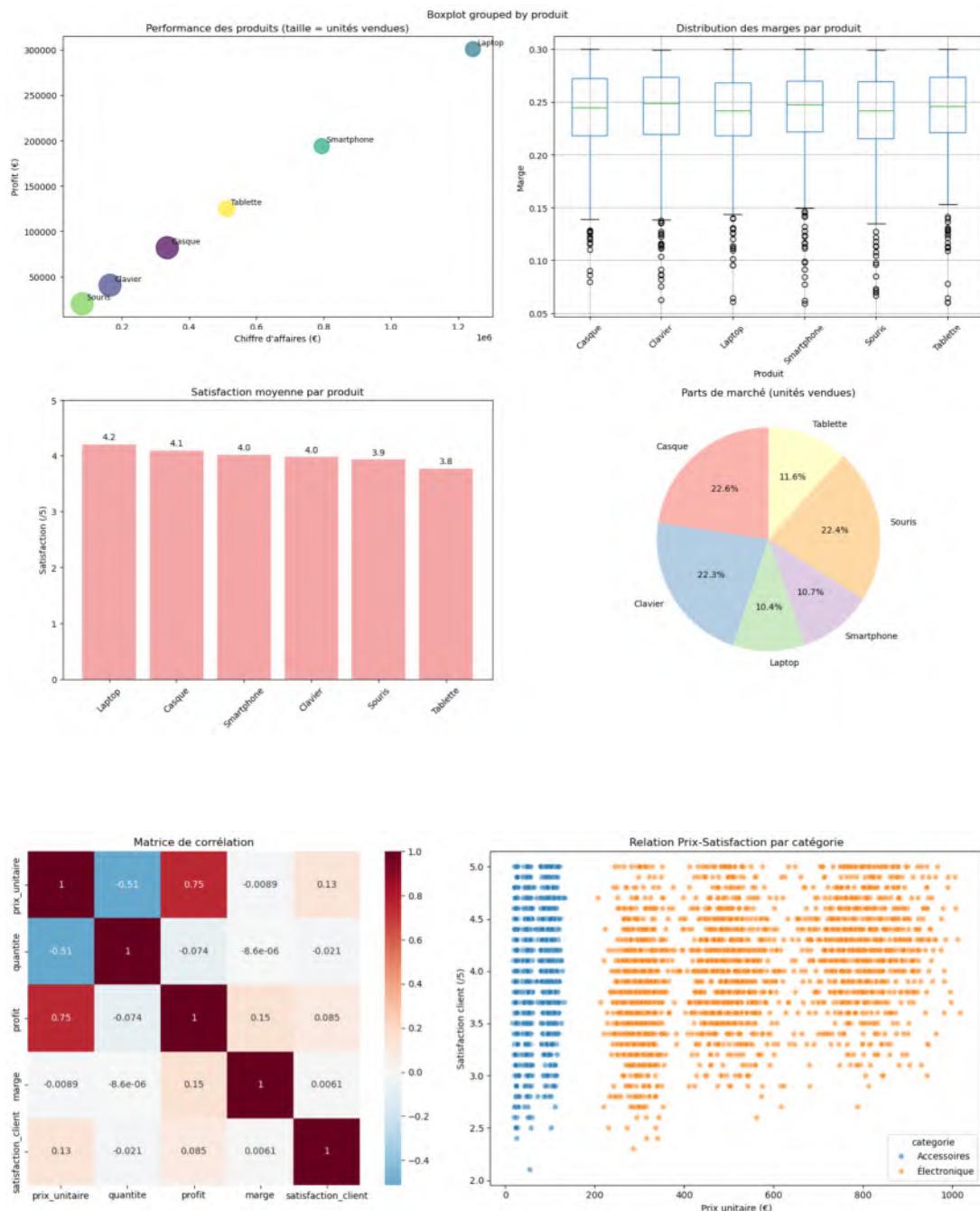
plt.tight_layout()

```

```
plt.show()
```







Phase 5 : Insights et recommandations

```
[72]: print("="*60)
print("RAPPORT D'ANALYSE - INSIGHTS ET RECOMMANDATIONS")
print("="*60)
```

```

# 1. Performance globale
print("\n1. PERFORMANCE GLOBALE")
print("-" * 30)
total_ca = df_clean['prix_apres_remise'].sum()
total_profit = df_clean['profit'].sum()
marge_globale = total_profit / total_ca
print(f"• Chiffre d'affaires: {total_ca:,.0f} €")
print(f"• Profit total: {total_profit:,.0f} €")
print(f"• Marge globale: {marge_globale:.1%}")
print(f"• Satisfaction moyenne: {df_clean['satisfaction_client'].mean():.1f}/5")

# 2. Analyse par région
print("\n2. ANALYSE RÉGIONALE")
print("-" * 30)
region_perf = df_clean.groupby('region').agg({
    'prix_apres_remise': 'sum',
    'profit': 'sum',
    'satisfaction_client': 'mean'
})
region_perf['marge'] = region_perf['profit'] / region_perf['prix_apres_remise']

best_region_ca = region_perf['prix_apres_remise'].idxmax()
best_region_marge = region_perf['marge'].idxmax()
best_region_satisfaction = region_perf['satisfaction_client'].idxmax()

print(f"• Meilleure région (CA): {best_region_ca}")
print(f"• Meilleure région (marge): {best_region_marge}")
print(f"• Meilleure région (satisfaction): {best_region_satisfaction}")

# 3. Analyse des produits
print("\n3. ANALYSE DES PRODUITS")
print("-" * 30)
product_perf = df_clean.groupby('produit').agg({
    'prix_apres_remise': 'sum',
    'profit': 'sum',
    'satisfaction_client': 'mean',
    'quantite': 'sum'
})
product_perf['marge'] = product_perf['profit'] / product_perf['prix_apres_remise']

top_product_ca = product_perf['prix_apres_remise'].idxmax()
top_product_marge = product_perf['marge'].idxmax()
top_product_volume = product_perf['quantite'].idxmax()

print(f"• Produit star (CA): {top_product_ca}")
print(f"• Produit le plus rentable (marge): {top_product_marge}")

```

```

print(f"• Produit le plus vendu (volume): {top_product_volume}")

# 4. Tendances temporelles
print("\n4. TENDANCES TEMPORELLES")
print("-" * 30)
monthly_trend = df_clean.groupby('mois')['prix_apres_remise'].sum()
best_month = monthly_trend.idxmax()
worst_month = monthly_trend.idxmin()
growth_rate = (monthly_trend.iloc[-1] - monthly_trend.iloc[0]) / monthly_trend.
    iloc[0]

print(f"• Meilleur mois: {months[best_month-1]} ({monthly_trend[best_month]:.
    0f} €)")
print(f"• Mois le plus faible: {months[worst_month-1]} ({monthly_trend[worst_month]:.
    0f} €)")
print(f"• Croissance annuelle: {growth_rate:.1%}")

# 5. Analyse des vendeurs
print("\n5. PERFORMANCE DES VENDEURS")
print("-" * 30)
vendeur_perf = df_clean.groupby('vendeur').agg({
    'prix_apres_remise': 'sum',
    'profit': 'sum',
    'satisfaction_client': 'mean'
}).sort_values('prix_apres_remise', ascending=False)

top_vendeur = vendeur_perf.index[0]
print(f"• Top vendeur (CA): {top_vendeur}")
print(f" - CA: {vendeur_perf.loc[top_vendeur, 'prix_apres_remise']:.0f} €")
print(f" - Profit: {vendeur_perf.loc[top_vendeur, 'profit']:.0f} €")
print(f" - Satisfaction: {vendeur_perf.loc[top_vendeur, 'satisfaction_client']:
    .1f}/5")

# 6. Recommandations
print("\n6. RECOMMANDATIONS STRATÉGIQUES")
print("-" * 40)

# Recommandations basées sur l'analyse
recommendations = []

# Régions
worst_region = region_perf['prix_apres_remise'].idxmin()
recommendations.append(f"• Région {worst_region}: Renforcer l'équipe
    commerciale")

# Produits

```

```

low_satisfaction_products = product_perf[product_perf['satisfaction_client'] <= 4.0]
if not low_satisfaction_products.empty:
    for product in low_satisfaction_products.index:
        recommendations.append(f"• Produit {product}: Améliorer la qualité/service")

# Saisonalité
if monthly_trend.std() > monthly_trend.mean() * 0.1:
    recommendations.append("• Forte saisonnalité détectée: Planifier les stocks et promotions")

# Corrélations
price_satisfaction_corr = df_clean['prix_unitaire'].corr(df_clean['satisfaction_client'])
if price_satisfaction_corr < -0.3:
    recommendations.append("• Corrélation négative prix-satisfaction: Revoir la stratégie tarifaire")

for rec in recommendations:
    print(rec)

print("\n7. MÉTRIQUES CLÉS À SURVEILLER")
print("-" * 40)
print("• Évolution mensuelle du CA par région")
print("• Taux de satisfaction par produit")
print("• Performance individuelle des vendeurs")
print("• Marge par catégorie de produit")
print("• Saisonalité des ventes")

print("\n" + "="*60)

```

=====

RAPPORT D'ANALYSE - INSIGHTS ET RECOMMANDATIONS

=====

1. PERFORMANCE GLOBALE

- Chiffre d'affaires: 3,134,178 €
- Profit total: 761,645 €
- Marge globale: 24.3%
- Satisfaction moyenne: 4.0/5

2. ANALYSE RÉGIONALE

- Meilleure région (CA): Ouest
- Meilleure région (marge): Sud

- Meilleure région (satisfaction): Nord

3. ANALYSE DES PRODUITS

- Produit star (CA): Laptop
- Produit le plus rentable (marge): Clavier
- Produit le plus vendu (volume): Casque

4. TENDANCES TEMPORELLES

- Meilleur mois: Mar (286,733 €)
- Mois le plus faible: Jul (231,300 €)
- Croissance annuelle: 10.8%

5. PERFORMANCE DES VENDEURS

- Top vendeur (CA): Vendeur_09
 - CA: 192,463 €
 - Profit: 46,729 €
 - Satisfaction: 4.0/5

6. RECOMMANDATIONS STRATÉGIQUES

- Région Sud: Renforcer l'équipe commerciale
- Produit Clavier: Améliorer la qualité/service
- Produit Souris: Améliorer la qualité/service
- Produit Tablette: Améliorer la qualité/service

7. MÉTRIQUES CLÉS À SURVEILLER

- Évolution mensuelle du CA par région
- Taux de satisfaction par produit
- Performance individuelle des vendeurs
- Marge par catégorie de produit
- Saisonnalité des ventes

1.5.2 Finalisation du projet

```
[74]: # Sauvegarde des résultats
print("== SAUVEGARDE DES RÉSULTATS ==")

# 1. Dataset nettoyé
df_clean.to_csv('donnees_ventes_clean.csv', index=False)
print("+ Dataset nettoyé sauvé: donnees_ventes_clean.csv")
```

```

# 2. Rapport de synthèse
summary_stats = {
    'Metrics_globales': {
        'CA_total': float(df_clean['prix_apres_remise'].sum()),
        'Profit_total': float(df_clean['profit'].sum()),
        'Marge_moyenne': float(df_clean['marge'].mean()),
        'Satisfaction_moyenne': float(df_clean['satisfaction_client'].mean())
    },
    'Performance_regions': region_perf.to_dict(),
    'Performance_produits': product_perf.to_dict(),
    'Tendances_mensuelles': monthly_trend.to_dict()
}

import json
with open('rapport_analyse.json', 'w', encoding='utf-8') as f:
    json.dump(summary_stats, f, indent=2, ensure_ascii=False)
print("+ Rapport d'analyse sauvégarde: rapport_analyse.json")

# 3. Résumé pour présentation
resume = f"""
RÉSUMÉ EXÉCUTIF - ANALYSE DES VENTES
=====

PERFORMANCE GLOBALE:
- Chiffre d'affaires: {total_ca:,.0f} €
- Profit: {total_profit:,.0f} €
- Marge: {marge_globale:.1%}
- Satisfaction: {df_clean['satisfaction_client'].mean():.1f}/5

POINTS CLÉS:
- Meilleure région: {best_region_ca}
- Produit star: {top_product_ca}
- Meilleur mois: {months[best_month-1]}
- Top vendeur: {top_vendeur}

RECOMMANDATIONS:
1. Renforcer la région {worst_region}
2. Améliorer la satisfaction des produits <4.0/5
3. Optimiser la gestion saisonnière
4. Former les équipes commerciales
"""

with open('resume_executif.txt', 'w', encoding='utf-8') as f:
    f.write(resume)
print("+ Résumé exécutif sauvégarde: resume_executif.txt")

print("\n==== PROJET TERMINÉ AVEC SUCCÈS ===")

```

```

print("Fichiers générés:")
print("1. donnees_ventes_clean.csv - Dataset nettoyé")
print("2. rapport_analyse.json - Données détaillées")
print("3. resume_executif.txt - Résumé pour direction")

==== SAUVEGARDE DES RÉSULTATS ===
+ Dataset nettoyé sauvégarde: donnees_ventes_clean.csv
+ Rapport d'analyse sauvégarde: rapport_analyse.json
+ Résumé exécutif sauvégarde: resume_executif.txt

==== PROJET TERMINÉ AVEC SUCCÈS ===
Fichiers générés:
1. donnees_ventes_clean.csv - Dataset nettoyé
2. rapport_analyse.json - Données détaillées
3. resume_executif.txt - Résumé pour direction

```

1.6 Introduction au Machine Learning

1.6.1 Introduction au ML : supervision, pipeline, scikit-learn

Le Machine Learning (apprentissage automatique) est une branche de l'intelligence artificielle qui permet aux ordinateurs d'apprendre et de faire des prédictions ou des décisions sans être explicitement programmés pour chaque tâche.

Concepts fondamentaux :

Types d'apprentissage

```

[76]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import load_iris, make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.metrics import accuracy_score, mean_squared_error,classification_report
from sklearn.pipeline import Pipeline

# Configuration
plt.rcParams['figure.figsize'] = (12, 8)
sns.set_style("whitegrid")

print("==== TYPES D'APPRENTISSAGE AUTOMATIQUE ====")

```

```

print("""
1. APPRENTISSAGE SUPERVISE
- Utilise des données étiquetées (X, y)
- Objectif: Prédire y à partir de X
- Types:
  * Classification: Prédire des catégories
  * Régression: Prédire des valeurs continues

2. APPRENTISSAGE NON SUPERVISE
- Utilise des données non étiquetées (X seulement)
- Objectif: Découvrir des structures cachées
- Types:
  * Clustering: Regrouper des observations similaires
  * Réduction de dimension: Simplifier les données

3. APPRENTISSAGE PAR RENFORCEMENT
- Apprend par essai-erreur
- Agent interagit avec environnement
- Maximise les récompenses
""")

# Exemple de données supervisées
print("\n==== EXEMPLE DE DONNÉES SUPERVISÉES ====")
# Dataset Iris pour classification
iris = load_iris()
X_iris, y_iris = iris.data, iris.target

print(f"Classification - Dataset Iris:")
print(f"Features (X): {iris.feature_names}")
print(f"Target (y): {iris.target_names}")
print(f"Forme: X={X_iris.shape}, y={y_iris.shape}")

# Visualisation
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Classification: Iris
df_iris = pd.DataFrame(X_iris, columns=iris.feature_names)
df_iris['species'] = y_iris
sns.scatterplot(data=df_iris, x='sepal length (cm)', y='sepal width (cm)',
                 hue='species', ax=axes[0])
axes[0].set_title('Classification: Prédire l\'espèce d\'iris')

# Régression: Relation prix-surface
np.random.seed(42)
surface = np.random.uniform(50, 200, 100)
prix = 2000 * surface + np.random.normal(0, 50000, 100)
axes[1].scatter(surface, prix, alpha=0.6)

```

```

axes[1].set_xlabel('Surface (m²)')
axes[1].set_ylabel('Prix (€)')
axes[1].set_title('Régression: Prédire le prix d\'un logement')

plt.tight_layout()
plt.show()

```

==== TYPES D'APPRENTISSAGE AUTOMATIQUE ====

1. APPRENTISSAGE SUPERVISÉ

- Utilise des données étiquetées (X, y)
- Objectif: Prédire y à partir de X
- Types:
 - * Classification: Prédire des catégories
 - * Régression: Prédire des valeurs continues

2. APPRENTISSAGE NON SUPERVISÉ

- Utilise des données non étiquetées (X seulement)
- Objectif: Découvrir des structures cachées
- Types:
 - * Clustering: Regrouper des observations similaires
 - * Réduction de dimension: Simplifier les données

3. APPRENTISSAGE PAR RENFORCEMENT

- Apprend par essai-erreur
- Agent interagit avec environnement
- Maximise les récompenses

==== EXEMPLE DE DONNÉES SUPERVISÉES ====

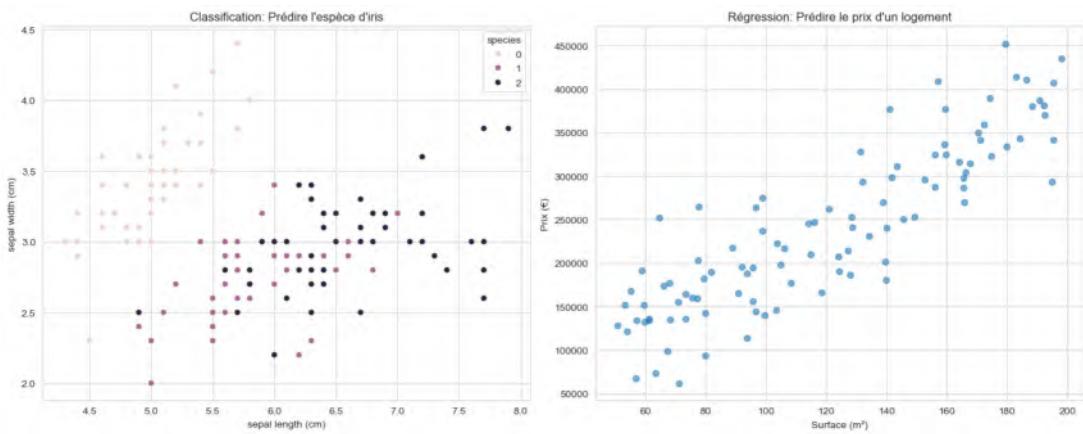
Classification - Dataset Iris:

Features (X): ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',

'petal width (cm)']

Target (y): ['setosa' 'versicolor' 'virginica']

Forme: X=(150, 4), y=(150,)



Introduction à Scikit-learn

```
[78]: print("== ECOSYSTÈME SCIKIT-LEARN ==")
print("""
Scikit-learn est LA bibliothèque de référence pour le ML en Python
```

ARCHITECTURE:

1. **Estimators:** Tous les algorithmes ML
 - `.fit(X, y)`: Entraîner le modèle
 - `.predict(X)`: Faire des prédictions
 - `.score(X, y)`: Évaluer la performance
2. **Transformers:** Préparation des données
 - `.fit(X)`: Apprendre les paramètres
 - `.transform(X)`: Appliquer la transformation
 - `.fit_transform(X)`: Fit + transform en une fois
3. **Pipelines:** Enchaîner les étapes
 - Preprocessing + Model
 - Évite les fuites de données

```
""")
```

```
# Exemple complet avec Scikit-learn
print("\n== EXEMPLE COMPLET: PIPELINE ML ==")

# 1. Chargement des données
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
X, y = cancer.data, cancer.target

print(f"Dataset: {cancer.feature_names[:5]}... (30 features)")
print(f"Classes: {cancer.target_names}")
```

```

print(f"Forme: {X.shape}")

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

print(f"Train: {X_train.shape}, Test: {X_test.shape}")

# 3. Pipeline complet
pipeline = Pipeline([
    ('scaler', StandardScaler()),           # Normalisation
    ('classifier', LogisticRegression())    # Classification
])

# 4. Entraînement
pipeline.fit(X_train, y_train)

# 5. Prédictions
y_pred = pipeline.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print(f"\nPREDICTIONS:")
print(f"Précision: {accuracy:.3f}")
print(f"Erreurs: {(y_test != y_pred).sum()}/{len(y_test)}")

# 6. Rapport détaillé
print(f"\nRAPPORT DÉTAILLÉ:")
print(classification_report(y_test, y_pred, target_names=cancer.target_names))

```

==== ÉCOSYSTÈME SCIKIT-LEARN ===

Scikit-learn est LA bibliothèque de référence pour le ML en Python

ARCHITECTURE:

1. Estimators: Tous les algorithmes ML
 - .fit(X, y): Entrainer le modèle
 - .predict(X): Faire des prédictions
 - .score(X, y): Évaluer la performance
2. Transformers: Préparation des données
 - .fit(X): Apprendre les paramètres
 - .transform(X): Appliquer la transformation
 - .fit_transform(X): Fit + transform en une fois
3. Pipelines: Enchaîner les étapes
 - Preprocessing + Model
 - Évite les fuites de données

```
== EXEMPLE COMPLET: PIPELINE ML ==
Dataset: ['mean radius' 'mean texture' 'mean perimeter' 'mean area'
'mean smoothness']... (30 features)
Classes: ['malignant' 'benign']
Forme: (569, 30)
Train: (455, 30), Test: (114, 30)
```

PREDICTIONS:

Précision: 0.982
Erreurs: 2/114

RAPPORT DÉTAILLÉ:

	precision	recall	f1-score	support
malignant	0.98	0.98	0.98	42
benign	0.99	0.99	0.99	72
accuracy			0.98	114
macro avg	0.98	0.98	0.98	114
weighted avg	0.98	0.98	0.98	114

Workflow ML standard

```
[80]: print("== WORKFLOW MACHINE LEARNING STANDARD ==")
```

```
def ml_workflow_demo():
    """Démonstration du workflow ML complet"""

    # 1. DÉFINITION DU PROBLÈME
    print("1. DÉFINITION DU PROBLÈME")
    print("  - Type: Classification binaire")
    print("  - Objectif: Diagnostic cancer du sein")
    print("  - Métrique: Précision (minimiser faux négatifs)")

    # 2. COLLECTE ET EXPLORATION DES DONNÉES
    print("\n2. EXPLORATION DES DONNÉES")
    cancer = load_breast_cancer()
    X, y = cancer.data, cancer.target

    print(f"  - Échantillons: {X.shape[0]}")
    print(f"  - Features: {X.shape[1]}")
    print(f"  - Classes: {np.unique(y, return_counts=True)}")

    # Vérifier les valeurs manquantes
    df = pd.DataFrame(X, columns=cancer.feature_names)
```

```

print(f" - Valeurs manquantes: {df.isnull().sum().sum()}")


# 3. PRÉPARATION DES DONNÉES
print("\n3. PRÉPARATION DES DONNÉES")

# Division des données
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

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

print(f" - Train set: {X_train.shape}")
print(f" - Test set: {X_test.shape}")
print(f" - Normalisation: Moyenne=0, Écart-type=1")


# 4. SÉLECTION ET ENTRAÎNEMENT DU MODÈLE
print("\n4. ENTRAÎNEMENT DU MODÈLE")

model = LogisticRegression(random_state=42)
model.fit(X_train_scaled, y_train)

print(f" - Algorithme: Régression Logistique")
print(f" - Paramètres: {model.get_params()}")


# 5. ÉVALUATION
print("\n5. ÉVALUATION")

train_score = model.score(X_train_scaled, y_train)
test_score = model.score(X_test_scaled, y_test)

print(f" - Score train: {train_score:.3f}")
print(f" - Score test: {test_score:.3f}")
print(f" - Surapprentissage: {'Non' if abs(train_score - test_score) < 0.
-05 else 'Possible'}")


# 6. PRÉDICTIONS SUR NOUVELLES DONNÉES
print("\n6. MISE EN PRODUCTION")

# Simulation de nouvelles données
new_sample = X_test_scaled[:3] # 3 nouveaux échantillons
predictions = model.predict(new_sample)
probabilities = model.predict_proba(new_sample)

```

```

print("  - Prédictions sur nouveaux échantillons:")
for i, (pred, prob) in enumerate(zip(predictions, probabilities)):
    diagnosis = cancer.target_names[pred]
    confidence = prob.max()
    print(f"    Échantillon {i+1}: {diagnosis} (confiance: {confidence:.2f})")

return model, scaler, X_test_scaled, y_test

# Exécuter la démonstration
model, scaler, X_test_scaled, y_test = ml_workflow_demo()

```

==== WORKFLOW MACHINE LEARNING STANDARD ====

1. DÉFINITION DU PROBLÈME

- Type: Classification binaire
- Objectif: Diagnostic cancer du sein
- Métrique: Précision (minimiser faux négatifs)

2. EXPLORATION DES DONNÉES

- Échantillons: 569
- Features: 30
- Classes: (array([0, 1]), array([212, 357], dtype=int64))
- Valeurs manquantes: 0

3. PRÉPARATION DES DONNÉES

- Train set: (455, 30)
- Test set: (114, 30)
- Normalisation: Moyenne=0, Écart-type=1

4. ENTRAÎNEMENT DU MODÈLE

- Algorithme: Régression Logistique
- Paramètres: {'C': 1.0, 'class_weight': None, 'dual': False, 'fit_intercept': True, 'intercept_scaling': 1, 'l1_ratio': None, 'max_iter': 100, 'multi_class': 'deprecated', 'n_jobs': None, 'penalty': 'l2', 'random_state': 42, 'solver': 'lbfgs', 'tol': 0.0001, 'verbose': 0, 'warm_start': False}

5. ÉVALUATION

- Score train: 0.989
- Score test: 0.982
- Surapprentissage: Non

6. MISE EN PRODUCTION

- Prédictions sur nouveaux échantillons:
 - Échantillon 1: malignant (confiance: 1.00)
 - Échantillon 2: benign (confiance: 1.00)
 - Échantillon 3: malignant (confiance: 0.99)

1.6.2 Régression linéaire

La régression linéaire est l'un des algorithmes les plus fondamentaux du machine learning. Elle modélise la relation entre une variable dépendante et une ou plusieurs variables indépendantes en ajustant une ligne droite (ou un hyperplan) aux données.

Théorie et mathématiques

```
[82]: print("== RÉGRESSION LINÉAIRE - THÉORIE ==")
print("""
ÉQUATION MATHÉMATIQUE:
- Simple:  $y = \beta_0 + \beta_1 x + \epsilon$ 
- Multiple:  $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n + \epsilon$ 

Où:
-  $y$ : Variable dépendante (target)
-  $x$ : Variables indépendantes (features)
-  $\beta_0$ : Intercept (ordonnée à l'origine)
-  $\beta_1, \beta_2, \dots$ : Coefficients (pentes)
-  $\epsilon$ : Erreur/bruit

OBJECTIF:
Minimiser la somme des carrés des erreurs (MSE)
MSE =  $\frac{1}{n} \times \sum (y_{réel} - y_{prédict})^2$ 
""")

# Génération de données synthétiques
np.random.seed(42)
n_samples = 100
X_simple = np.random.uniform(0, 10, n_samples)
noise = np.random.normal(0, 2, n_samples)
y_simple = 2 * X_simple + 5 + noise  #  $y = 2x + 5 + \text{bruit}$ 

# Visualisation des données
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# 1. Données originales
axes[0].scatter(X_simple, y_simple, alpha=0.6, color='blue')
axes[0].set_xlabel('X')
axes[0].set_ylabel('y')
axes[0].set_title('Données avec relation linéaire + bruit')
axes[0].grid(True, alpha=0.3)

# 2. Ajustement manuel de la droite
X_line = np.linspace(0, 10, 100)
y_true = 2 * X_line + 5  # Relation vraie
y_manual = 1.8 * X_line + 6  # Ajustement manuel

axes[1].scatter(X_simple, y_simple, alpha=0.6, color='blue', label='Données')
```

```

axes[1].plot(X_line, y_true, 'g-', linewidth=2, label='Relation vraie')
axes[1].plot(X_line, y_manual, 'r--', linewidth=2, label='Ajustement manuel')
axes[1].set_xlabel('X')
axes[1].set_ylabel('y')
axes[1].set_title('Comparaison ajustements')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

# 3. Régression avec scikit-learn
X_simple_reshaped = X_simple.reshape(-1, 1)
model_simple = LinearRegression()
model_simple.fit(X_simple_reshaped, y_simple)

y_pred_sklearn = model_simple.predict(X_line.reshape(-1, 1))

axes[2].scatter(X_simple, y_simple, alpha=0.6, color='blue', label='Données')
axes[2].plot(X_line, y_pred_sklearn, 'orange', linewidth=2, u
             ↪label='Scikit-learn')
axes[2].set_xlabel('X')
axes[2].set_ylabel('y')
axes[2].set_title('Régression avec Scikit-learn')
axes[2].legend()
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Affichage des paramètres appris
print(f"\nPARAMÈTRES APPRIS:")
print(f"Coeficient (pente): {model_simple.coef_[0]:.3f}")
print(f"Intercept: {model_simple.intercept_:.3f}")
print(f"Équation: y = {model_simple.coef_[0]:.3f}x + {model_simple.intercept_:.3f}")
print(f"Relation vraie: y = 2.000x + 5.000")

```

==== RÉGRESSION LINÉAIRE - THÉORIE ===

ÉQUATION MATHÉMATIQUE:

- Simple: $y = \beta_0 + \beta_1 x + \epsilon$
- Multiple: $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n + \epsilon$

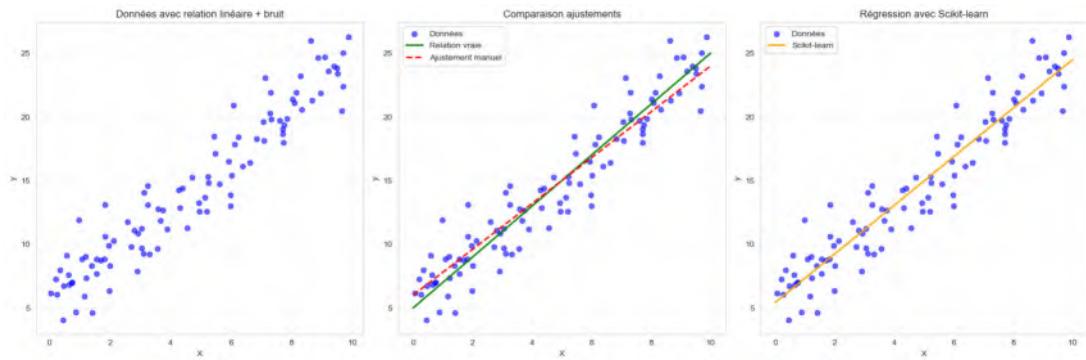
Où:

- y : Variable dépendante (target)
- x : Variables indépendantes (features)
- β_0 : Intercept (ordonnée à l'origine)
- β_1, β_2, \dots : Coefficients (pentes)
- ϵ : Erreur/bruit

OBJECTIF:

Minimiser la somme des carrés des erreurs (MSE)

$$MSE = \frac{1}{n} \times \sum (y_{\text{réel}} - y_{\text{prédict}})^2$$



PARAMÈTRES APPRIS:

Coefficient (pente): 1.908

Intercept: 5.430

Équation: $y = 1.908x + 5.430$

Relation vraie: $y = 2.000x + 5.000$

Régression linéaire multiple

```
[84]: print("\n==== RÉGRESSION LINÉAIRE MULTIPLE ===")
```

```
# Imports nécessaires
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Création du dataset
X_multi, y_multi = make_regression(
    n_samples=200,
    n_features=3,
    noise=10,
    random_state=42
)
```

```

# Ajout de noms de colonnes parlants
feature_names = ['Surface (m²)', 'Nombre de chambres', 'Âge (années)']
df_housing = pd.DataFrame(X_multi, columns=feature_names)
df_housing['Prix (k€)'] = y_multi

print("Dataset immobilier synthétique:")
print(df_housing.head())
print(f"\nStatistiques:\n{df_housing.describe()}")


# Visualisation des relations
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

for i, feature in enumerate(feature_names):
    row, col = i // 2, i % 2
    axes[row, col].scatter(df_housing[feature], df_housing['Prix (k€)'],  

    alpha=0.6)
    axes[row, col].set_xlabel(feature)
    axes[row, col].set_ylabel('Prix (k€)')
    axes[row, col].set_title(f'Prix vs {feature}')

    z = np.polyfit(df_housing[feature], df_housing['Prix (k€)'], 1)
    p = np.poly1d(z)
    axes[row, col].plot(df_housing[feature], p(df_housing[feature]), "r--",  

    alpha=0.8)

# Matrice de corrélation
corr_matrix = df_housing.corr()
sns.heatmap(corr_matrix, annot=True, cmap='RdBu_r', center=0, ax=axes[1, 1])
axes[1, 1].set_title('Matrice de corrélation')

plt.tight_layout()
plt.show()

# Entraînement du modèle
X_train, X_test, y_train, y_test = train_test_split(
    X_multi, y_multi, test_size=0.2, random_state=42
)

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

y_train_pred = model_multi.predict(X_train)
y_test_pred = model_multi.predict(X_test)

train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

```

```

train_r2 = model_multi.score(X_train, y_train)
test_r2 = model_multi.score(X_test, y_test)

print(f"\nRÉSULTATS DU MODÈLE:")
print(f"R2 Train: {train_r2:.3f}")
print(f"R2 Test: {test_r2:.3f}")
print(f"MAE Train: {train_mse:.2f}")
print(f"MAE Test: {test_mse:.2f}")

print(f"\nCOEFFICIENTS:")
print(f"Intercept: {model_multi.intercept_:.2f}")
for i, coef in enumerate(model_multi.coef_):
    print(f"{feature_names[i]}: {coef:.2f}")

# Visualisation des prédictions
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

axes[0].scatter(y_train, y_train_pred, alpha=0.6, color='blue')
axes[0].plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()], 'r--', lw=2)
axes[0].set_xlabel('Prix réel')
axes[0].set_ylabel('Prix prédict')
axes[0].set_title(f'Train Set (R2 = {train_r2:.3f})')
axes[0].grid(True, alpha=0.3)

axes[1].scatter(y_test, y_test_pred, alpha=0.6, color='green')
axes[1].plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'r--', lw=2)
axes[1].set_xlabel('Prix réel')
axes[1].set_ylabel('Prix prédict')
axes[1].set_title(f'Test Set (R2 = {test_r2:.3f})')
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

```

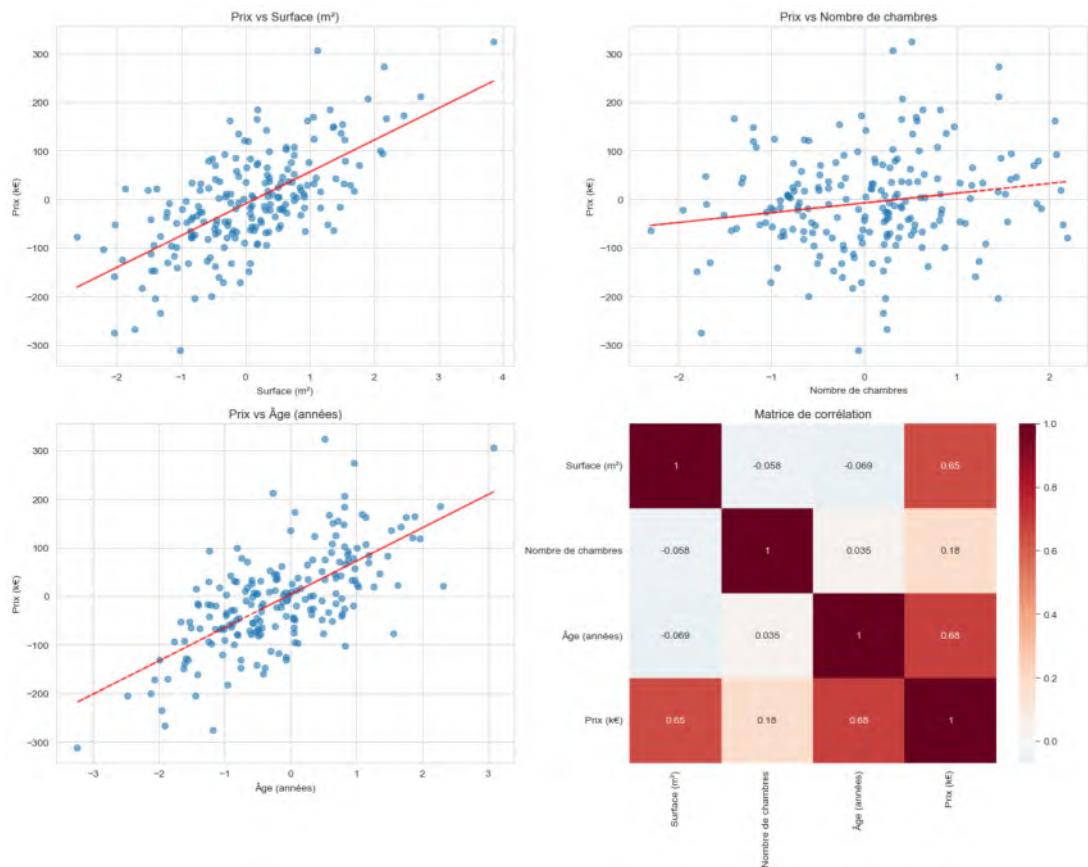
==== RÉGRESSION LINÉAIRE MULTIPLE ===

Dataset immobilier synthétique:

	Surface (m ²)	Nombre de chambres	Âge (années)	Prix (k€)
0	0.130741	-1.430141	-0.440044	-64.558855
1	1.502357	-0.269407	0.717542	154.776889
2	0.647689	0.496714	-0.138264	61.896041
3	0.341756	-0.759133	0.150394	16.485672
4	0.856399	-1.514847	-0.446515	-31.907412

Statistiques:

	Surface (m ²)	Nombre de chambres	Âge (années)	Prix (k€)
count	200.000000	200.000000	200.000000	200.000000
mean	0.043873	0.062469	-0.146899	-6.547163
std	1.006211	0.893358	1.003705	101.162943
min	-2.619745	-2.301921	-3.241267	-311.517807
25%	-0.594139	-0.602276	-0.856352	-66.574140
50%	0.069183	0.106376	-0.208246	-10.925565
75%	0.640866	0.563311	0.636783	45.018658
max	3.852731	2.189803	3.078881	323.840597



RÉSULTATS DU MODÈLE:

R² Train: 0.990

R² Test: 0.985

MSE Train: 103.22

MSE Test: 132.62

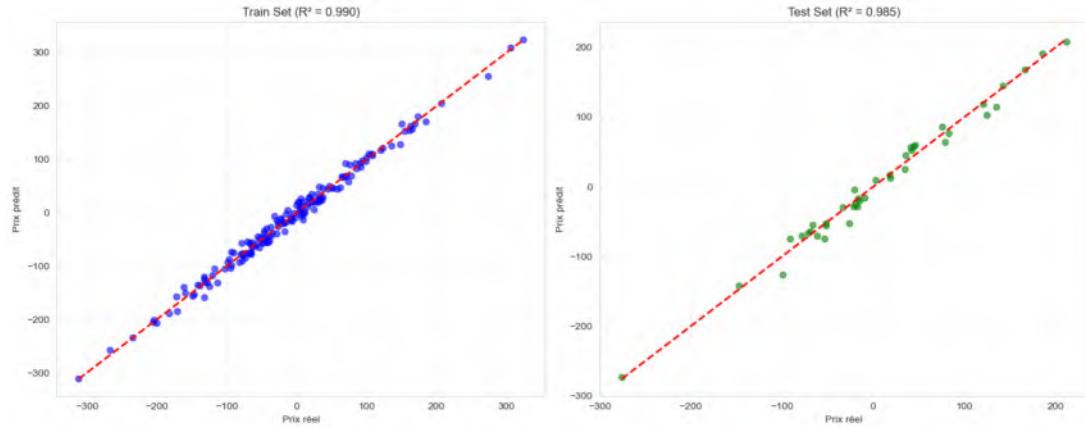
COEFFICIENTS:

Intercept: -0.64

Surface (m²): 71.89

Nombre de chambres: 22.54

Âge (années): 72.48



Analyse des résidus et diagnostics

```
[86]: print("== ANALYSE DES RÉSIDUS ==")
```

```
# Calcul des résidus
residus_train = y_train - y_train_pred
residus_test = y_test - y_test_pred

# Visualisation des résidus
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# 1. Résidus vs prédictions
axes[0, 0].scatter(y_train_pred, residus_train, alpha=0.6, color='blue')
axes[0, 0].axhline(y=0, color='red', linestyle='--')
axes[0, 0].set_xlabel('Prédictions')
axes[0, 0].set_ylabel('Résidus')
axes[0, 0].set_title('Résidus vs Prédictions (Train)')
axes[0, 0].grid(True, alpha=0.3)

# 2. Distribution des résidus
axes[0, 1].hist(residus_train, bins=20, alpha=0.7, color='skyblue', edgecolor='black')
axes[0, 1].set_xlabel('Résidus')
axes[0, 1].set_ylabel('Fréquence')
axes[0, 1].set_title('Distribution des résidus')
axes[0, 1].grid(True, alpha=0.3)

# 3. QQ-plot pour la normalité
from scipy import stats
```

```

stats.probplot(residus_train, dist="norm", plot=axes[1, 0])
axes[1, 0].set_title('Q-Q Plot (Normalité des résidus)')
axes[1, 0].grid(True, alpha=0.3)

# 4. Résidus par feature
feature_idx = 0 # Surface
axes[1, 1].scatter(X_train[:, feature_idx], residus_train, alpha=0.6, color='orange')
axes[1, 1].axhline(y=0, color='red', linestyle='--')
axes[1, 1].set_xlabel(feature_names[feature_idx])
axes[1, 1].set_ylabel('Résidus')
axes[1, 1].set_title(f'Résidus vs {feature_names[feature_idx]}')
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Tests statistiques
print("\nTESTS STATISTIQUES:")

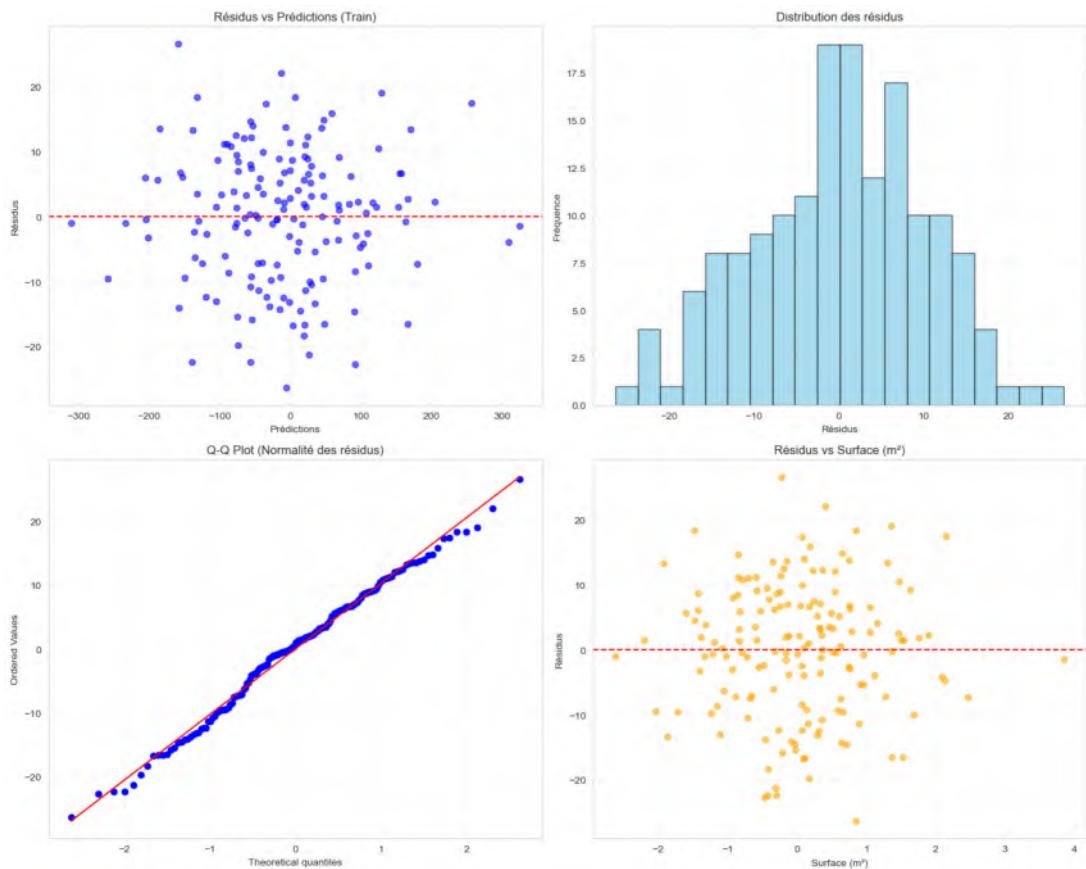
# Test de normalité des résidus
shapiro_stat, shapiro_p = stats.shapiro(residus_train)
print(f"Test de Shapiro-Wilk (normalité): p-value = {shapiro_p:.4f}")
print(f"Résidus normaux: {'Oui' if shapiro_p > 0.05 else 'Non'}")

# Durbin-Watson (autocorrélation)
from statsmodels.stats.stattools import durbin_watson # Correction ici
dw_stat = durbin_watson(residus_train)
print(f"Durbin-Watson (autocorrélation): {dw_stat:.3f}")
print(f"Autocorrélation: {'Faible' if 1.5 < dw_stat < 2.5 else 'Présente'}")

# Variance des résidus
print(f"Variance des résidus: {np.var(residus_train):.2f}")

```

==== ANALYSE DES RÉSIDUS ===



TESTS STATISTIQUES:

Test de Shapiro-Wilk (normalité): p-value = 0.4573

Résidus normaux: Oui

Durbin-Watson (autocorrélation): 2.117

Autocorrélation: Faible

Variance des résidus: 103.22

1.6.3 Validation croisée, biais-variance

La validation croisée est une technique fondamentale pour évaluer la performance d'un modèle de manière robuste et comprendre le compromis biais-variance.

Validation croisée

```
[88]: print("== VALIDATION CROISÉE ==")
```

```
from sklearn.model_selection import cross_val_score, KFold, StratifiedKFold
from sklearn.model_selection import validation_curve, learning_curve

# Exemple avec le dataset de cancer du sein
```

```

cancer = load_breast_cancer()
X, y = cancer.data, cancer.target

print("PRINCIPE DE LA VALIDATION CROISÉE:")
print("1. Diviser les données en K plis (folds)")
print("2. Pour chaque pli:")
print("    - Entraîner sur K-1 plis")
print("    - Tester sur le pli restant")
print("3. Moyenner les scores")

# Validation croisée simple
cv_scores = cross_val_score(LogisticRegression(), X, y, cv=5)
print(f"\nValidation croisée 5-fold:")
print(f"Scores: {cv_scores}")
print(f"Moyenne: {cv_scores.mean():.3f} ± {cv_scores.std():.3f}")

# Différents types de validation croisée
print("\nDIFFÉRENTS TYPES DE CV:")

# 1. K-Fold standard
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
scores_kfold = cross_val_score(LogisticRegression(), X, y, cv=kfold)

# 2. Stratified K-Fold (maintient les proportions de classes)
stratified = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores_stratified = cross_val_score(LogisticRegression(), X, y, cv=stratified)

# 3. Leave-One-Out (LOO)
from sklearn.model_selection import LeaveOneOut
loo = LeaveOneOut()
scores_loo = cross_val_score(LogisticRegression(), X[:100], y[:100], cv=loo) # ↳Sous-échantillon

print(f"K-Fold: {scores_kfold.mean():.3f} ± {scores_kfold.std():.3f}")
print(f"Stratified K-Fold: {scores_stratified.mean():.3f} ± {scores_stratified.std():.3f}")
print(f"Leave-One-Out: {scores_loo.mean():.3f} ± {scores_loo.std():.3f}")

# Visualisation de la validation croisée
def plot_cv_indices(cv, X, y, group=None, ax=None, n_splits=None, lw=10):
    """Visualise les indices de train/test pour différents CV"""
    if ax is None:
        fig, ax = plt.subplots()

    n_splits = cv.get_n_splits() if n_splits is None else n_splits

    for ii, (tr, tt) in enumerate(cv.split(X=X, y=y, groups=group)):

```

```

    indices = np.arange(len(y))

    # Train
    ax.scatter(tr, [ii] * len(tr), c='blue', marker='_', lw=lw, u
    ↪label='Train' if ii == 0 else "")
    # Test
    ax.scatter(tt, [ii] * len(tt), c='red', marker='_', lw=lw, label='Test' u
    ↪if ii == 0 else "")

    ax.set_xlabel("Indice des échantillons")
    ax.set_ylabel("Fold")
    ax.set_title("Visualisation de la validation croisée")
    ax.legend()
    return ax

# Visualisation pour un petit dataset
X_small, y_small = X[:50], y[:50]
fig, axes = plt.subplots(2, 1, figsize=(12, 8))

plot_cv_indices(KFold(n_splits=5), X_small, y_small, ax=axes[0])
axes[0].set_title("K-Fold (5 splits)")

plot_cv_indices(StratifiedKFold(n_splits=5), X_small, y_small, ax=axes[1])
axes[1].set_title("Stratified K-Fold (5 splits)")

plt.tight_layout()
plt.show()

```

==== VALIDATION CROISÉE ===

PRINCIPE DE LA VALIDATION CROISÉE:

1. Diviser les données en K plis (folds)
2. Pour chaque pli:
 - Entraîner sur K-1 plis
 - Tester sur le pli restant
3. Moyenner les scores

Validation croisée 5-fold:

Scores: [0.92982456 0.93859649 0.96491228 0.94736842 0.95575221]

Moyenne: 0.947 ± 0.012

DIFFÉRENTS TYPES DE CV:

K-Fold: 0.944 ± 0.027

Stratified K-Fold: 0.944 ± 0.020

Leave-One-Out: 0.930 ± 0.255



Compromis biais-variance

```
[90]: print("\n==== COMPROMIS BIAIS-VARIANCE ===")
```

```
print("""
```

CONCEPTS CLÉS:

BIAIS:

- Erreur due à des hypothèses simplificatrices
- Modèle trop simple → Sous-apprentissage
- Haute précision mais faible justesse

VARIANCE:

- Sensibilité aux variations dans les données d'entraînement
- Modèle trop complexe → Surapprentissage
- Haute justesse mais faible précision

COMPROMIS:

- Augmenter complexité → ↓ Biais, ↑ Variance
 - Diminuer complexité → ↑ Biais, ↓ Variance
 - Objectif: Trouver l'équilibre optimal
- """)

```
# Démonstration avec régression polynomiale
```

```

np.random.seed(42)
n_samples = 100
X_demo = np.random.uniform(0, 1, n_samples)
noise = np.random.normal(0, 0.1, n_samples)
y_demo = np.sin(2 * np.pi * X_demo) + noise # Fonction sinusoïdale + bruit

# Test avec différents degrés de polynômes
degrees = [1, 2, 5, 10, 15]
X_demo_reshaped = X_demo.reshape(-1, 1)

fig, axes = plt.subplots(2, 3, figsize=(18, 12))
axes = axes.ravel()

X_plot = np.linspace(0, 1, 100).reshape(-1, 1)
y_true = np.sin(2 * np.pi * X_plot.ravel())

for i, degree in enumerate(degrees):
    # Création des features polynomiales
    from sklearn.preprocessing import PolynomialFeatures
    poly_features = PolynomialFeatures(degree=degree)
    X_poly = poly_features.fit_transform(X_demo_reshaped)
    X_plot_poly = poly_features.transform(X_plot)

    # Entraînement
    model = LinearRegression()
    model.fit(X_poly, y_demo)

    # Prédictions
    y_plot_pred = model.predict(X_plot_poly)

    # Visualisation
    axes[i].scatter(X_demo, y_demo, alpha=0.6, color='blue', s=20, label='Données')
    axes[i].plot(X_plot, y_true, 'g-', linewidth=2, label='Vraie fonction')
    axes[i].plot(X_plot, y_plot_pred, 'r-', linewidth=2, label=f'Prédition degré {degree}')
    axes[i].set_title(f'Polynôme degré {degree}')
    axes[i].set_xlabel('X')
    axes[i].set_ylabel('y')
    axes[i].legend()
    axes[i].grid(True, alpha=0.3)

    # Calcul du score
    score = model.score(X_poly, y_demo)
    axes[i].text(0.05, 0.95, f'R² = {score:.3f}', transform=axes[i].transAxes,
                bbox=dict(boxstyle="round", facecolor='white', alpha=0.8))

```

```

# Dernière subplot: courbe de validation
axes[5].remove()
ax_val = fig.add_subplot(2, 3, 6)

# Courbe de validation pour différents degrés
degrees_range = range(1, 16)
train_scores = []
val_scores = []

for degree in degrees_range:
    poly_features = PolynomialFeatures(degree=degree)
    X_poly = poly_features.fit_transform(X_demo_reshaped)

    # Validation croisée
    model = LinearRegression()
    scores = cross_val_score(model, X_poly, y_demo, cv=5)

    # Score d'entraînement
    model.fit(X_poly, y_demo)
    train_score = model.score(X_poly, y_demo)

    train_scores.append(train_score)
    val_scores.append(scores.mean())

ax_val.plot(degrees_range, train_scores, 'o-', color='blue', label='Score\u20d7train')
ax_val.plot(degrees_range, val_scores, 'o-', color='red', label='Score\u20d7validation')
ax_val.set_xlabel('Degré du polynôme')
ax_val.set_ylabel('Score R2')
ax_val.set_title('Courbe de validation')
ax_val.legend()
ax_val.grid(True, alpha=0.3)

# Annotations
best_degree = degrees_range[np.argmax(val_scores)]
ax_val.axvline(x=best_degree, color='green', linestyle='--', alpha=0.7)
ax_val.text(best_degree + 0.5, 0.1, f'Optimal: {best_degree}', rotation=90,
            bbox=dict(boxstyle="round", facecolor='lightgreen', alpha=0.8))

plt.tight_layout()
plt.show()

print(f"\nRÉSULTATS:")
print(f"Degré optimal: {best_degree}")
print(f"Score de validation maximal: {max(val_scores):.3f}")

```

==== COMPROMIS BIAIS-VARIANCE ===

CONCEPTS CLÉS:

BIAIS:

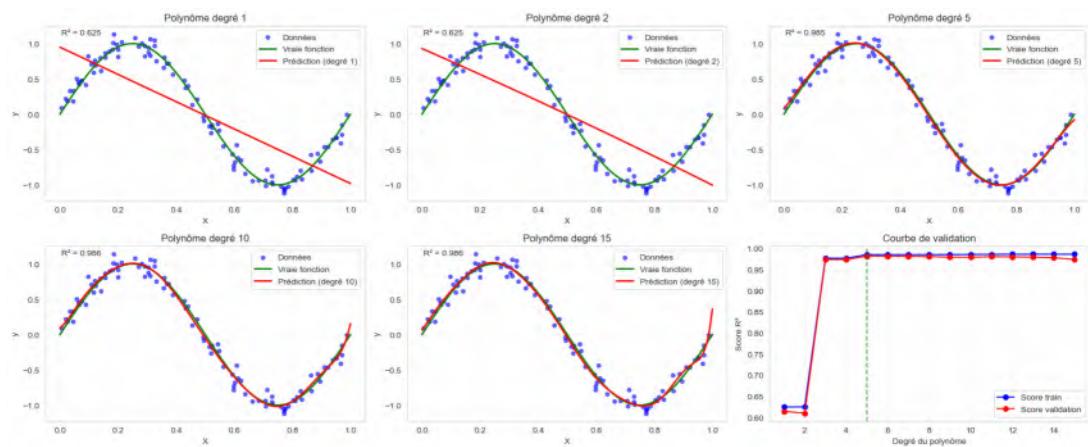
- Erreur due à des hypothèses simplificatrices
- Modèle trop simple → Sous-apprentissage
- Haute précision mais faible justesse

VARIANCE:

- Sensibilité aux variations dans les données d'entraînement
- Modèle trop complexe → Surapprentissage
- Haute justesse mais faible précision

COMPROMIS:

- Augmenter complexité → ↓ Biais, ↑ Variance
- Diminuer complexité → ↑ Biais, ↓ Variance
- Objectif: Trouver l'équilibre optimal



RÉSULTATS:

Degré optimal: 5

Score de validation maximal: 0.982

Courbes d'apprentissage

```
[92]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import learning_curve
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

print("\n==== COURBES D'APPRENTISSAGE ====")

# Fonction pour tracer les courbes d'apprentissage
def plot_learning_curve(estimator, X, y, title="Courbe d'apprentissage"):
    """Trace les courbes d'apprentissage"""
    train_sizes, train_scores, val_scores = learning_curve(
        estimator, X, y, cv=5, n_jobs=-1,
        train_sizes=np.linspace(0.1, 1.0, 10),
        random_state=42
    )

    train_mean = np.mean(train_scores, axis=1)
    train_std = np.std(train_scores, axis=1)
    val_mean = np.mean(val_scores, axis=1)
    val_std = np.std(val_scores, axis=1)

    plt.figure(figsize=(10, 6))
    plt.plot(train_sizes, train_mean, 'o-', color='blue', label='Score train')
    plt.fill_between(train_sizes, train_mean - train_std, train_mean + val_std, alpha=0.1, color='blue')

    plt.plot(train_sizes, val_mean, 'o-', color='red', label='Score validation')
    plt.fill_between(train_sizes, val_mean - val_std, val_mean + val_std, alpha=0.1, color='red')

    plt.xlabel('Taille du set d\'entraînement')
    plt.ylabel('Score')
    plt.title(title)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

    return train_sizes, train_scores, val_scores

# Comparaison de différents modèles
print("Comparaison des courbes d'apprentissage:")

results_kernels = {}
```

```

# 1. Régression logistique (modèle simple)
print("\n1. Régression Logistique (faible variance)")
results_kernels['LogisticRegression'] =_
    plot_learning_curve(LogisticRegression(), X, y, "Régression Logistique")

# 2. Arbre de décision (modèle complexe)
print("\n2. Arbre de Décision (forte variance)")
results_kernels['DecisionTree'] =_
    plot_learning_curve(DecisionTreeClassifier(random_state=42), X, y, "Arbre de_"
    "Décision")

# 3. Random Forest ( compromis)
print("\n3. Random Forest ( compromis)")
results_kernels['RandomForest'] =_
    plot_learning_curve(RandomForestClassifier(random_state=42), X, y, "Random_"
    "Forest")

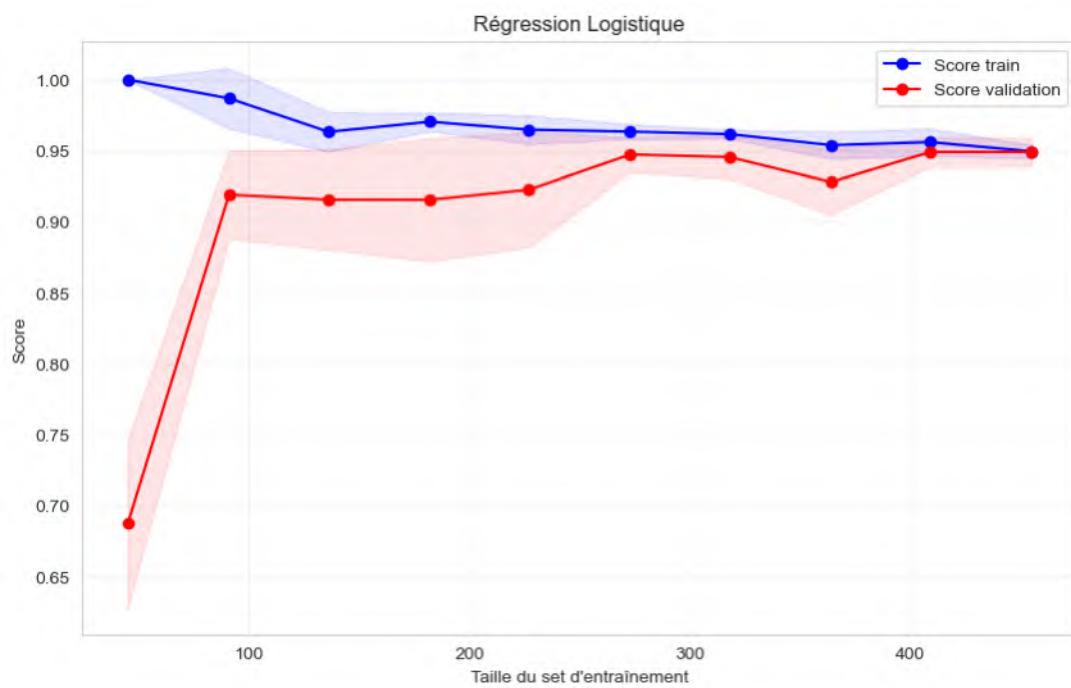
# Résumé des scores moyens finaux pour chaque modèle
df_kernels = pd.DataFrame({
    model: {
        'train_score_mean_last': np.mean(results_kernels[model][1][:, -1]),
        'val_score_mean_last': np.mean(results_kernels[model][2][:, -1])
    }
    for model in results_kernels
}).T

print("\nRésumé des scores moyens finaux (à la plus grande taille_"
    "d'entraînement) :")
print(df_kernels.round(3))

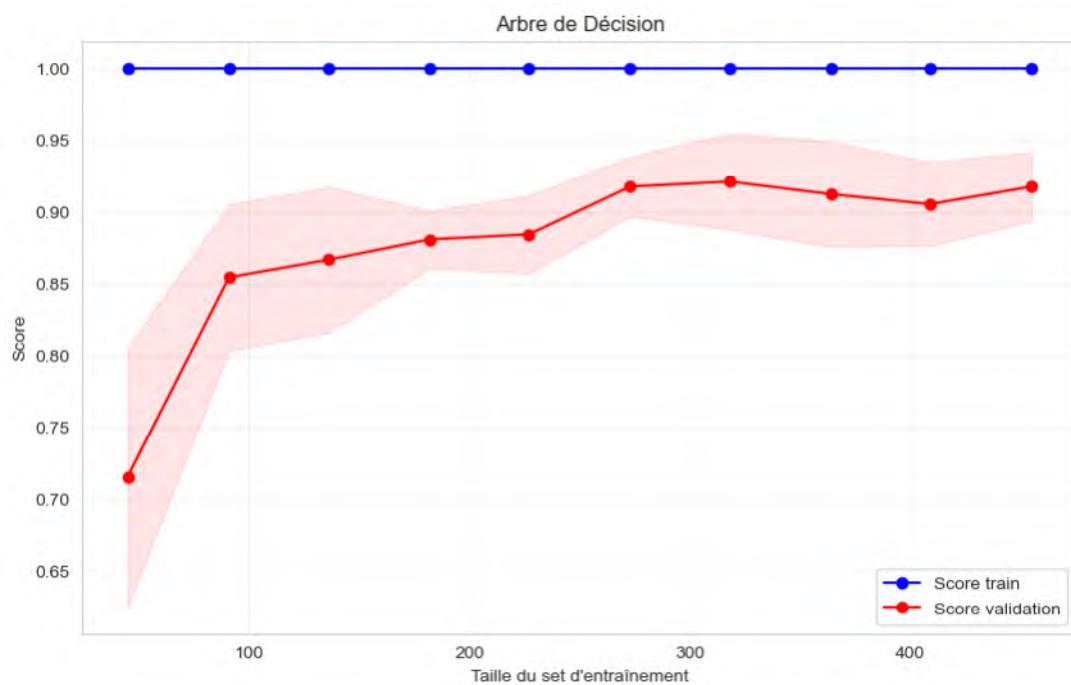
```

==== COURBES D'APPRENTISSAGE ====
Comparaison des courbes d'apprentissage:

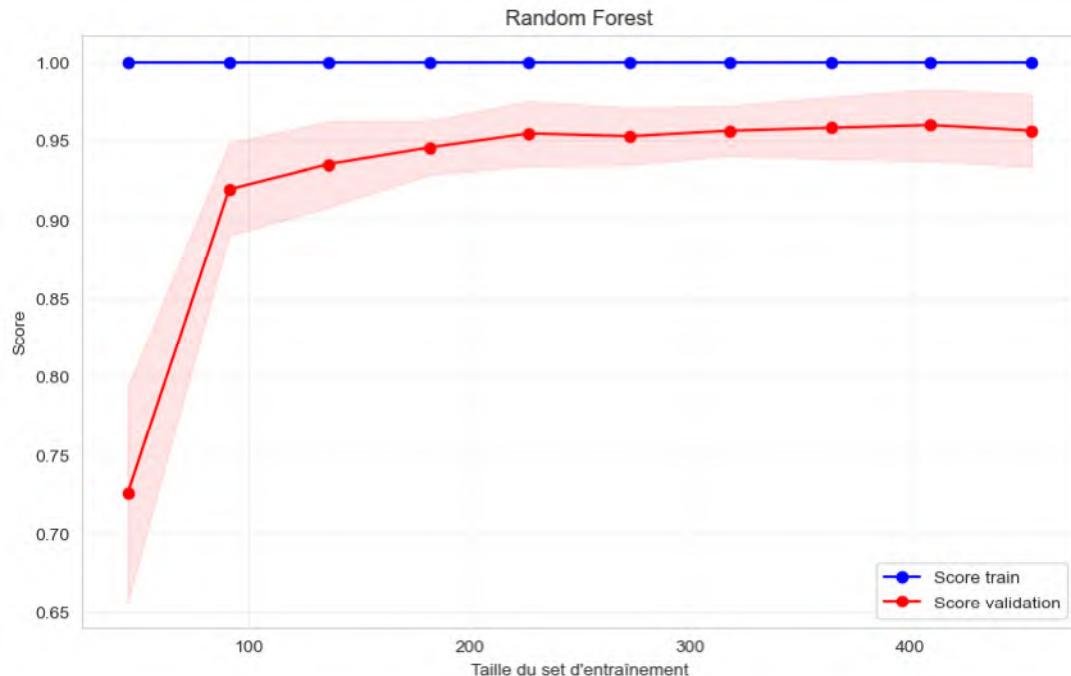
1. Régression Logistique (faible variance)



2. Arbre de Décision (forte variance)



3. Random Forest (compromis)



Résumé des scores moyens finaux (à la plus grande taille d'entraînement) :

	train_score_mean_last	val_score_mean_last
LogisticRegression	0.969	0.888
DecisionTree	1.000	0.826
RandomForest	1.000	0.919

1.6.4 Régression logistique

La régression logistique est un algorithme de classification qui utilise la fonction logistique (sigmoïde) pour modéliser la probabilité qu'un échantillon appartienne à une classe particulière.

Théorie mathématique

```
[94]: print("== RÉGRESSION LOGISTIQUE - THÉORIE ==")  
  
print("")  
PRINCIPE:  
La régression logistique modélise la probabilité  $P(y=1|x)$  avec la fonction  
sigmoïde:
```

```

P(y=1|x) = 1 / (1 + e^(-( + x + x + ... + x)))

FONCTION LOGIT:
logit(p) = ln(p/(1-p)) = + x + x + ... + x

CARACTÉRISTIQUES:
- Sortie entre 0 et 1 (probabilité)
- Courbe en S (sigmoïde)
- Linéaire en logit
- Maximum de vraisemblance pour l'optimisation
""")

# Visualisation de la fonction sigmoïde
z = np.linspace(-10, 10, 100)
sigmoid = 1 / (1 + np.exp(-z))

plt.figure(figsize=(12, 8))

# Subplot 1: Fonction sigmoïde
plt.subplot(2, 2, 1)
plt.plot(z, sigmoid, 'b-', linewidth=2)
plt.axhline(y=0.5, color='r', linestyle='--', alpha=0.7)
plt.axvline(x=0, color='r', linestyle='--', alpha=0.7)
plt.xlabel('z = + x + x + ...')
plt.ylabel('P(y=1|x)')
plt.title('Fonction Sigmoïde')
plt.grid(True, alpha=0.3)
plt.text(2, 0.2, 'P(y=1|x) = 1/(1+e^(-z))', fontsize=12,
        bbox=dict(boxstyle="round", facecolor='lightblue'))

# Subplot 2: Comparaison avec régression linéaire
x_comp = np.linspace(-3, 3, 100)
y_linear = 0.5 + 0.3 * x_comp # Régression linéaire
y_logistic = 1 / (1 + np.exp(-(0.0 + 1.5 * x_comp))) # Régression logistique

plt.subplot(2, 2, 2)
plt.plot(x_comp, y_linear, 'g-', linewidth=2, label='Régression linéaire')
plt.plot(x_comp, y_logistic, 'b-', linewidth=2, label='Régression logistique')
plt.ylim(-0.1, 1.1)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Comparaison: Linéaire vs Logistique')
plt.legend()
plt.grid(True, alpha=0.3)

# Subplot 3: Effet des coefficients
x_effects = np.linspace(-5, 5, 100)

```

```

for beta in [0.5, 1, 2, 3]:
    y_effect = 1 / (1 + np.exp(-beta * x_effects))
    plt.subplot(2, 2, 3)
    plt.plot(x_effects, y_effect, linewidth=2, label=f' = {beta}')
    plt.xlabel('x')
    plt.ylabel('P(y=1|x)')
    plt.title('Effet du coefficient ')
    plt.legend()
    plt.grid(True, alpha=0.3)

# Subplot 4: Données binaires exemple
np.random.seed(42)
n_points = 100
x_binary = np.random.normal(0, 2, n_points)
prob_true = 1 / (1 + np.exp(-1.5 * x_binary))
y_binary = np.random.binomial(1, prob_true)

plt.subplot(2, 2, 4)
plt.scatter(x_binary, y_binary, alpha=0.6, color='red')
x_smooth = np.linspace(x_binary.min(), x_binary.max(), 100)
prob_smooth = 1 / (1 + np.exp(-1.5 * x_smooth))
plt.plot(x_smooth, prob_smooth, 'b-', linewidth=2, label='Probabilité vraie')
plt.xlabel('x')
plt.ylabel('y (0 ou 1)')
plt.title('Données binaires et probabilité')
plt.legend()
plt.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

```

==== RÉGRESSION LOGISTIQUE - THÉORIE ====

PRINCIPE:

La régression logistique modélise la probabilité $P(y=1|x)$ avec la fonction sigmoïde:

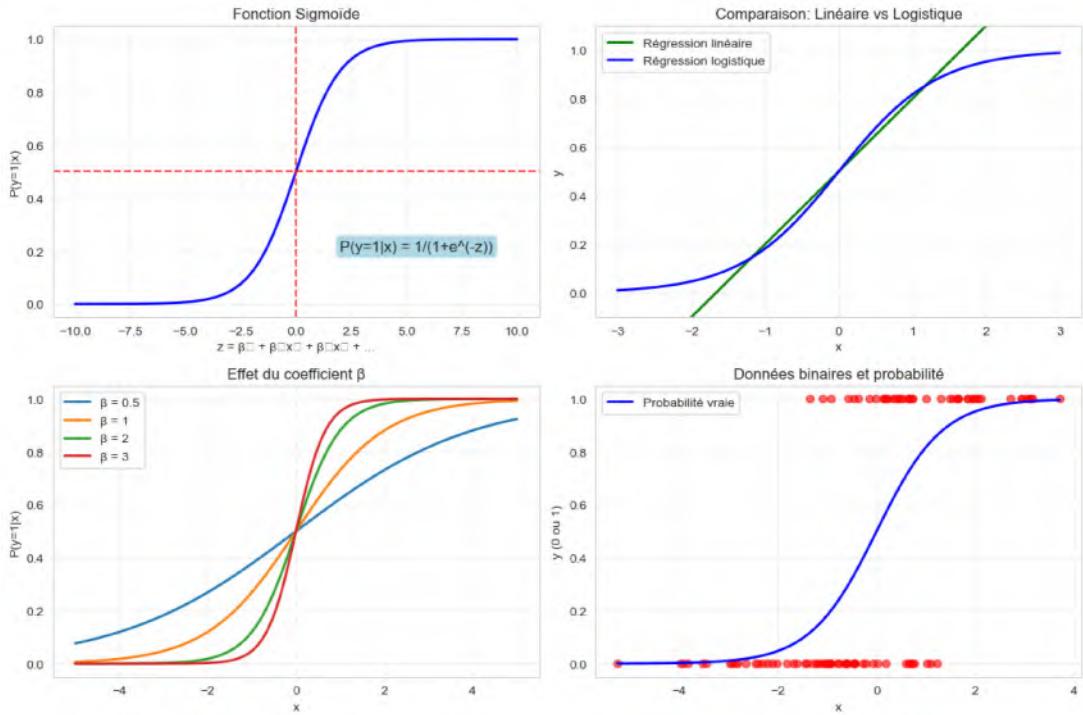
$$P(y=1|x) = 1 / (1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n)})$$

FONCTION LOGIT:

$$\text{logit}(p) = \ln(p/(1-p)) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_n x_n$$

CARACTÉRISTIQUES:

- Sortie entre 0 et 1 (probabilité)
- Courbe en S (sigmoïde)
- Linéaire en logit
- Maximum de vraisemblance pour l'optimisation



Implémentation pratique

```
[96]: print("\n==== IMPLÉMENTATION PRATIQUE ====")

# Dataset de démonstration: Admission universitaire
np.random.seed(42)
n_students = 1000

# Variables indépendantes
gre_scores = np.random.normal(500, 100, n_students) # Score GRE
gpa = np.random.normal(3.0, 0.5, n_students) # GPA
research = np.random.binomial(1, 0.3, n_students) # Expérience recherche

# Normalisation
gre_norm = (gre_scores - 500) / 100
gpa_norm = (gpa - 3.0) / 0.5

# Variable dépendante (admission)
# Formule: plus le GRE et GPA sont élevés, plus la probabilité d'admission est forte
linear_combination = -1 + 0.8 * gre_norm + 1.2 * gpa_norm + 0.5 * research
admission_prob = 1 / (1 + np.exp(-linear_combination))
```

```

admitted = np.random.binomial(1, admission_prob)

# Création du DataFrame
df_admission = pd.DataFrame({
    'gre_score': gre_scores,
    'gpa': gpa,
    'research_exp': research,
    'admitted': admitted
})

print("Dataset d'admission universitaire:")
print(df_admission.head(10))
print(f"\nTaux d'admission: {admitted.mean():.1%}")

# Visualisation des données
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# GRE vs Admission
axes[0, 0].scatter(df_admission[df_admission['admitted']==0]['gre_score'],
                   df_admission[df_admission['admitted']==0]['admitted'] + np.
                   random.normal(0, 0.02, sum(df_admission['admitted']==0)),
                   alpha=0.6, color='red', label='Refusé')
axes[0, 0].scatter(df_admission[df_admission['admitted']==1]['gre_score'],
                   df_admission[df_admission['admitted']==1]['admitted'] + np.
                   random.normal(0, 0.02, sum(df_admission['admitted']==1)),
                   alpha=0.6, color='green', label='Admis')
axes[0, 0].set_xlabel('Score GRE')
axes[0, 0].set_ylabel('Admission')
axes[0, 0].set_title('Admission vs Score GRE')
axes[0, 0].legend()

# GPA vs Admission
axes[0, 1].scatter(df_admission[df_admission['admitted']==0]['gpa'],
                   df_admission[df_admission['admitted']==0]['admitted'] + np.
                   random.normal(0, 0.02, sum(df_admission['admitted']==0)),
                   alpha=0.6, color='red', label='Refusé')
axes[0, 1].scatter(df_admission[df_admission['admitted']==1]['gpa'],
                   df_admission[df_admission['admitted']==1]['admitted'] + np.
                   random.normal(0, 0.02, sum(df_admission['admitted']==1)),
                   alpha=0.6, color='green', label='Admis')
axes[0, 1].set_xlabel('GPA')
axes[0, 1].set_ylabel('Admission')
axes[0, 1].set_title('Admission vs GPA')
axes[0, 1].legend()

# Recherche vs Admission

```

```

research_counts = df_admission.groupby(['research_exp', 'admitted']).size().
    ↪unstack()
research_counts.plot(kind='bar', ax=axes[1, 0], color=['red', 'green'])
axes[1, 0].set_xlabel('Expérience recherche')
axes[1, 0].set_ylabel('Nombre d\'étudiants')
axes[1, 0].set_title('Admission vs Expérience recherche')
axes[1, 0].legend(['Refusé', 'Admis'])
axes[1, 0].tick_params(axis='x', rotation=0)

# Matrice de corrélation
corr_admission = df_admission.corr()
sns.heatmap(corr_admission, annot=True, cmap='RdBu_r', center=0, ax=axes[1, 1])
axes[1, 1].set_title('Matrice de corrélation')

plt.tight_layout()
plt.show()

# Préparation des données pour le modèle
X_admission = df_admission[['gre_score', 'gpa', 'research_exp']]
y_admission = df_admission['admitted']

# Division train/test
X_train, X_test, y_train, y_test = train_test_split(
    X_admission, y_admission, test_size=0.2, random_state=42,
    ↪stratify=y_admission
)

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

# Entraînement du modèle
logistic_model = LogisticRegression(random_state=42)
logistic_model.fit(X_train_scaled, y_train)

# Prédictions
y_train_pred = logistic_model.predict(X_train_scaled)
y_test_pred = logistic_model.predict(X_test_scaled)
y_train_proba = logistic_model.predict_proba(X_train_scaled)[:, 1]
y_test_proba = logistic_model.predict_proba(X_test_scaled)[:, 1]

# Évaluation
train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

print(f"\nRÉSULTATS DU MODÈLE:")

```

```

print(f"Précision Train: {train_accuracy:.3f}")
print(f"Précision Test: {test_accuracy:.3f}")

# Coefficients et interprétation
feature_names = ['Score GRE', 'GPA', 'Expérience recherche']
coefficients = logistic_model.coef_[0]
intercept = logistic_model.intercept_[0]

print(f"\nCOEFFICIENTS DU MODÈLE:")
print(f"Intercept: {intercept:.3f}")
for name, coef in zip(feature_names, coefficients):
    odds_ratio = np.exp(coef)
    print(f"{name}: {coef:.3f} (Odds Ratio: {odds_ratio:.3f})")

print(f"\nINTERPRÉTATION:")
for name, coef in zip(feature_names, coefficients):
    odds_ratio = np.exp(coef)
    if odds_ratio > 1:
        effect = f"augmente les chances d'admission de {(odds_ratio-1)*100:..1f}%""
    else:
        effect = f"diminue les chances d'admission de {(1-odds_ratio)*100:..1f}%""
    print(f"• Une augmentation d'1 écart-type de {name} {effect}")

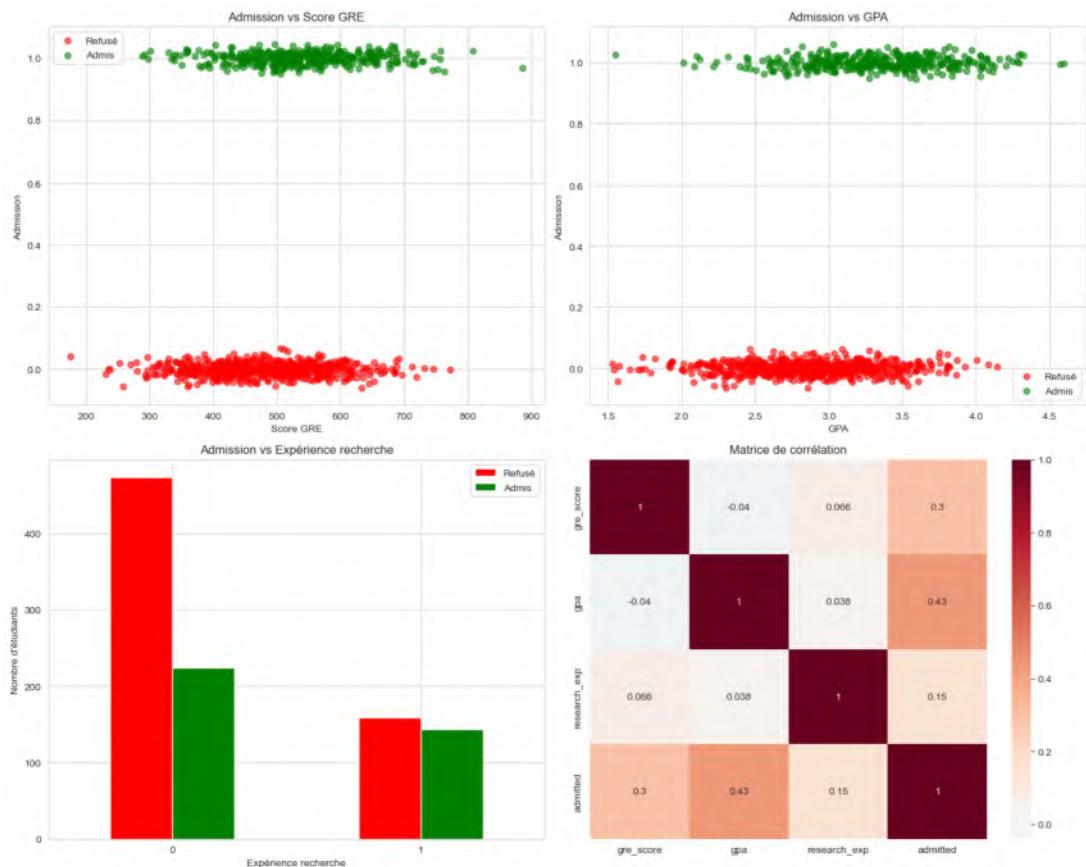
```

==== IMPLÉMENTATION PRATIQUE ===

Dataset d'admission universitaire:

	gre_score	gpa	research_exp	admitted
0	549.671415	3.699678	0	1
1	486.173570	3.462317	0	1
2	564.768854	3.029815	0	1
3	652.302986	2.676532	0	0
4	476.584663	3.349112	1	0
5	476.586304	3.196743	1	0
6	657.921282	3.447597	0	1
7	576.743473	3.317586	1	1
8	453.052561	3.524776	0	0
9	554.256004	2.732382	0	0

Taux d'admission: 36.8%



RÉSULTATS DU MODÈLE:

Précision Train: 0.789

Précision Test: 0.720

COEFFICIENTS DU MODÈLE:

Intercept: -0.811

Score GRE: 0.887 (Odds Ratio: 2.427)

GPA: 1.385 (Odds Ratio: 3.995)

Expérience recherche: 0.380 (Odds Ratio: 1.462)

INTERPRÉTATION:

- Une augmentation d'1 écart-type de Score GRE augmente les chances d'admission de 142.7%
- Une augmentation d'1 écart-type de GPA augmente les chances d'admission de 299.5%
- Une augmentation d'1 écart-type de Expérience recherche augmente les chances d'admission de 46.2%

Évaluation et métriques

```
[98]: print("\n==== ÉVALUATION DÉTAILLÉE ===")  
  
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, □  
    ↪auc  
from sklearn.metrics import precision_recall_curve, average_precision_score  
  
# Matrice de confusion  
cm = confusion_matrix(y_test, y_test_pred)  
print("Matrice de confusion:")  
print(cm)  
  
# Rapport de classification détaillé  
print(f"\nRapport de classification:")  
print(classification_report(y_test, y_test_pred))  
  
# Visualisation des métriques  
fig, axes = plt.subplots(2, 2, figsize=(15, 12))  
  
# 1. Matrice de confusion  
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[0, 0])  
axes[0, 0].set_xlabel('Prédiction')  
axes[0, 0].set_ylabel('Réalité')  
axes[0, 0].set_title('Matrice de confusion')  
  
# 2. Distribution des probabilités prédites  
axes[0, 1].hist(y_test_proba[y_test == 0], bins=20, alpha=0.7, color='red', □  
    ↪label='Refusé')  
axes[0, 1].hist(y_test_proba[y_test == 1], bins=20, alpha=0.7, color='green', □  
    ↪label='Admis')  
axes[0, 1].axvline(x=0.5, color='black', linestyle='--', label='Seuil 0.5')  
axes[0, 1].set_xlabel('Probabilité prédite')  
axes[0, 1].set_ylabel('Fréquence')  
axes[0, 1].set_title('Distribution des probabilités')  
axes[0, 1].legend()  
  
# 3. Courbe ROC  
fpr, tpr, _ = roc_curve(y_test, y_test_proba)  
roc_auc = auc(fpr, tpr)  
  
axes[1, 0].plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC (AUC = {roc_auc: □  
    ↪.3f})')  
axes[1, 0].plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', □  
    ↪label='Hasard')  
axes[1, 0].set_xlabel('Taux de faux positifs')  
axes[1, 0].set_ylabel('Taux de vrais positifs')  
axes[1, 0].set_title('Courbe ROC')
```

```

axes[1, 0].legend()

# 4. Courbe Précision-Rappel
precision, recall, _ = precision_recall_curve(y_test, y_test_proba)
avg_precision = average_precision_score(y_test, y_test_proba)

axes[1, 1].plot(recall, precision, color='blue', lw=2, label=f'PR (AP = {avg_precision:.3f})')
axes[1, 1].axhline(y=y_test.mean(), color='navy', linestyle='--', label='Baseline')
axes[1, 1].set_xlabel('Rappel')
axes[1, 1].set_ylabel('Précision')
axes[1, 1].set_title('Courbe Précision-Rappel')
axes[1, 1].legend()

plt.tight_layout()
plt.show()

# Analyse de l'impact du seuil de décision
thresholds = np.linspace(0.1, 0.9, 9)
metrics_by_threshold = {
    'threshold': [],
    'precision': [],
    'recall': [],
    'f1_score': [],
    'accuracy': []
}

from sklearn.metrics import precision_score, recall_score, f1_score

for threshold in thresholds:
    y_pred_threshold = (y_test_proba >= threshold).astype(int)

    metrics_by_threshold['threshold'].append(threshold)
    metrics_by_threshold['precision'].append(precision_score(y_test, y_pred_threshold))
    metrics_by_threshold['recall'].append(recall_score(y_test, y_pred_threshold))
    metrics_by_threshold['f1_score'].append(f1_score(y_test, y_pred_threshold))
    metrics_by_threshold['accuracy'].append(accuracy_score(y_test, y_pred_threshold))

# Visualisation de l'impact du seuil
plt.figure(figsize=(10, 6))
for metric in ['precision', 'recall', 'f1_score', 'accuracy']:
    plt.plot(metrics_by_threshold['threshold'], metrics_by_threshold[metric], 'o--', label=metric.replace('_', ' ').title())

```

```

plt.xlabel('Seuil de décision')
plt.ylabel('Score')
plt.title('Impact du seuil de décision sur les métriques')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

print(f"\nMETRIQUES PAR SEUIL:")
df_metrics = pd.DataFrame(metrics_by_threshold)
print(df_metrics.round(3))

```

==== ÉVALUATION DÉTAILLÉE ===

Matrice de confusion:

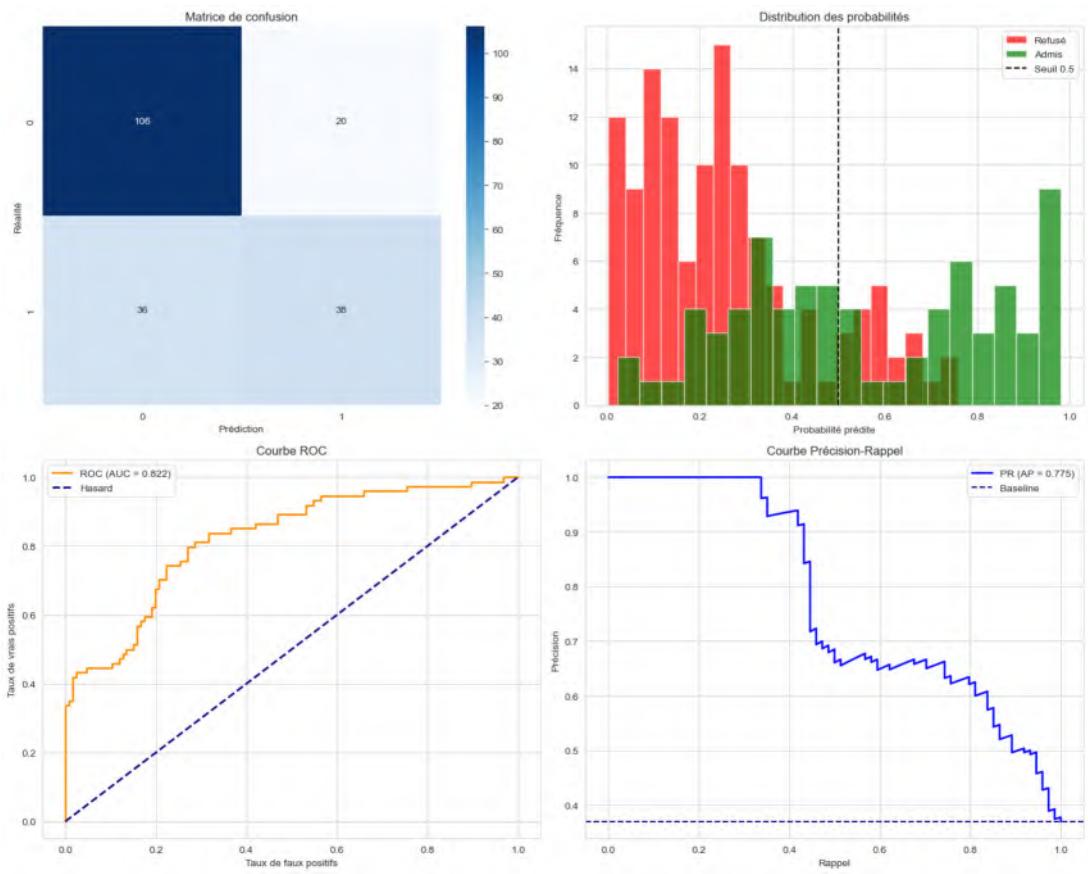
```

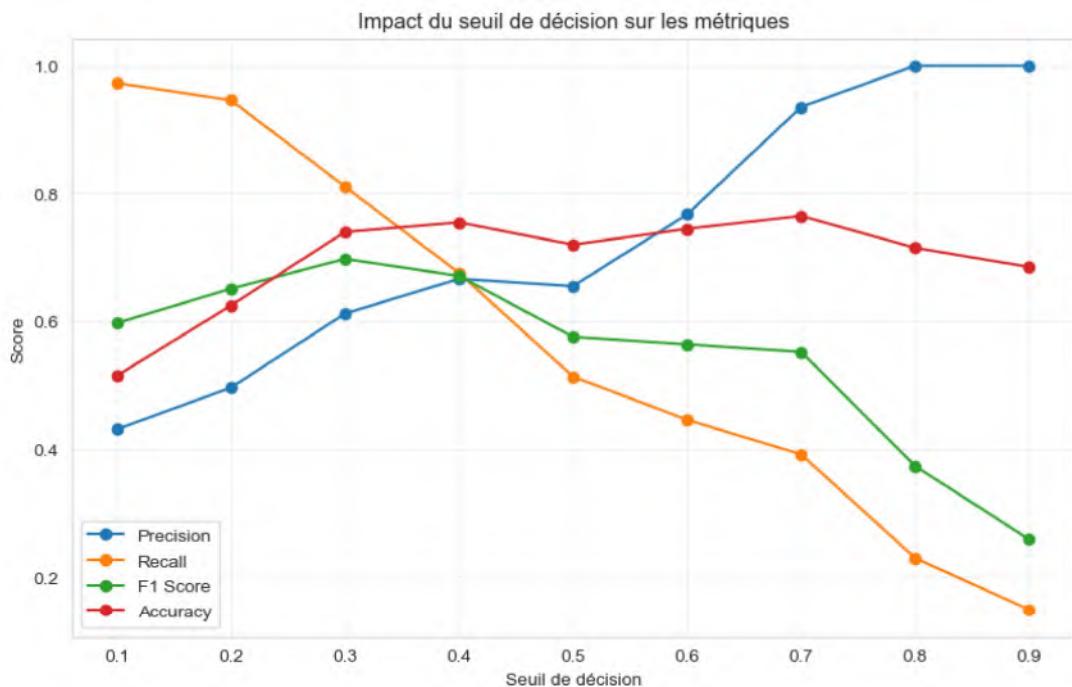
[[106  20]
 [ 36  38]]

```

Rapport de classification:

	precision	recall	f1-score	support
0	0.75	0.84	0.79	126
1	0.66	0.51	0.58	74
accuracy			0.72	200
macro avg	0.70	0.68	0.68	200
weighted avg	0.71	0.72	0.71	200





METRIQUES PAR SEUIL:

	threshold	precision	recall	f1_score	accuracy
0	0.1	0.431	0.973	0.598	0.515
1	0.2	0.496	0.946	0.651	0.625
2	0.3	0.612	0.811	0.698	0.740
3	0.4	0.667	0.676	0.671	0.755
4	0.5	0.655	0.514	0.576	0.720
5	0.6	0.767	0.446	0.564	0.745
6	0.7	0.935	0.392	0.552	0.765
7	0.8	1.000	0.230	0.374	0.715
8	0.9	1.000	0.149	0.259	0.685

Cas d'usage avancés

```
[100]: print("\n==== CAS D'USAGE AVANCÉS ====")

# 1. Régression logistique multiclasse
print("1. CLASSIFICATION MULTICLASSE")
iris = load_iris()
X_iris, y_iris = iris.data, iris.target

# Modèle multiclasse (One-vs-Rest par défaut)
logistic_multi = LogisticRegression(multi_class='ovr', random_state=42)
logistic_multi.fit(X_iris, y_iris)
```

```

# Prédictions
y_iris_pred = logistic_multi.predict(X_iris)
y_iris_proba = logistic_multi.predict_proba(X_iris)

print(f"Classes: {iris.target_names}")
print(f"Précision: {accuracy_score(y_iris, y_iris_pred):.3f}")

# Matrice de confusion multiclasse
cm_multi = confusion_matrix(y_iris, y_iris_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm_multi, annot=True, fmt='d', cmap='Blues',
            xticklabels=iris.target_names, yticklabels=iris.target_names)
plt.xlabel('Prédition')
plt.ylabel('Réalité')
plt.title('Matrice de confusion - Classification multiclasse')
plt.show()

# 2. Régularisation (Ridge et Lasso)
print("\n2. RÉGULARISATION")
print("La régularisation aide à éviter le surapprentissage")

# Données avec plus de features pour démontrer la régularisation
from sklearn.datasets import make_classification
X_reg, y_reg = make_classification(n_samples=1000, n_features=20,
                                     n_informative=5,
                                     n_redundant=15, random_state=42)

X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(
    X_reg, y_reg, test_size=0.2, random_state=42
)

# Modèles avec différents niveaux de régularisation
C_values = [0.01, 0.1, 1, 10, 100]
models = {}

for C in C_values:
    # L2 (Ridge)
    model_l2 = LogisticRegression(C=C, penalty='l2', random_state=42)
    model_l2.fit(X_train_reg, y_train_reg)

    # L1 (Lasso)
    model_l1 = LogisticRegression(C=C, penalty='l1', solver='liblinear',
                                  random_state=42)
    model_l1.fit(X_train_reg, y_train_reg)

    models[f'L2_C{C}'] = model_l2

```

```

models[f'L1_C{C}'] = model_l1

# Comparaison des performances
results = []
for name, model in models.items():
    train_score = model.score(X_train_reg, y_train_reg)
    test_score = model.score(X_test_reg, y_test_reg)
    n_features_used = np.sum(np.abs(model.coef_[0]) > 1e-5)

    results.append({
        'Model': name,
        'Train_Score': train_score,
        'Test_Score': test_score,
        'Features_Used': n_features_used
    })

df_results = pd.DataFrame(results)
print("\nComparaison des modèles régularisés:")
print(df_results.round(3))

# Visualisation de l'effet de la régularisation
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Coefficients L2
C_vals = [0.01, 0.1, 1, 10, 100]
coef_l2 = []
coef_l1 = []

for C in C_vals:
    model_l2 = LogisticRegression(C=C, penalty='l2', random_state=42)
    model_l2.fit(X_train_reg, y_train_reg)
    coef_l2.append(model_l2.coef_[0])

    model_l1 = LogisticRegression(C=C, penalty='l1', solver='liblinear', random_state=42)
    model_l1.fit(X_train_reg, y_train_reg)
    coef_l1.append(model_l1.coef_[0])

# L2 regularization
for i in range(5): # Montrer seulement 5 features pour la clarté
    coef_feature = [coef[i] for coef in coef_l2]
    axes[0].plot(C_vals, coef_feature, 'o-', label=f'Feature {i+1}')

    axes[0].set_xscale('log')
    axes[0].set_xlabel('C (inverse de la régularisation)')
    axes[0].set_ylabel('Coefficient')
    axes[0].set_title('Régularisation L2 (Ridge)')

```

```

axes[0].legend()
axes[0].grid(True, alpha=0.3)

# L1 regularization
for i in range(5):
    coef_feature = [coef[i] for coef in coef_l1]
    axes[1].plot(C_vals, coef_feature, 'o-', label=f'Feature {i+1}')

axes[1].set_xscale('log')
axes[1].set_xlabel('C (inverse de la régularisation)')
axes[1].set_ylabel('Coefficient')
axes[1].set_title('Régularisation L1 (Lasso)')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

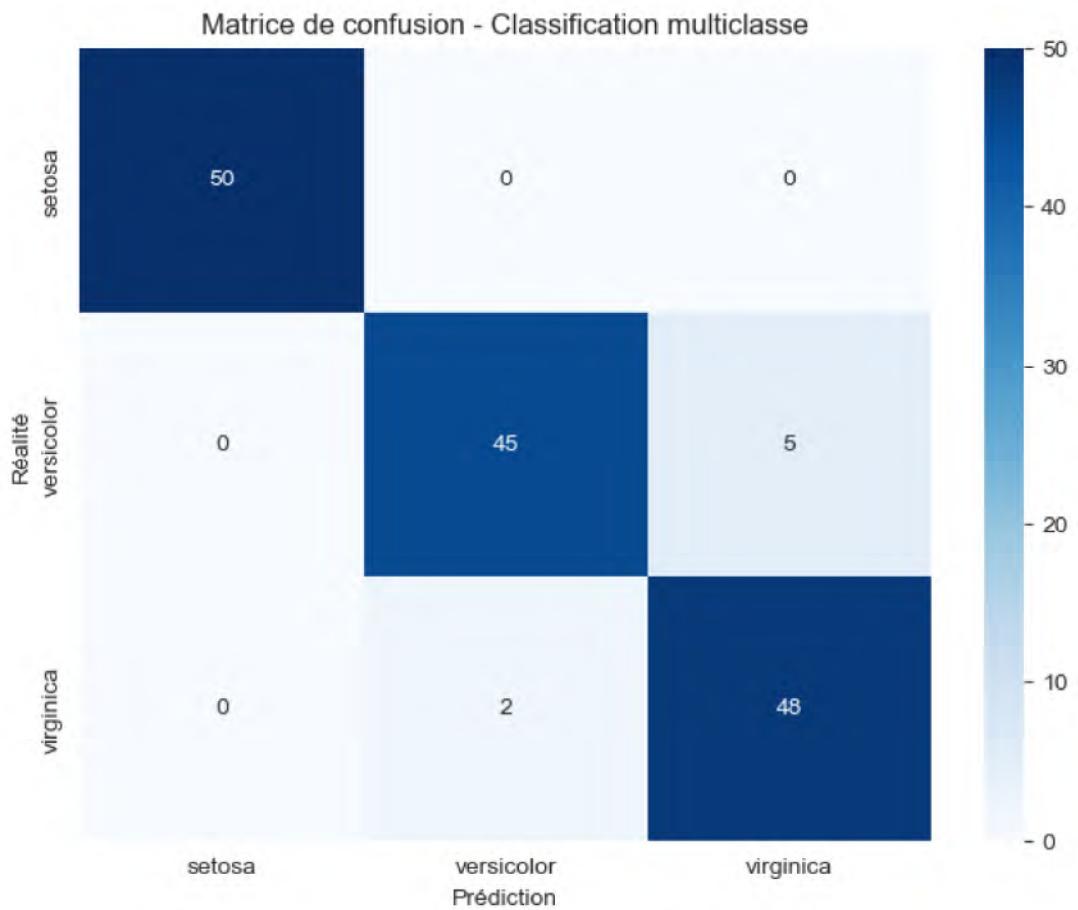
print("\nOBSERVATIONS:")
print("• L2 (Ridge): Réduit les coefficients mais ne les annule pas")
print("• L1 (Lasso): Peut annuler des coefficients (sélection de features)")
print("• C faible = forte régularisation")
print("• C élevé = faible régularisation")

```

```

==== CAS D'USAGE AVANCÉS ===
1. CLASSIFICATION MULTICLASSE
Classes: ['setosa' 'versicolor' 'virginica']
Précision: 0.953

```

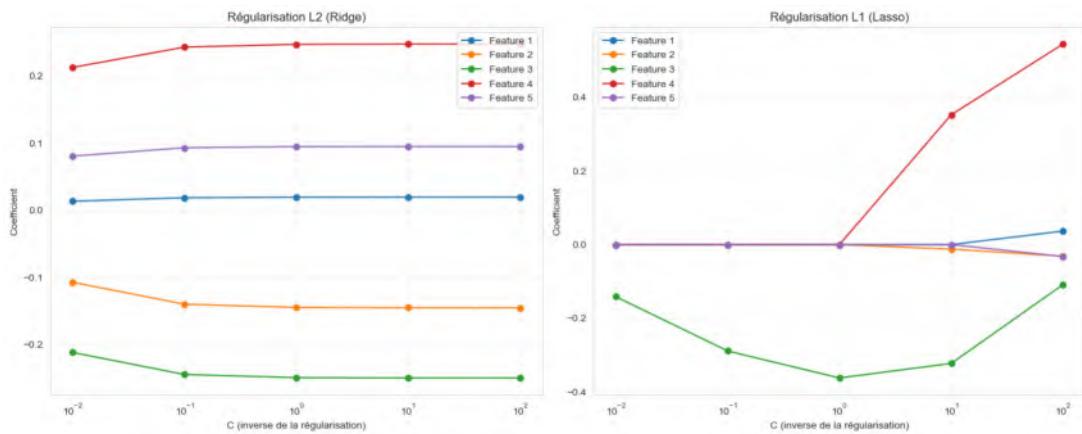


2. RÉGULARISATION

La régularisation aide à éviter le surapprentissage

Comparaison des modèles régularisés:

	Model	Train_Score	Test_Score	Features_Used
0	L2_C0.01	0.819	0.840	20
1	L1_C0.01	0.770	0.765	3
2	L2_C0.1	0.815	0.840	20
3	L1_C0.1	0.809	0.835	5
4	L2_C1	0.814	0.835	20
5	L1_C1	0.815	0.840	5
6	L2_C10	0.814	0.835	20
7	L1_C10	0.814	0.835	8
8	L2_C100	0.814	0.835	20
9	L1_C100	0.814	0.835	20



OBSERVATIONS:

- L2 (Ridge): Réduit les coefficients mais ne les annule pas
- L1 (Lasso): Peut annuler des coefficients (sélection de features)
- C faible = forte régularisation
- C élevé = faible régularisation

1.7 Algorithmes ML avancés

1.7.1 KNN (K-Nearest Neighbors)

L'algorithme des k plus proches voisins (KNN) est un algorithme d'apprentissage paresseux qui classe un nouvel échantillon en se basant sur la classe majoritaire de ses k plus proches voisins.

Principe et implémentation

```
[102]: print("== K-NEAREST NEIGHBORS (KNN) ==")  
  
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor  
from sklearn.datasets import make_classification, make_regression  
from sklearn.model_selection import train_test_split, cross_val_score  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd  
  
print("""  
PRINCIPE DE KNN:  
1. Algorithme "paresseux" (lazy learning)  
2. Pas d'entraînement explicite  
3. Classification/Régression basée sur les k plus proches voisins  
4. Distance généralement euclidienne
```

AVANTAGES:

- Simple à comprendre et implémenter
- Pas d'hypothèses sur les données
- Efficace pour des données localement cohérentes

INCONVÉNIENTS:

- Coûteux en calcul ($O(n)$ pour chaque prédiction)
 - Sensible à la dimension (fléau de la dimension)
 - Sensible aux features non normalisées
- """)

```
# Génération de données 2D pour visualisation
np.random.seed(42)
X_2d, y_2d = make_classification(n_samples=300, n_features=2, n_redundant=0,
                                 n_informative=2, n_clusters_per_class=1,
                                 random_state=42)

# Visualisation des données
plt.figure(figsize=(15, 10))

# Dataset original
plt.subplot(2, 3, 1)
scatter = plt.scatter(X_2d[:, 0], X_2d[:, 1], c=y_2d, cmap='viridis', alpha=0.7)
plt.colorbar(scatter)
plt.title('Dataset original')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')

# Fonction pour visualiser les frontières de décision
def plot_decision_boundary(X, y, classifier, title, ax=None):
    if ax is None:
        fig, ax = plt.subplots()

    h = 0.02 # Pas de la grille
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    ax.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')
    scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', edgecolors='black')
    ax.set_title(title)
    ax.set_xlabel('Feature 1')
```

```

ax.set_ylabel('Feature 2')

# Test avec différentes valeurs de k
k_values = [1, 3, 15, 50]
for i, k in enumerate(k_values):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_2d, y_2d)

    plt.subplot(2, 3, i + 2)
    plot_decision_boundary(X_2d, y_2d, knn, f'KNN avec k={k}', plt.gca())

plt.tight_layout()
plt.show()

# Analyse de l'impact de k
print("\n==== IMPACT DU PARAMÈTRE K ====")

k_range = range(1, 51)
cv_scores = []

for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X_2d, y_2d, cv=5, scoring='accuracy')
    cv_scores.append(scores.mean())

# Visualisation
plt.figure(figsize=(10, 6))
plt.plot(k_range, cv_scores, 'o-', linewidth=2, markersize=6)
plt.xlabel('Valeur de k')
plt.ylabel('Précision (validation croisée)')
plt.title('Impact de k sur la performance')
plt.grid(True, alpha=0.3)

# Marquer le k optimal
optimal_k = k_range[np.argmax(cv_scores)]
plt.axvline(x=optimal_k, color='red', linestyle='--', alpha=0.7)
plt.text(optimal_k + 1, max(cv_scores) - 0.01, f'k optimal = {optimal_k}', 
         bbox=dict(boxstyle="round", facecolor='lightcoral', alpha=0.8))

plt.show()

print(f"K optimal: {optimal_k}")
print(f"Meilleur score: {max(cv_scores):.3f}")

```

==== K-NEAREST NEIGHBORS (KNN) ====

PRINCIPE DE KNN:

1. Algorithme "paresseux" (lazy learning)

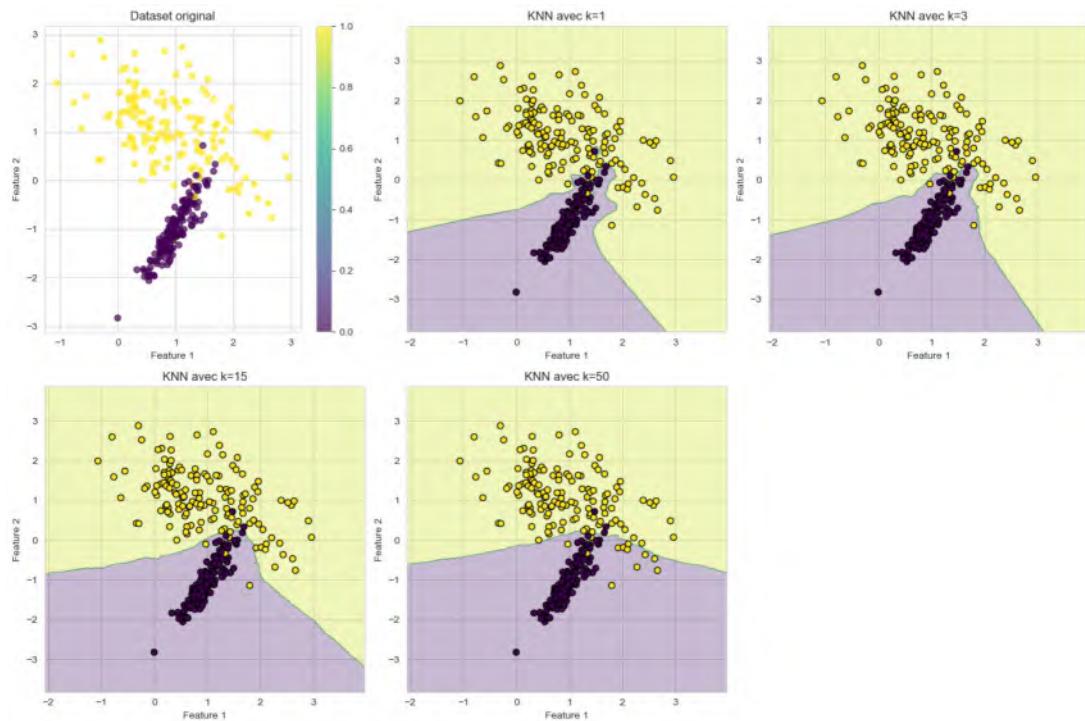
2. Pas d'entraînement explicite
3. Classification/Régression basée sur les k plus proches voisins
4. Distance généralement euclidienne

AVANTAGES:

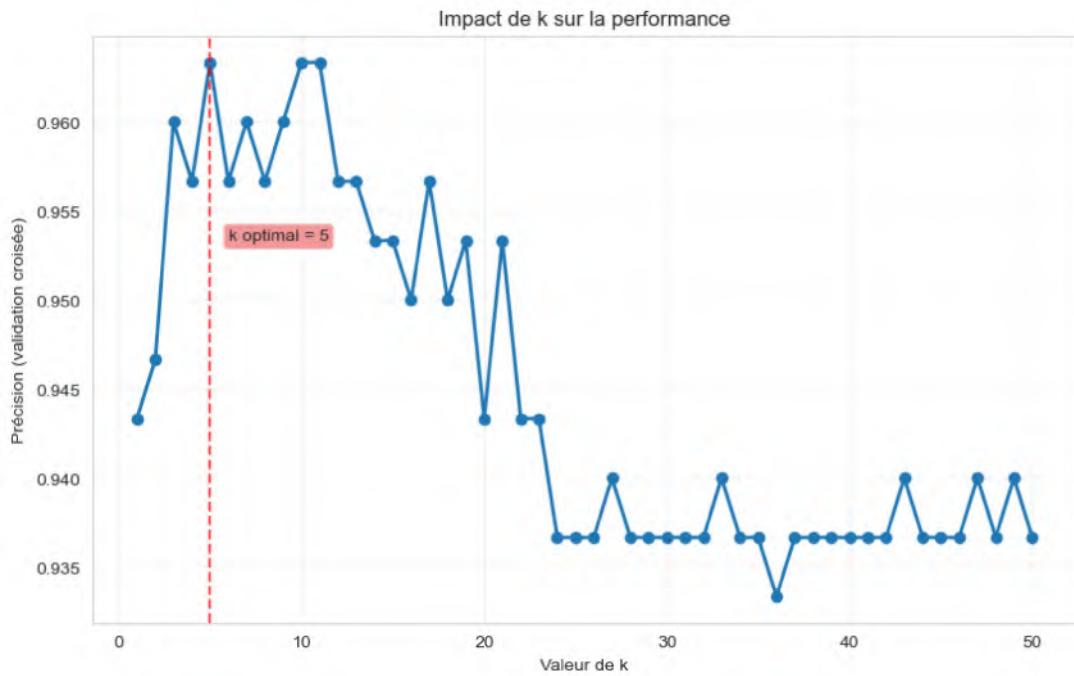
- Simple à comprendre et implémenter
- Pas d'hypothèses sur les données
- Efficace pour des données localement cohérentes

INCONVÉNIENTS:

- Coûteux en calcul ($O(n)$ pour chaque prédition)
- Sensible à la dimension (fléau de la dimension)
- Sensible aux features non normalisées



==== IMPACT DU PARAMÈTRE K ===



K optimal: 5

Meilleur score: 0.963

Effet de la normalisation

```
[104]: print("\n==== IMPORTANCE DE LA NORMALISATION ===")
```

```
# Création de données avec des échelles différentes
np.random.seed(42)
X_scale = np.random.randn(200, 2)
X_scale[:, 0] *= 100 # Feature 1: échelle 0-100
X_scale[:, 1] *= 1    # Feature 2: échelle 0-1
y_scale = (X_scale[:, 0] + X_scale[:, 1] > 0).astype(int)

# Test avec et sans normalisation
from sklearn.preprocessing import StandardScaler

# Sans normalisation
knn_no_scale = KNeighborsClassifier(n_neighbors=5)
scores_no_scale = cross_val_score(knn_no_scale, X_scale, y_scale, cv=5)

# Avec normalisation
scaler = StandardScaler()
X_scale_normalized = scaler.fit_transform(X_scale)
knn_scale = KNeighborsClassifier(n_neighbors=5)
```

```

scores_scale = cross_val_score(knn_scale, X_scale_normalized, y_scale, cv=5)

print(f"Performance sans normalisation: {scores_no_scale.mean():.3f} ± {scores_no_scale.std():.3f}")
print(f"Performance avec normalisation: {scores_scale.mean():.3f} ± {scores_scale.std():.3f}")

# Visualisation
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Sans normalisation
knn_no_scale.fit(X_scale, y_scale)
plot_decision_boundary(X_scale, y_scale, knn_no_scale, 'Sans normalisation', axes[0])

# Avec normalisation
knn_scale.fit(X_scale_normalized, y_scale)
plot_decision_boundary(X_scale_normalized, y_scale, knn_scale, 'Avec normalisation', axes[1])

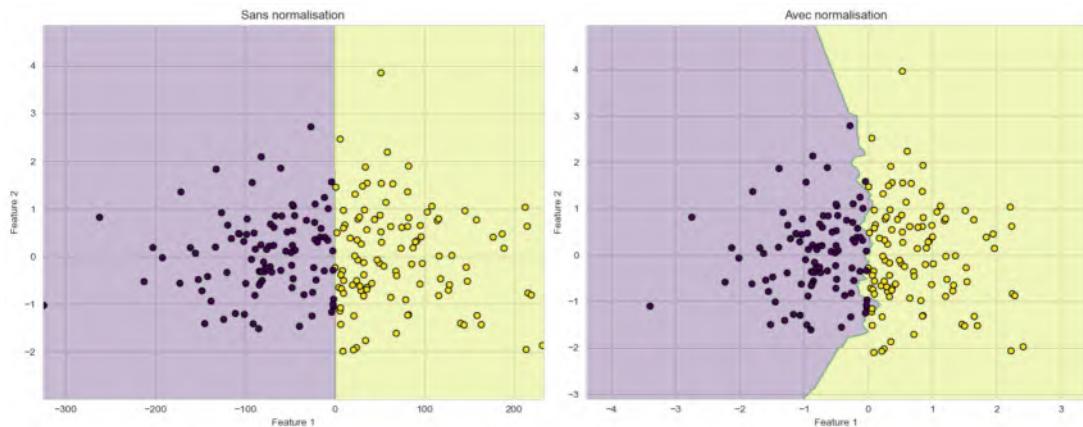
plt.tight_layout()
plt.show()

```

==== IMPORTANCE DE LA NORMALISATION ===

Performance sans normalisation: 0.990 ± 0.012

Performance avec normalisation: 0.940 ± 0.034



KNN pour la régression

[106]: `print("\n==== KNN POUR LA RÉGRESSION ===")`

```

# Génération de données de régression
X_reg, y_reg = make_regression(n_samples=100, n_features=1, noise=20, random_state=42)

# Division des données
X_train, X_test, y_train, y_test = train_test_split(X_reg, y_reg, test_size=0.3, random_state=42)

# Comparaison avec différents k
fig, axes = plt.subplots(2, 2, figsize=(15, 12))
k_values_reg = [1, 3, 10, 30]

for i, k in enumerate(k_values_reg):
    row, col = i // 2, i % 2

    # Entraînement du modèle
    knn_reg = KNeighborsRegressor(n_neighbors=k)
    knn_reg.fit(X_train, y_train)

    # Prédictions pour visualisation
    X_plot = np.linspace(X_reg.min(), X_reg.max(), 100).reshape(-1, 1)
    y_plot = knn_reg.predict(X_plot)

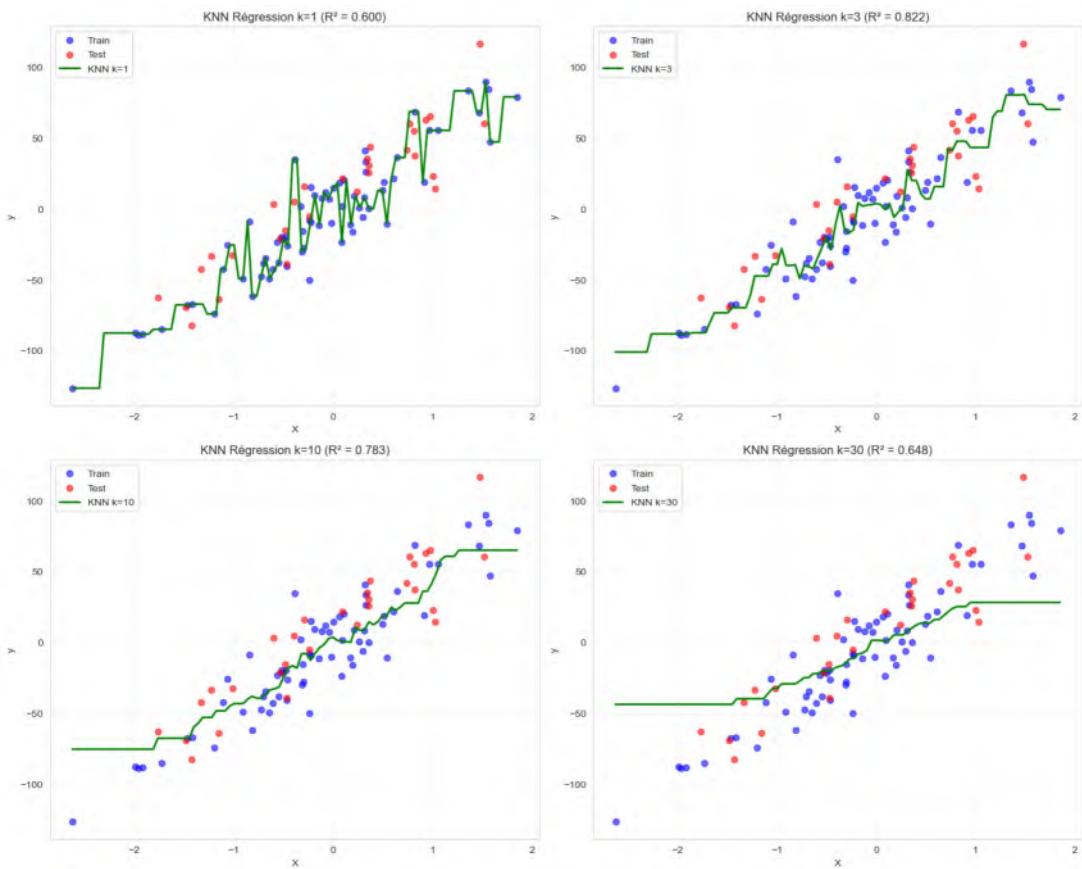
    # Score
    score = knn_reg.score(X_test, y_test)

    # Visualisation
    axes[row, col].scatter(X_train, y_train, alpha=0.6, color='blue', label='Train')
    axes[row, col].scatter(X_test, y_test, alpha=0.6, color='red', label='Test')
    axes[row, col].plot(X_plot, y_plot, color='green', linewidth=2, label=f'KNN k={k}')
    axes[row, col].set_title(f'KNN Régression k={k} (R² = {score:.3f})')
    axes[row, col].set_xlabel('X')
    axes[row, col].set_ylabel('y')
    axes[row, col].legend()
    axes[row, col].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

```

==== KNN POUR LA RÉGRESSION ===



1.7.2 Arbres de décision Forêts aléatoires

Les arbres de décision sont des modèles interprétables qui partitionnent récursivement l'espace des features. Les forêts aléatoires combinent plusieurs arbres pour améliorer la performance et réduire le surapprentissage.

Arbres de décision

```
[108]: print("== ARBRES DE DÉCISION ==")

from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor, plot_tree
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.model_selection import validation_curve

print("""
PRINCIPE DES ARBRES DE DÉCISION:
1. Partitionnement récursif de l'espace des features
2. Critères de division: Gini, Entropie, MSE
3. Arrêt basé sur profondeur, nombre d'échantillons, etc.
```

AVANTAGES:

- Hautement interprétable
- Gère les features catégorielles et numériques
- Pas besoin de normalisation
- Déetecte automatiquement les interactions

INCONVÉNIENTS:

- Tendance au surapprentissage
- Instabilité (petit changement → arbre très différent)
- Biais vers features avec plus de modalités
"")

```
# Dataset Iris pour démonstration
iris = load_iris()
X_iris, y_iris = iris.data, iris.target

# Arbre simple
tree_simple = DecisionTreeClassifier(max_depth=3, random_state=42)
tree_simple.fit(X_iris, y_iris)

# Visualisation de l'arbre
plt.figure(figsize=(20, 10))
plot_tree(tree_simple,
          feature_names=iris.feature_names,
          class_names=iris.target_names,
          filled=True,
          fontsize=10)
plt.title('Arbre de Décision - Dataset Iris')
plt.show()

# Performance selon la profondeur
max_depths = range(1, 16)
train_scores, val_scores = validation_curve(
    DecisionTreeClassifier(random_state=42), X_iris, y_iris,
    param_name='max_depth', param_range=max_depths,
    cv=5, scoring='accuracy'
)

plt.figure(figsize=(10, 6))
plt.plot(max_depths, train_scores.mean(axis=1), 'o-', color='blue',  
label='Score train')
plt.fill_between(max_depths, train_scores.mean(axis=1) - train_scores.  
std(axis=1),
                 train_scores.mean(axis=1) + train_scores.std(axis=1), alpha=0.  
1, color='blue')
```

```

plt.plot(max_depths, val_scores.mean(axis=1), 'o-', color='red', label='Score de validation')
plt.fill_between(max_depths, val_scores.mean(axis=1) - val_scores.std(axis=1),
                 val_scores.mean(axis=1) + val_scores.std(axis=1), alpha=0.1, color='red')

plt.xlabel('Profondeur maximale')
plt.ylabel('Précision')
plt.title('Courbe de validation - Profondeur de l\'arbre')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Importance des features
feature_importance = tree_simple.feature_importances_
feature_names = iris.feature_names

plt.figure(figsize=(10, 6))
indices = np.argsort(feature_importance)[::-1]
plt.bar(range(len(feature_importance)), feature_importance[indices])
plt.xticks(range(len(feature_importance)), [feature_names[i] for i in indices], rotation=45)
plt.title('Importance des Features - Arbre de Décision')
plt.ylabel('Importance')
plt.tight_layout()
plt.show()

print(f"\nImportance des features:")
for i, (name, importance) in enumerate(zip(feature_names, feature_importance)):
    print(f"{name}: {importance:.3f}")

```

==== ARBRES DE DÉCISION ===

PRINCIPE DES ARBRES DE DÉCISION:

1. Partitionnement récursif de l'espace des features
2. Critères de division: Gini, Entropie, MSE
3. Arrêt basé sur profondeur, nombre d'échantillons, etc.

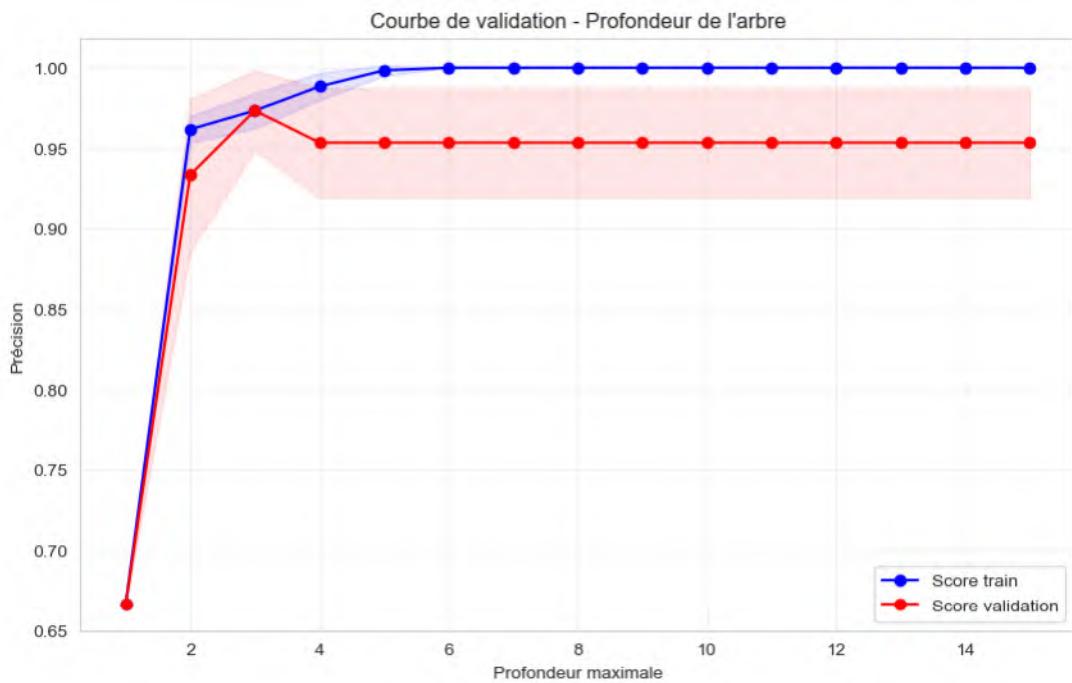
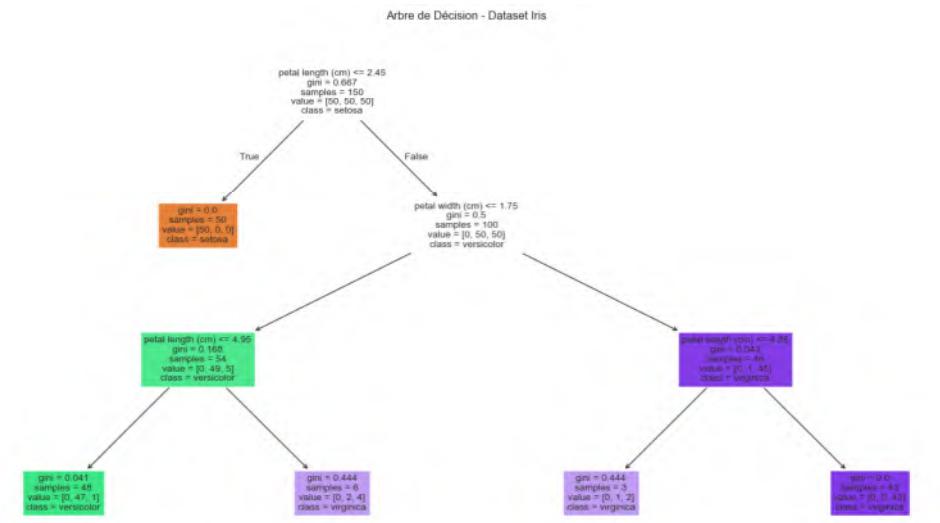
AVANTAGES:

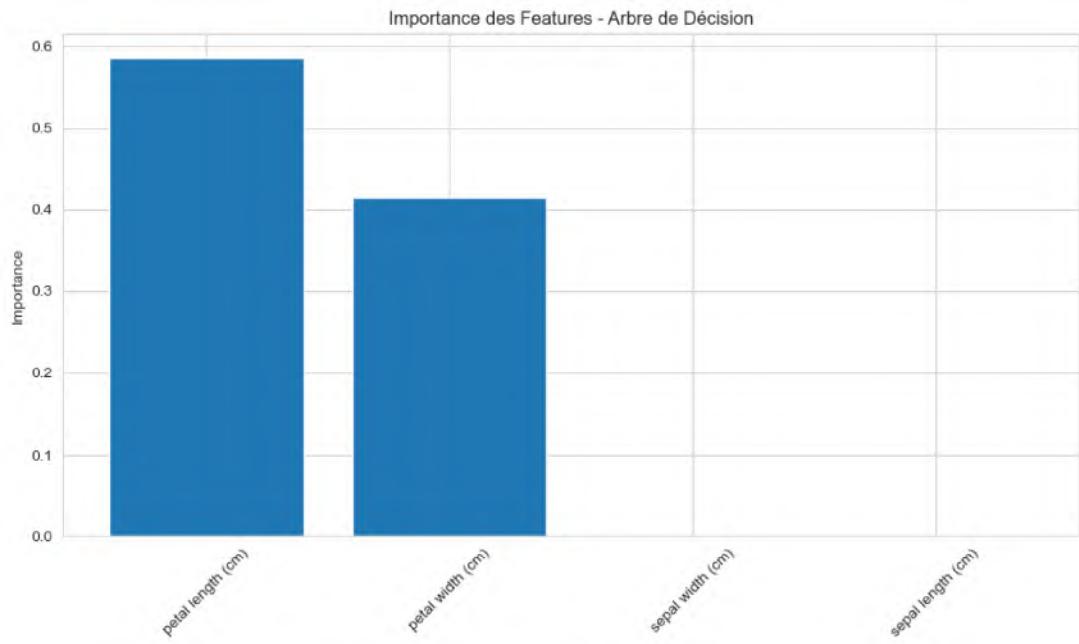
- Hautement interprétable
- Gère les features catégorielles et numériques
- Pas besoin de normalisation
- Détecte automatiquement les interactions

INCONVÉNIENTS:

- Tendance au surapprentissage
- Instabilité (petit changement → arbre très différent)

- Biais vers features avec plus de modalités





Importance des features:

sepal length (cm): 0.000

sepal width (cm): 0.000

petal length (cm): 0.586

petal width (cm): 0.414

Forêts aléatoires

[110]: `print("\n==== FORÊTS ALÉATOIRES ===")`

`print("")`

PRINCIPE DES FORÊTS ALÉATOIRES:

1. Ensemble d'arbres de décision
2. Bootstrap aggregating (bagging)
3. Sélection aléatoire de features à chaque division
4. Vote majoritaire (classification) ou moyenne (régression)

AVANTAGES:

- Réduit le surapprentissage des arbres individuels
- Estimation de l'importance des features
- Gère bien les gros datasets
- Robuste aux valeurs aberrantes

HYPERPARAMÈTRES CLÉS:

- `n_estimators`: nombre d'arbres

```

• max_depth: profondeur maximale
• max_features: nombre de features par division
• min_samples_split: échantillons minimum pour diviser
""")

# Comparaison Arbre vs Forêt
tree_single = DecisionTreeClassifier(random_state=42)
forest = RandomForestClassifier(n_estimators=100, random_state=42)

# Évaluation avec validation croisée
tree_scores = cross_val_score(tree_single, X_iris, y_iris, cv=5)
forest_scores = cross_val_score(forest, X_iris, y_iris, cv=5)

print(f"Arbre unique: {tree_scores.mean():.3f} ± {tree_scores.std():.3f}")
print(f"Forêt aléatoire: {forest_scores.mean():.3f} ± {forest_scores.std():.
    ↪3f}")

# Entraînement pour l'analyse des features
forest.fit(X_iris, y_iris)
tree_single.fit(X_iris, y_iris)

# Comparaison de l'importance des features
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Arbre unique
indices_tree = np.argsort(tree_single.feature_importances_)[-1]
axes[0].bar(range(len(tree_single.feature_importances_)),
            tree_single.feature_importances_[indices_tree])
axes[0].set_xticks(range(len(tree_single.feature_importances_)))
axes[0].set_xticklabels([feature_names[i] for i in indices_tree], rotation=45)
axes[0].set_title('Importance - Arbre unique')
axes[0].set_ylabel('Importance')

# Forêt aléatoire
indices_forest = np.argsort(forest.feature_importances_)[-1]
axes[1].bar(range(len(forest.feature_importances_)),
            forest.feature_importances_[indices_forest])
axes[1].set_xticks(range(len(forest.feature_importances_)))
axes[1].set_xticklabels([feature_names[i] for i in indices_forest], rotation=45)
axes[1].set_title('Importance - Forêt aléatoire')
axes[1].set_ylabel('Importance')

plt.tight_layout()
plt.show()

# Optimisation des hyperparamètres
print("\n==== OPTIMISATION DES HYPERPARAMÈTRES ===")

```

```

from sklearn.model_selection import GridSearchCV

# Grille de paramètres
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [3, 5, 10, None],
    'min_samples_split': [2, 5, 10],
    'max_features': ['sqrt', 'log2', None]
}

# Recherche par grille
rf_grid = GridSearchCV(
    RandomForestClassifier(random_state=42),
    param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

rf_grid.fit(X_iris, y_iris)

print(f"Meilleurs paramètres: {rf_grid.best_params_}")
print(f"Meilleur score: {rf_grid.best_score_:.3f}")

# Analyse de la convergence (nombre d'arbres)
n_estimators_range = range(10, 201, 10)
train_scores_trees = []
oob_scores = []

for n_est in n_estimators_range:
    rf_temp = RandomForestClassifier(n_estimators=n_est, oob_score=True,
                                     random_state=42)
    rf_temp.fit(X_iris, y_iris)
    train_scores_trees.append(rf_temp.score(X_iris, y_iris))
    oob_scores.append(rf_temp.oob_score_)

plt.figure(figsize=(10, 6))
plt.plot(n_estimators_range, train_scores_trees, 'o-', color='blue',
         label='Score train')
plt.plot(n_estimators_range, oob_scores, 'o-', color='red', label='Score OOB')
plt.xlabel('Nombre d\'arbres')
plt.ylabel('Précision')
plt.title('Convergence avec le nombre d\'arbres')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

```

```
print(f"Score OOB stabilisé autour de: {oob_scores[-5:]} (dernières 5 valeurs)")
```

==== FORÊTS ALÉATOIRES ===

PRINCIPE DES FORÊTS ALÉATOIRES:

1. Ensemble d'arbres de décision
2. Bootstrap aggregating (bagging)
3. Sélection aléatoire de features à chaque division
4. Vote majoritaire (classification) ou moyenne (régression)

AVANTAGES:

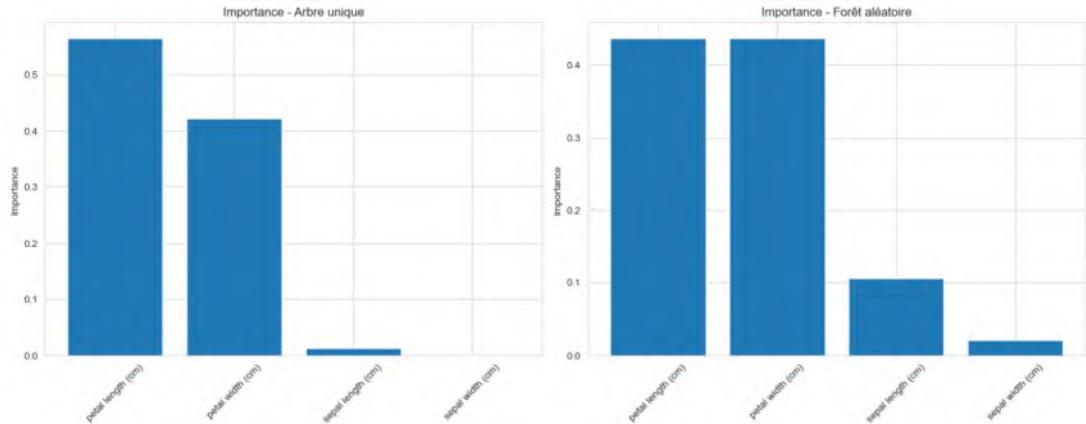
- Réduit le surapprentissage des arbres individuels
- Estimation de l'importance des features
- Gère bien les gros datasets
- Robuste aux valeurs aberrantes

HYPERPARAMÈTRES CLÉS:

- `n_estimators`: nombre d'arbres
- `max_depth`: profondeur maximale
- `max_features`: nombre de features par division
- `min_samples_split`: échantillons minimum pour diviser

Arbre unique: 0.953 ± 0.034

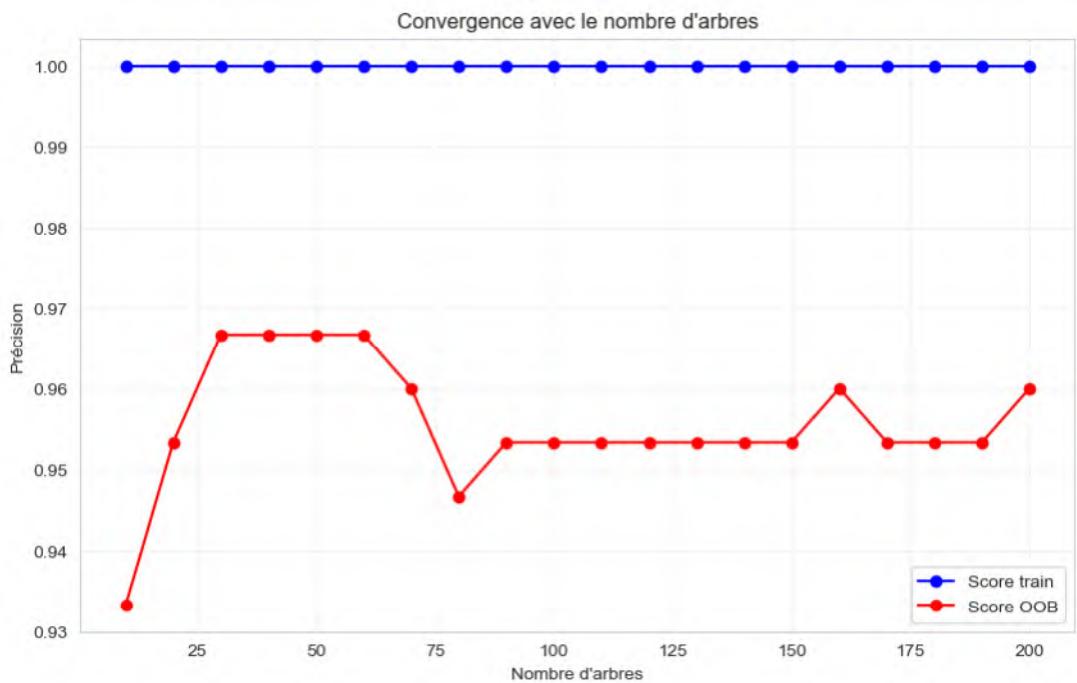
Forêt aléatoire: 0.967 ± 0.021



==== OPTIMISATION DES HYPERPARAMÈTRES ===

Meilleurs paramètres: {'max_depth': 3, 'max_features': 'sqrt', 'min_samples_split': 2, 'n_estimators': 50}

Meilleur score: 0.967



Score OOB stabilisé autour de: [0.96, 0.953333333333334, 0.953333333333334, 0.953333333333334, 0.96] (dernières 5 valeurs)

Exemple pratique complet

```
[112]: print("\n==== EXEMPLE PRATIQUE COMPLET ====")

# Dataset plus complexe pour démonstration
from sklearn.datasets import load_wine
wine = load_wine()
X_wine, y_wine = wine.data, wine.target

print(f"Dataset Vin: {X_wine.shape[0]} échantillons, {X_wine.shape[1]} features")
print(f"Classes: {wine.target_names}")

# Division des données
X_train, X_test, y_train, y_test = train_test_split(
    X_wine, y_wine, test_size=0.3, random_state=42, stratify=y_wine
)

# Modèles à comparer
models = {
    'Arbre Simple': DecisionTreeClassifier(random_state=42),
```

```

'Arbre Optimisé': DecisionTreeClassifier(max_depth=5, min_samples_split=10, random_state=42),
'Forêt Aléatoire': RandomForestClassifier(n_estimators=100, random_state=42),
'Forêt Optimisée': RandomForestClassifier(
    n_estimators=200, max_depth=10, min_samples_split=5,
    max_features='sqrt', random_state=42
)
}

# Évaluation des modèles
results = {}
for name, model in models.items():
    # Validation croisée
    cv_scores = cross_val_score(model, X_train, y_train, cv=5)

    # Entraînement et test
    model.fit(X_train, y_train)
    train_score = model.score(X_train, y_train)
    test_score = model.score(X_test, y_test)

    results[name] = {
        'CV_Mean': cv_scores.mean(),
        'CV_Std': cv_scores.std(),
        'Train_Score': train_score,
        'Test_Score': test_score,
        'Overfitting': train_score - test_score
    }

# Tableau des résultats
df_results = pd.DataFrame(results).T
print("\nComparaison des modèles:")
print(df_results.round(3))

# Visualisation des résultats
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Scores de validation croisée
model_names = list(results.keys())
cv_means = [results[name]['CV_Mean'] for name in model_names]
cv_stds = [results[name]['CV_Std'] for name in model_names]

axes[0].bar(model_names, cv_means, yerr=cv_stds, capsize=5, alpha=0.7)
axes[0].set_ylabel('Précision (CV)')
axes[0].set_title('Validation Croisée')
axes[0].tick_params(axis='x', rotation=45)

```

```

# Train vs Test
train_scores = [results[name]['Train_Score'] for name in model_names]
test_scores = [results[name]['Test_Score'] for name in model_names]

x_pos = np.arange(len(model_names))
width = 0.35

axes[1].bar(x_pos - width/2, train_scores, width, label='Train', alpha=0.7)
axes[1].bar(x_pos + width/2, test_scores, width, label='Test', alpha=0.7)
axes[1].set_ylabel('Précision')
axes[1].set_title('Train vs Test')
axes[1].set_xticks(x_pos)
axes[1].set_xticklabels(model_names, rotation=45)
axes[1].legend()

plt.tight_layout()
plt.show()

# Analyse détaillée du meilleur modèle
best_model_name = max(results.keys(), key=lambda x: results[x]['Test_Score'])
best_model = models[best_model_name]
best_model.fit(X_train, y_train)

print(f"\nMeilleur modèle: {best_model_name}")

# Matrice de confusion
from sklearn.metrics import confusion_matrix, classification_report
y_pred = best_model.predict(X_test)
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=wine.target_names, yticklabels=wine.target_names)
plt.xlabel('Prédiction')
plt.ylabel('Réalité')
plt.title(f'Matrice de confusion - {best_model_name}')
plt.show()

print(f"\nRapport de classification:")
print(classification_report(y_test, y_pred, target_names=wine.target_names))

# Importance des features (si applicable)
if hasattr(best_model, 'feature_importances_'):
    importance_df = pd.DataFrame({
        'Feature': wine.feature_names,
        'Importance': best_model.feature_importances_
    }).sort_values('Importance', ascending=False)

```

```

print(f"\nTop 10 features les plus importantes:")
print(importance_df.head(10))

# Visualisation
plt.figure(figsize=(10, 8))
sns.barplot(data=importance_df.head(10), y='Feature', x='Importance')
plt.title(f'Top 10 Features - {best_model_name}')
plt.tight_layout()
plt.show()

```

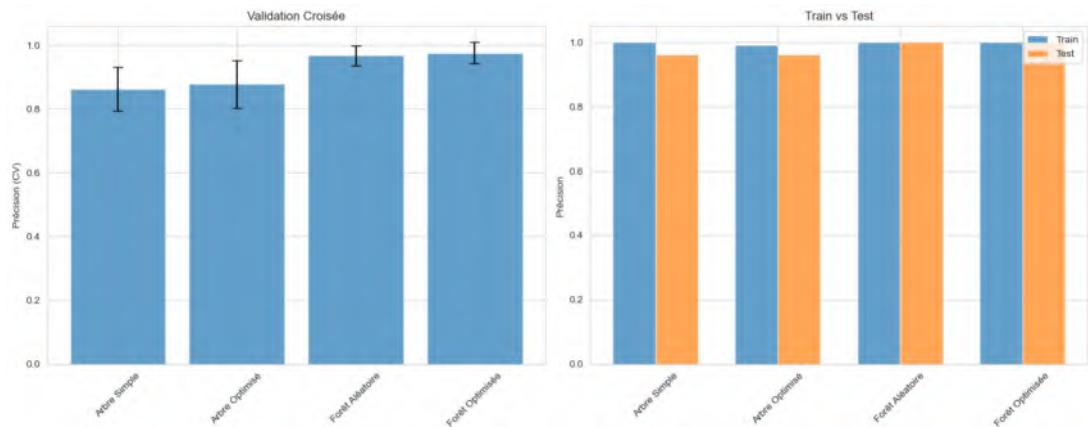
==== EXEMPLE PRATIQUE COMPLET ====

Dataset Vin: 178 échantillons, 13 features

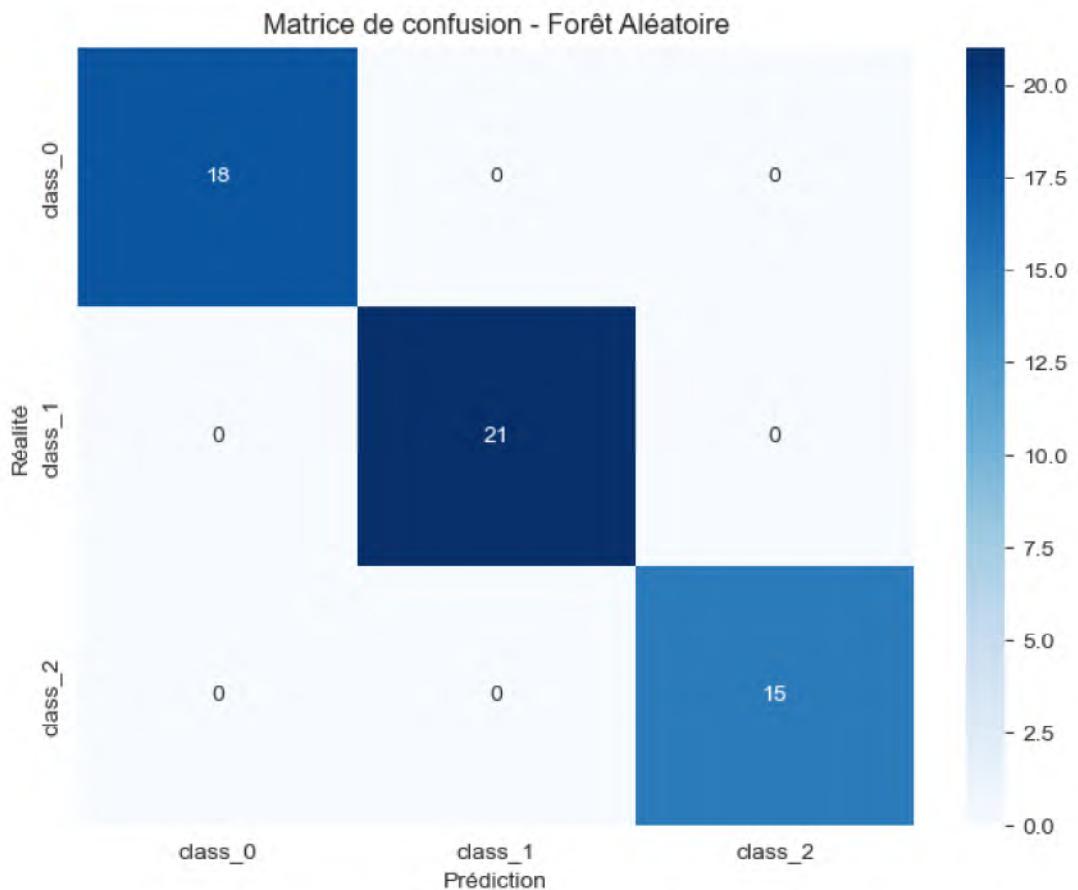
Classes: ['class_0' 'class_1' 'class_2']

Comparaison des modèles:

	CV_Mean	CV_Std	Train_Score	Test_Score	Overfitting
Arbre Simple	0.862	0.068	1.000	0.963	0.037
Arbre Optimisé	0.878	0.075	0.992	0.963	0.029
Forêt Aléatoire	0.967	0.031	1.000	1.000	0.000
Forêt Optimisée	0.975	0.033	1.000	1.000	0.000



Meilleur modèle: Forêt Aléatoire



Rapport de classification:

	precision	recall	f1-score	support
class_0	1.00	1.00	1.00	18
class_1	1.00	1.00	1.00	21
class_2	1.00	1.00	1.00	15
accuracy			1.00	54
macro avg	1.00	1.00	1.00	54
weighted avg	1.00	1.00	1.00	54

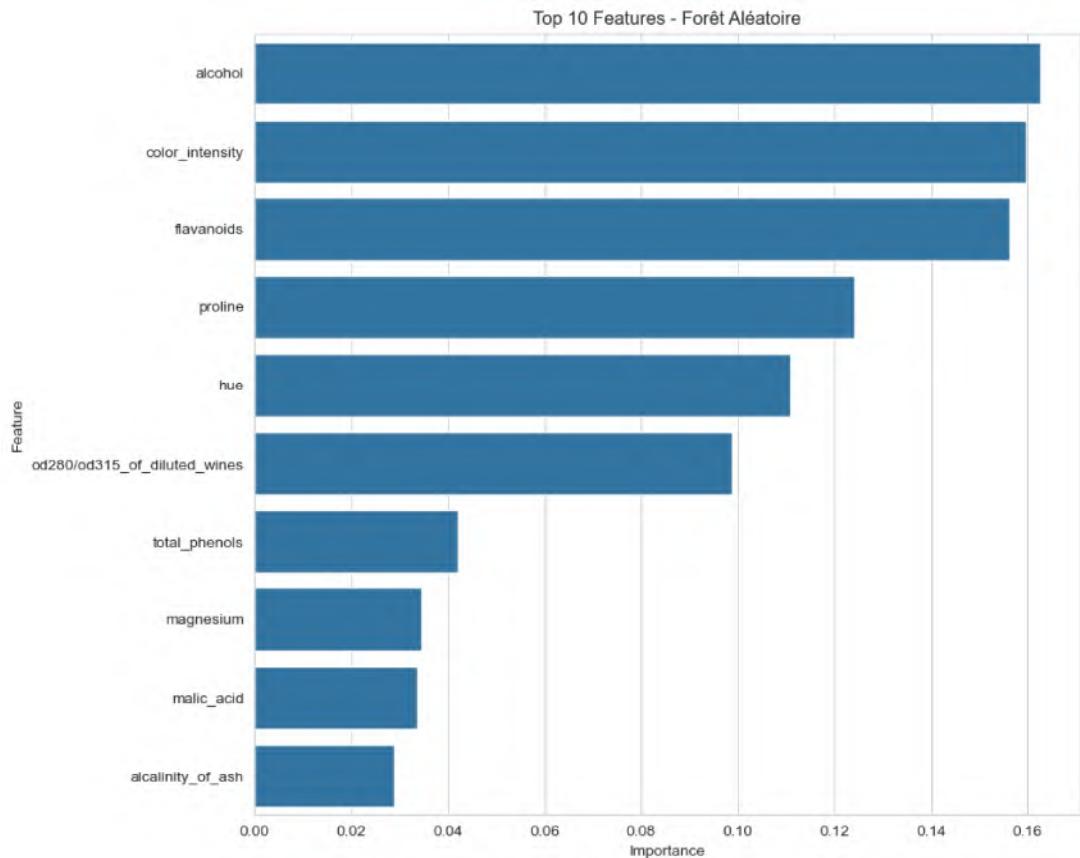
Top 10 features les plus importantes:

	Feature	Importance
0	alcohol	0.162715
9	color_intensity	0.159719
6	flavanoids	0.156283

```

12           proline  0.124081
10           hue     0.111019
11 od280/od315_of_diluted_wines 0.098852
5            total_phenols 0.042004
4            magnesium 0.034591
1            malic_acid 0.033700
3            alcalinity_of_ash 0.028841

```



1.7.3 SVM (Support Vector Machines)

Les Support Vector Machines sont des algorithmes puissants qui trouvent un hyperplan optimal pour séparer les classes en maximisant la marge entre elles.

Théorie et visualisation

```

[114]: print("== SUPPORT VECTOR MACHINES (SVM) ==")

from sklearn.svm import SVC, SVR
from sklearn.datasets import make_blobs, make_circles

```

```

print"""

PRINCIPE DES SVM:
1. Trouver l'hyperplan qui sépare les classes avec la marge maximale
2. Les "support vectors" sont les points les plus proches de la frontière
3. Seuls les support vectors influencent la décision
4. Kernel trick pour les données non-linéairement séparables

TYPES DE KERNELS:
• Linéaire:  $K(x,y) = x \cdot y$ 
• Polynomial:  $K(x,y) = (x \cdot y + r)^d$ 
• RBF (Radial):  $K(x,y) = \exp(-\|x-y\|^2)$ 
• Sigmoid:  $K(x,y) = \tanh(x \cdot y + r)$ 

AVANTAGES:
• Efficace en haute dimension
• Mémoire efficace (utilise seulement les support vectors)
• Polyvalent (différents kernels)

INCONVÉNIENTS:
• Lent sur gros datasets
• Sensible aux features non normalisées
• Pas de probabilités directes
""")

# Données linéairement séparables
np.random.seed(42)
X_linear, y_linear = make_blobs(n_samples=100, centers=2, n_features=2,
                                 center_box=(-3, 3), cluster_std=1, ↴
                                 random_state=42)

# Données non-linéairement séparables (cercles)
X_circles, y_circles = make_circles(n_samples=100, noise=0.1, factor=0.3, ↴
                                      random_state=42)

# Visualisation des datasets
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

axes[0].scatter(X_linear[:, 0], X_linear[:, 1], c=y_linear, cmap='viridis')
axes[0].set_title('Données linéairement séparables')
axes[0].set_xlabel('Feature 1')
axes[0].set_ylabel('Feature 2')

axes[1].scatter(X_circles[:, 0], X_circles[:, 1], c=y_circles, cmap='viridis')
axes[1].set_title('Données non-linéairement séparables')
axes[1].set_xlabel('Feature 1')
axes[1].set_ylabel('Feature 2')

```

```

plt.tight_layout()
plt.show()

# Fonction pour visualiser SVM avec frontières et support vectors
def plot_svm_decision_boundary(X, y, model, title):
    plt.figure(figsize=(10, 8))

    # Grille pour la frontière de décision
    h = 0.02
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                          np.arange(y_min, y_max, h))

    # Prédictions sur la grille
    Z = model.decision_function(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # Frontière de décision et marges
    plt.contour(xx, yy, Z, levels=[-1, 0, 1], linestyles=['--', '-'],
                colors=['red', 'black', 'red'], alpha=0.7)
    plt.contourf(xx, yy, Z, levels=50, alpha=0.3, cmap='viridis')

    # Points de données
    scatter = plt.scatter(X[:, 0], X[:, 1], c=y, cmap='viridis',
                          edgecolors='black')

    # Support vectors
    plt.scatter(model.support_vectors_[:, 0], model.support_vectors_[:, 1],
                s=200, facecolors='none', edgecolors='red', linewidths=2,
                label=f'Support Vectors ({len(model.support_vectors_)})')

    plt.title(title)
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.legend()
    plt.colorbar(scatter)
    plt.show()

# SVM linéaire
print("\n==== SVM LINÉAIRE ====")
svm_linear = SVC(kernel='linear', C=1.0)
svm_linear.fit(X_linear, y_linear)

plot_svm_decision_boundary(X_linear, y_linear, svm_linear,
                           'SVM Linéaire - Données séparables')

```

```

print(f"Nombre de support vectors: {len(svm_linear.support_vectors_)}")
print(f"Précision: {svm_linear.score(X_linear, y_linear):.3f}")

# SVM avec kernel RBF
print("\n==== SVM AVEC KERNEL RBF ====")
svm_rbf = SVC(kernel='rbf', C=1.0, gamma='scale')
svm_rbf.fit(X_circles, y_circles)

plot_svm_decision_boundary(X_circles, y_circles, svm_rbf,
                           'SVM RBF - Données non-linéaires')

print(f"Nombre de support vectors: {len(svm_rbf.support_vectors_)}")
print(f"Précision: {svm_rbf.score(X_circles, y_circles):.3f}")

```

==== SUPPORT VECTOR MACHINES (SVM) ====

PRINCIPE DES SVM:

1. Trouver l'hyperplan qui sépare les classes avec la marge maximale
2. Les "support vectors" sont les points les plus proches de la frontière
3. Seuls les support vectors influencent la décision
4. Kernel trick pour les données non-linéairement séparables

TYPES DE KERNELS:

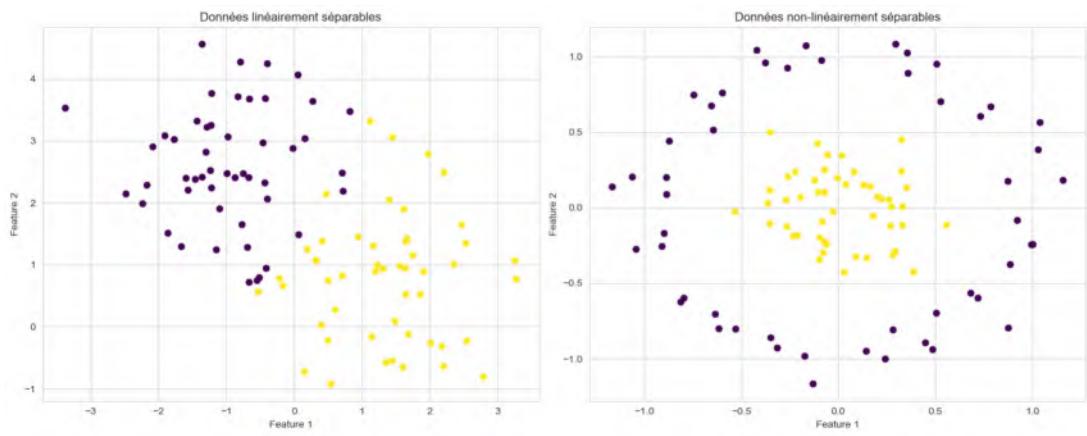
- Linéaire: $K(x,y) = x \cdot y$
- Polynomial: $K(x,y) = (x \cdot y + r)^d$
- RBF (Radial): $K(x,y) = \exp(-\|x-y\|^2)$
- Sigmoid: $K(x,y) = \tanh(x \cdot y + r)$

AVANTAGES:

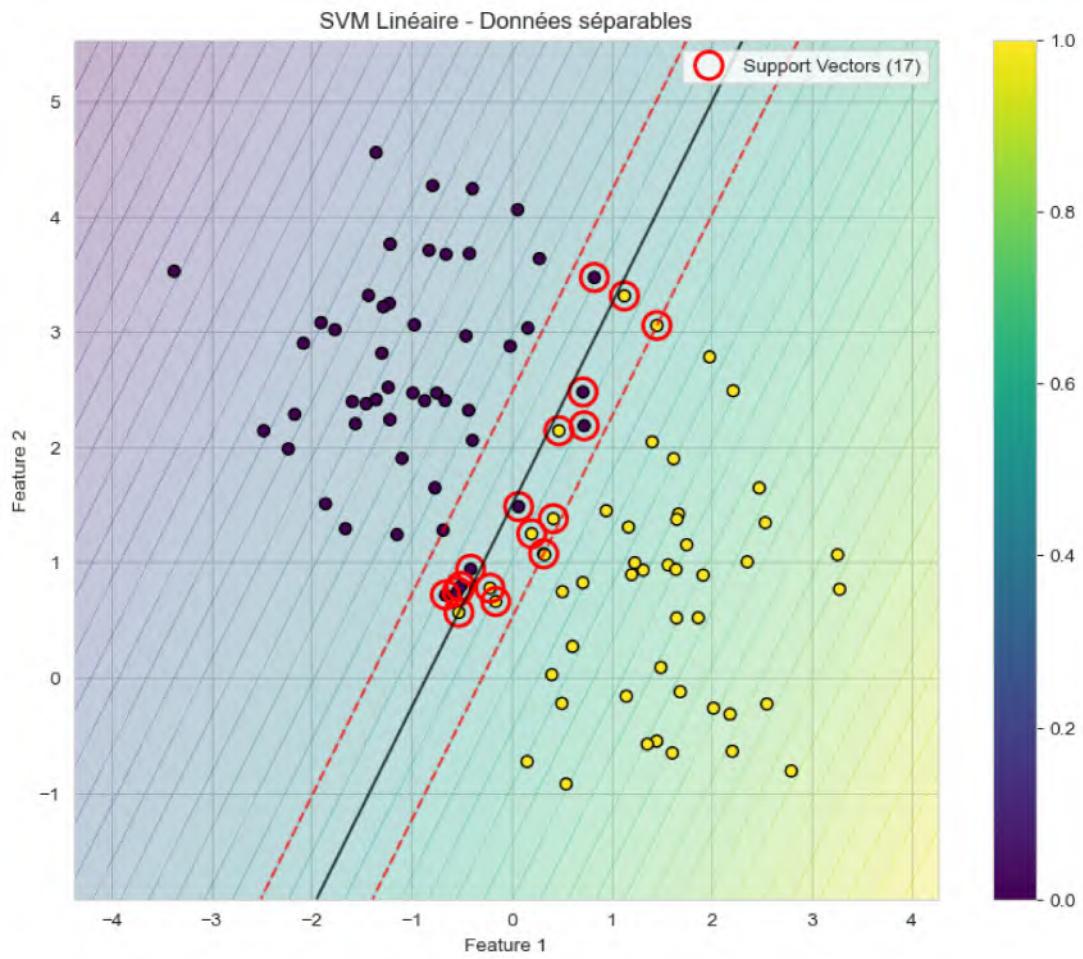
- Efficace en haute dimension
- Mémoire efficace (utilise seulement les support vectors)
- Polyvalent (différents kernels)

INCONVÉNIENTS:

- Lent sur gros datasets
- Sensible aux features non normalisées
- Pas de probabilités directes

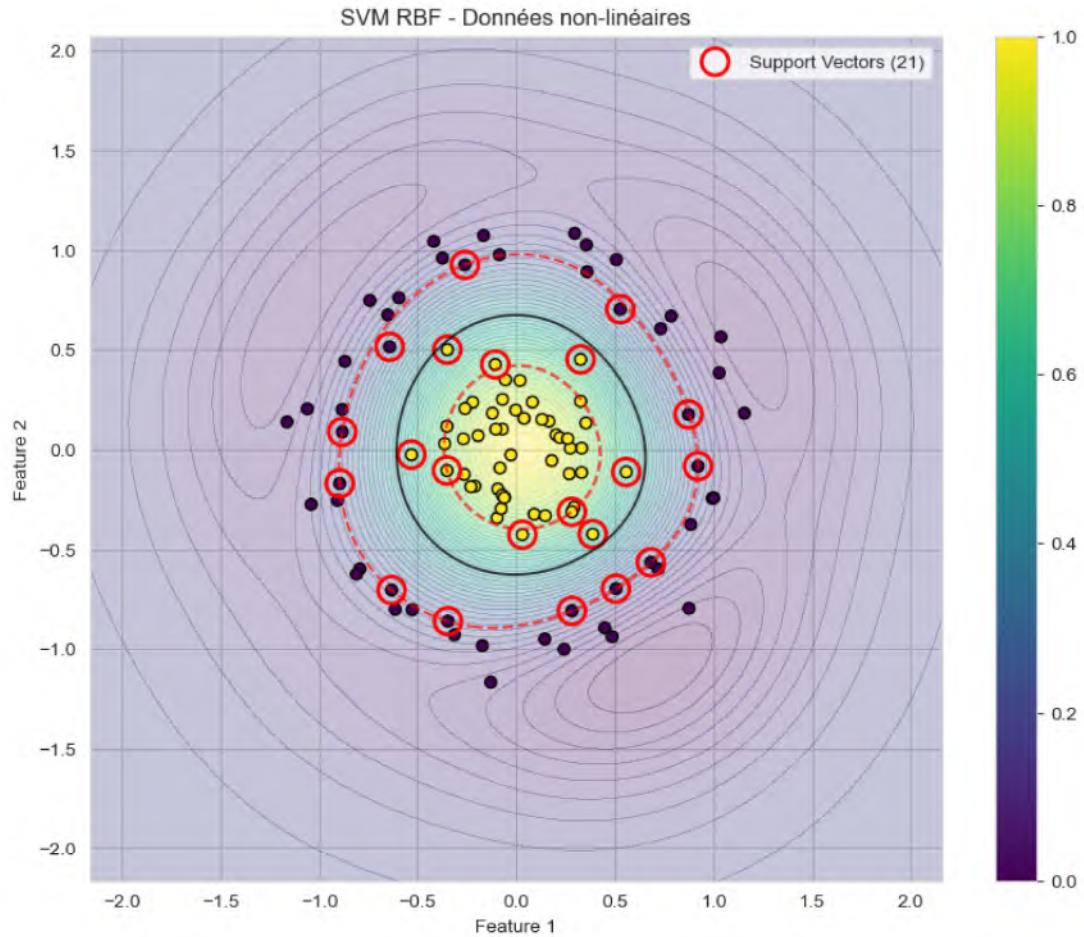


==== SVM LINÉAIRE ====



Nombre de support vectors: 17
Précision: 0.970

==== SVM AVEC KERNEL RBF ===



Nombre de support vectors: 21
Précision: 1.000

Impact des hyperparamètres

```
[116]: print("\n==== IMPACT DES HYPERPARAMÈTRES ===")  
  
# Test avec différents paramètres C et gamma  
C_values = [0.1, 1, 10, 100]  
gamma_values = [0.01, 0.1, 1, 10]
```

```

fig, axes = plt.subplots(len(C_values), len(gamma_values), figsize=(20, 16))

for i, C in enumerate(C_values):
    for j, gamma in enumerate(gamma_values):
        # Entraînement du modèle
        svm_temp = SVC(kernel='rbf', C=C, gamma=gamma)
        svm_temp.fit(X_circles, y_circles)

        # Visualisation rapide de la frontière
        h = 0.02
        x_min, x_max = X_circles[:, 0].min() - 1, X_circles[:, 0].max() + 1
        y_min, y_max = X_circles[:, 1].min() - 1, X_circles[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                             np.arange(y_min, y_max, h))

        Z = svm_temp.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)

        axes[i, j].contourf(xx, yy, Z, alpha=0.3, cmap='viridis')
        axes[i, j].scatter(X_circles[:, 0], X_circles[:, 1], c=y_circles, ▾
        ↪cmap='viridis')
        axes[i, j].set_title(f'C={C},\n= {gamma}\nScore: {svm_temp.\
        ↪score(X_circles, y_circles):.2f}')
        axes[i, j].set_xticks([])
        axes[i, j].set_yticks([])

plt.tight_layout()
plt.show()

# Analyse systématique avec validation croisée
print("\nAnalyse systématique des hyperparamètres:")

param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.01, 0.1, 1, 10]
}

from sklearn.model_selection import GridSearchCV

grid_search = GridSearchCV(
    SVC(kernel='rbf'),
    param_grid,
    cv=5,
    scoring='accuracy',
    n_jobs=-1
)

```

```

grid_search.fit(X_circles, y_circles)

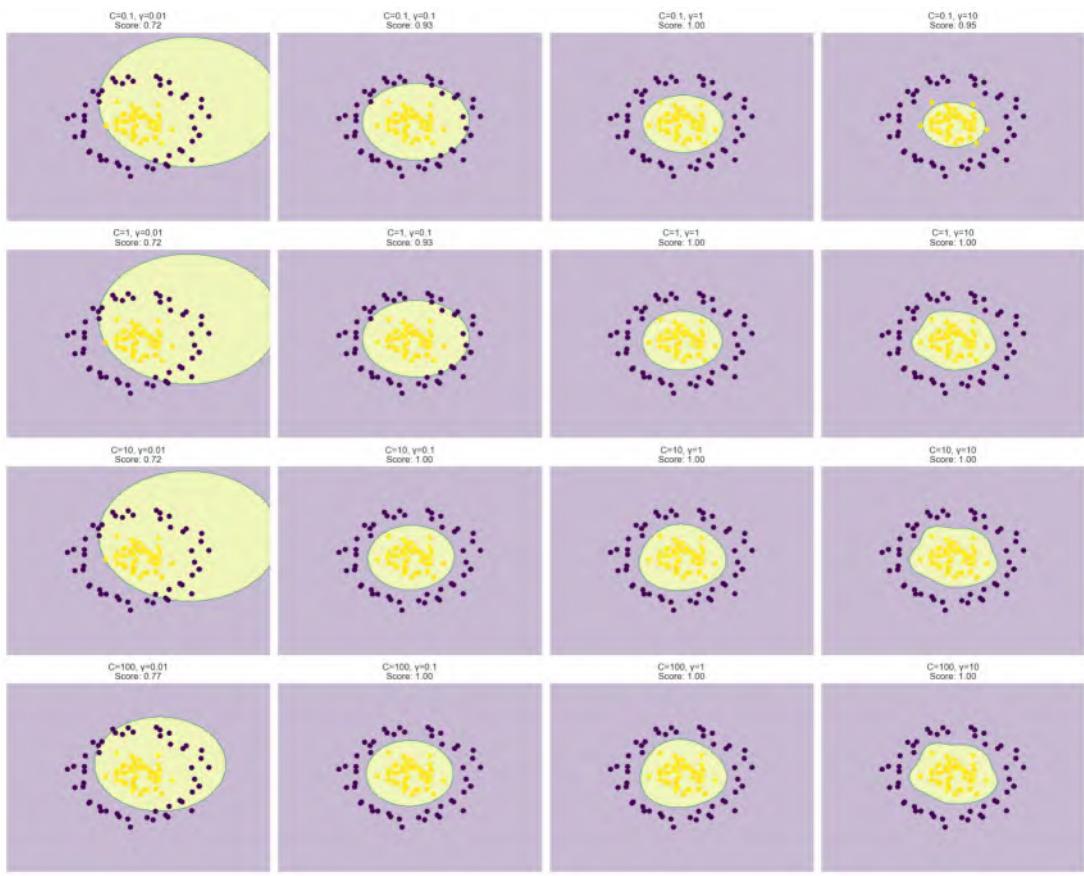
print(f"Meilleurs paramètres: {grid_search.best_params_}")
print(f"Meilleur score: {grid_search.best_score_:.3f}")

# Heatmap des résultats
scores_matrix = np.zeros((len(C_values), len(gamma_values)))
for i, (params, score) in enumerate(zip(grid_search.cv_results_['params'],
                                         grid_search.
                                         cv_results_['mean_test_score'])):
    c_idx = C_values.index(params['C'])
    g_idx = gamma_values.index(params['gamma'])
    scores_matrix[c_idx, g_idx] = score

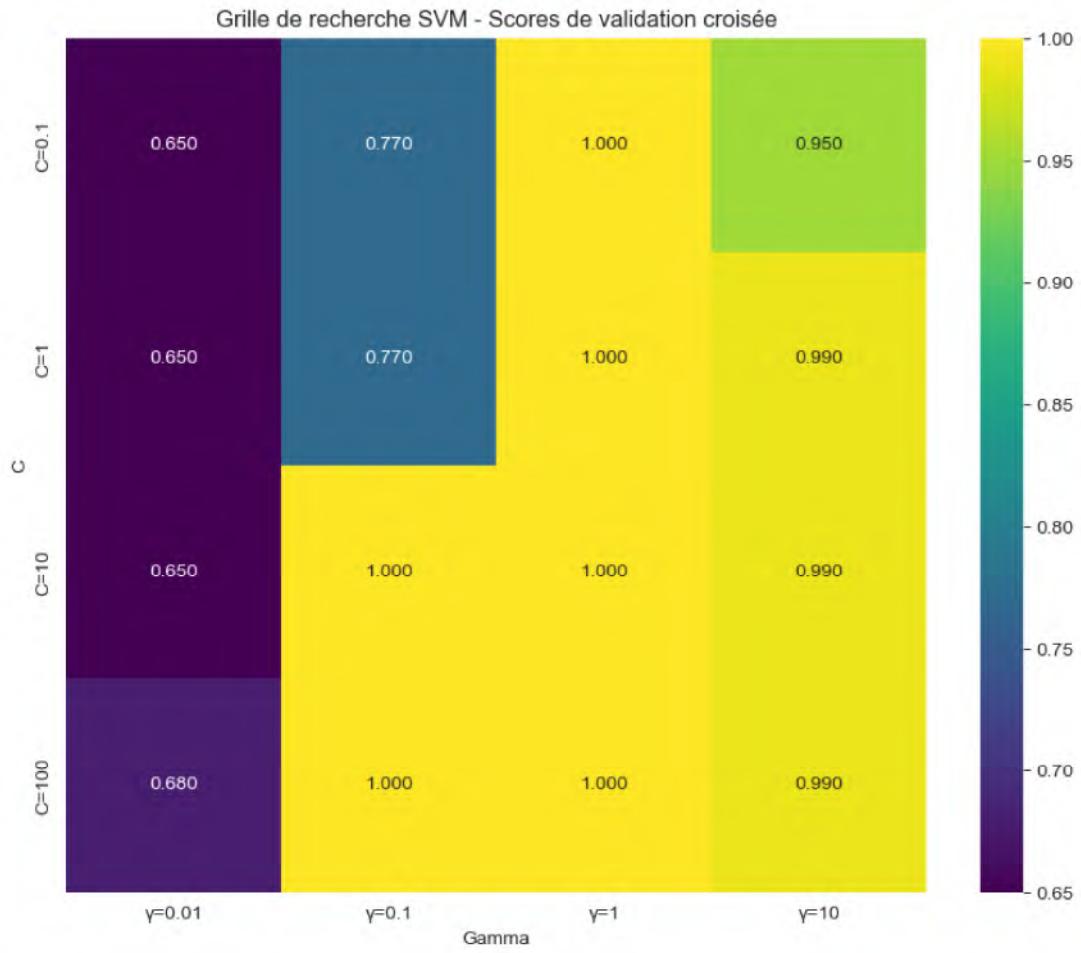
plt.figure(figsize=(10, 8))
sns.heatmap(scores_matrix,
            xticklabels=[f'={g}' for g in gamma_values],
            yticklabels=[f'C={c}' for c in C_values],
            annot=True, fmt='.3f', cmap='viridis')
plt.title('Grille de recherche SVM - Scores de validation croisée')
plt.xlabel('Gamma')
plt.ylabel('C')
plt.show()

```

==== IMPACT DES HYPERPARAMÈTRES ===



Analyse systématique des hyperparamètres:
Meilleurs paramètres: {'C': 0.1, 'gamma': 1}
Meilleur score: 1.000



Comparaison des kernels

```
[121]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs, load_iris
from sklearn.cluster import KMeans
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score
import pandas as pd
from sklearn.metrics import silhouette_samples, silhouette_score

# Générer des données synthétiques pour le clustering
X_blobs, _ = make_blobs(n_samples=300, centers=4, random_state=42,
cluster_std=1.0)

# Fonction pour afficher l'analyse de silhouette
def plot_silhouette_analysis(X, labels, n_clusters):
```

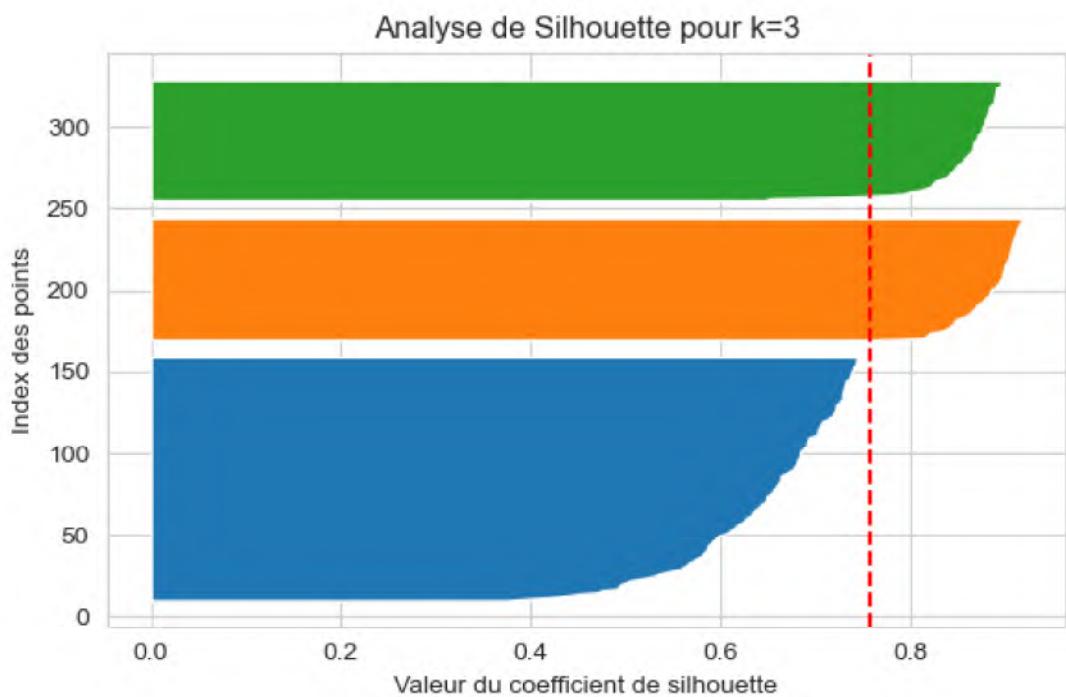
```

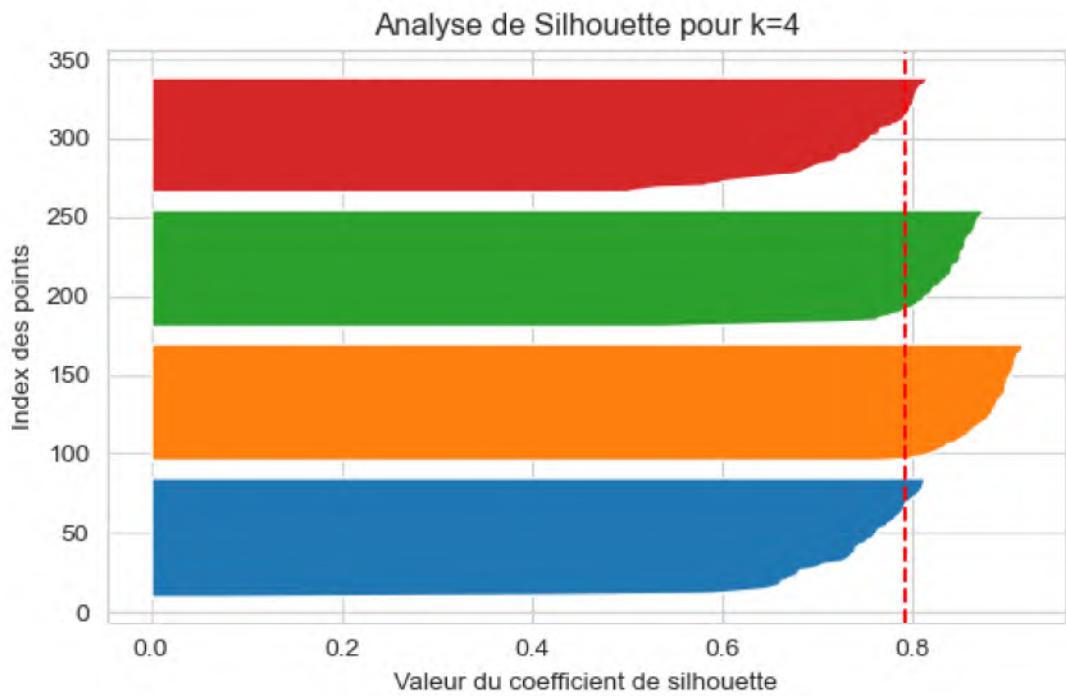
silhouette_vals = silhouette_samples(X, labels)
silhouette_avg = silhouette_score(X, labels)

fig, ax = plt.subplots(figsize=(6, 4))
y_lower = 10
for i in range(n_clusters):
    ith_cluster_silhouette_vals = silhouette_vals[labels == i]
    ith_cluster_silhouette_vals.sort()
    size_cluster_i = ith_cluster_silhouette_vals.shape[0]
    y_upper = y_lower + size_cluster_i
    ax.fill_betweenx(np.arange(y_lower, y_upper), 0, -1
                    ↵ith_cluster_silhouette_vals)
    y_lower = y_upper + 10
    ax.axvline(silhouette_avg, color="red", linestyle="--")
ax.set_title(f"Analyse de Silhouette pour k={n_clusters}")
ax.set_xlabel("Valeur du coefficient de silhouette")
ax.set_ylabel("Index des points")
plt.tight_layout()
plt.show()

# Exécution de l'analyse pour k=3 et k=4
for k in [3, 4]:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    cluster_labels = kmeans.fit_predict(X_blobs)
    plot_silhouette_analysis(X_blobs, cluster_labels, k)

```





Cas d'usage pratiques et limitations

```
[ ]: print("\n==== LIMITATIONS ET CAS D'USAGE ===")

# Démonstration des limitations du K-means
from sklearn.datasets import make_circles, make_moons

# 1. Clusters non-sphériques
X_moons, y_moons = make_moons(n_samples=200, noise=0.1, random_state=42)

# 2. Clusters de densités différentes
centers = [[0, 0], [3, 3]]
X_varied, y_varied = make_blobs(n_samples=200, centers=centers,
                                 cluster_std=[0.5, 2.0], random_state=42)

# 3. Clusters circulaires
X_circles, y_circles = make_circles(n_samples=200, noise=0.05, factor=0.3, random_state=42)

datasets = [
    (X_moons, y_moons, "Forme de lunes"),
    (X_varied, y_varied, "Densité de clusters"),
    (X_circles, y_circles, "Forme circulaire")]
```

```

(X_varied, y_varied, "Densités différentes"),
(X_circles, y_circles, "Cercles concentriques")
]

fig, axes = plt.subplots(3, 3, figsize=(18, 15))

for i, (X, y_true, title) in enumerate(datasets):
    # Données originales
    axes[i, 0].scatter(X[:, 0], X[:, 1], c=y_true, cmap='viridis')
    axes[i, 0].set_title(f'{title}\n(Clusters vrais)')

    # K-means
    kmeans = KMeans(n_clusters=2, random_state=42, n_init=10)
    y_kmeans = kmeans.fit_predict(X)

    axes[i, 1].scatter(X[:, 0], X[:, 1], c=y_kmeans, cmap='viridis')
    axes[i, 1].scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
                        marker='x', s=300, linewidths=3, color='red')
    axes[i, 1].set_title(f'K-means\nARI: {adjusted_rand_score(y_true, y_kmeans):.3f}')

    # Clustering hiérarchique (pour comparaison)
    from sklearn.cluster import AgglomerativeClustering
    hierarchical = AgglomerativeClustering(n_clusters=2)
    y_hierarchical = hierarchical.fit_predict(X)

    axes[i, 2].scatter(X[:, 0], X[:, 1], c=y_hierarchical, cmap='viridis')
    axes[i, 2].set_title(f'Clustering hiérarchique\nARI: {adjusted_rand_score(y_true, y_hierarchical):.3f}')

plt.tight_layout()
plt.show()

print("OBSERVATIONS:")
print("• K-means fonctionne bien avec des clusters sphériques et bien séparés")
print("• Il a des difficultés avec des formes complexes, des densités variables")
print("• D'autres algorithmes (clustering hiérarchique, DBSCAN) peuvent être plus adaptés")

```

Application sur données réelles

```
[139]: print("\n==== APPLICATION SUR DONNÉES RÉELLES ====")

# Utilisation du dataset Iris pour segmentation de clients
iris = load_iris()
```

```

X_iris = iris.data
feature_names = iris.feature_names

# Normalisation des données (importante pour K-means)
scaler = StandardScaler()
X_iris_scaled = scaler.fit_transform(X_iris)

print(f"Dataset Iris: {X_iris.shape[0]} échantillons, {X_iris.shape[1]}  

    ↪features")
print("Features:", feature_names)

# Détermination du nombre optimal de clusters
k_range = range(1, 8)
inertias_iris = []
silhouette_scores_iris = []

for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(X_iris_scaled)
    inertias_iris.append(kmeans.inertia_)

    if k > 1:
        silhouette_scores_iris.append(silhouette_score(X_iris_scaled, kmeans.  

            ↪labels_))
    else:
        silhouette_scores_iris.append(0)

# Visualisation des métriques
fig, axes = plt.subplots(1, 2, figsize=(15, 6))

axes[0].plot(k_range, inertias_iris, 'bo-')
axes[0].set_xlabel('Nombre de clusters (k)')
axes[0].set_ylabel('Inertie')
axes[0].set_title('Méthode du coude - Dataset Iris')
axes[0].grid(True, alpha=0.3)

axes[1].plot(k_range[1:], silhouette_scores_iris[1:], 'ro-')
axes[1].set_xlabel('Nombre de clusters (k)')
axes[1].set_ylabel('Score silhouette')
axes[1].set_title('Score silhouette - Dataset Iris')
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# K-means avec k=3 (nous savons qu'il y a 3 espèces)
kmeans_final = KMeans(n_clusters=3, random_state=42, n_init=10)

```

```

y_pred_iris = kmeans_final.fit_predict(X_iris_scaled)

# Comparaison avec les vraies classes
from sklearn.metrics import adjusted_rand_score
print(f"\nCOMPARAISON AVEC LES VRAIES CLASSES:")
print(f"ARI (Adjusted Rand Index): {adjusted_rand_score(iris.target, y_pred_iris):.3f}")
print(f"Score silhouette: {silhouette_score(X_iris_scaled, y_pred_iris):.3f}")

# Matrice de confusion
from sklearn.metrics import confusion_matrix
cm_iris = confusion_matrix(iris.target, y_pred_iris)

plt.figure(figsize=(8, 6))
sns.heatmap(cm_iris, annot=True, fmt='d', cmap='Blues',
            xticklabels=[f'Cluster {i}' for i in range(3)],
            yticklabels=iris.target_names)
plt.xlabel('Clusters prédits')
plt.ylabel('Espèces réelles')
plt.title('Matrice de confusion - K-means vs Vraies classes')
plt.show()

# Analyse des centres de clusters
centers_original = scaler.inverse_transform(kmeans_final.cluster_centers_)
df_centers = pd.DataFrame(centers_original, columns=feature_names)
df_centers.index = [f'Cluster {i}' for i in range(3)]

print(f"\nCENTRES DES CLUSTERS (échelle originale):")
print(df_centers.round(2))

# Visualisation 2D des clusters (PCA)
from sklearn.decomposition import PCA

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_iris_scaled)
centers_pca = pca.transform(kmeans_final.cluster_centers_)

fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Vraies classes
axes[0].scatter(X_pca[:, 0], X_pca[:, 1], c=iris.target, cmap='viridis')
axes[0].set_title('Vraies classes (PCA)')
axes[0].set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.1%} variance)')
axes[0].set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.1%} variance)')

# Clusters K-means
axes[1].scatter(X_pca[:, 0], X_pca[:, 1], c=y_pred_iris, cmap='viridis')

```

```

axes[1].scatter(centers_pca[:, 0], centers_pca[:, 1],
                 marker='x', s=300, linewidths=3, color='red', label='Centroïdes')
axes[1].set_title('Clusters K-means (PCA)')
axes[1].set_xlabel(f'PC1 ({pca.explained_variance_ratio_[0]:.1%} variance)')
axes[1].set_ylabel(f'PC2 ({pca.explained_variance_ratio_[1]:.1%} variance)')
axes[1].legend()

plt.tight_layout()
plt.show()

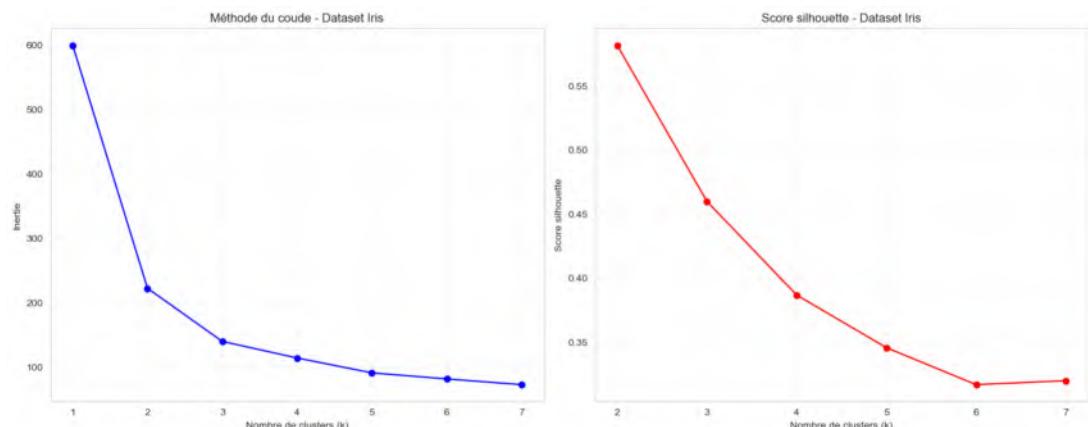
print(f"Variance expliquée par les 2 premières composantes: {pca.
      ↪explained_variance_ratio_.sum():.1%}")

```

==== APPLICATION SUR DONNÉES RÉELLES ===

Dataset Iris: 150 échantillons, 4 features

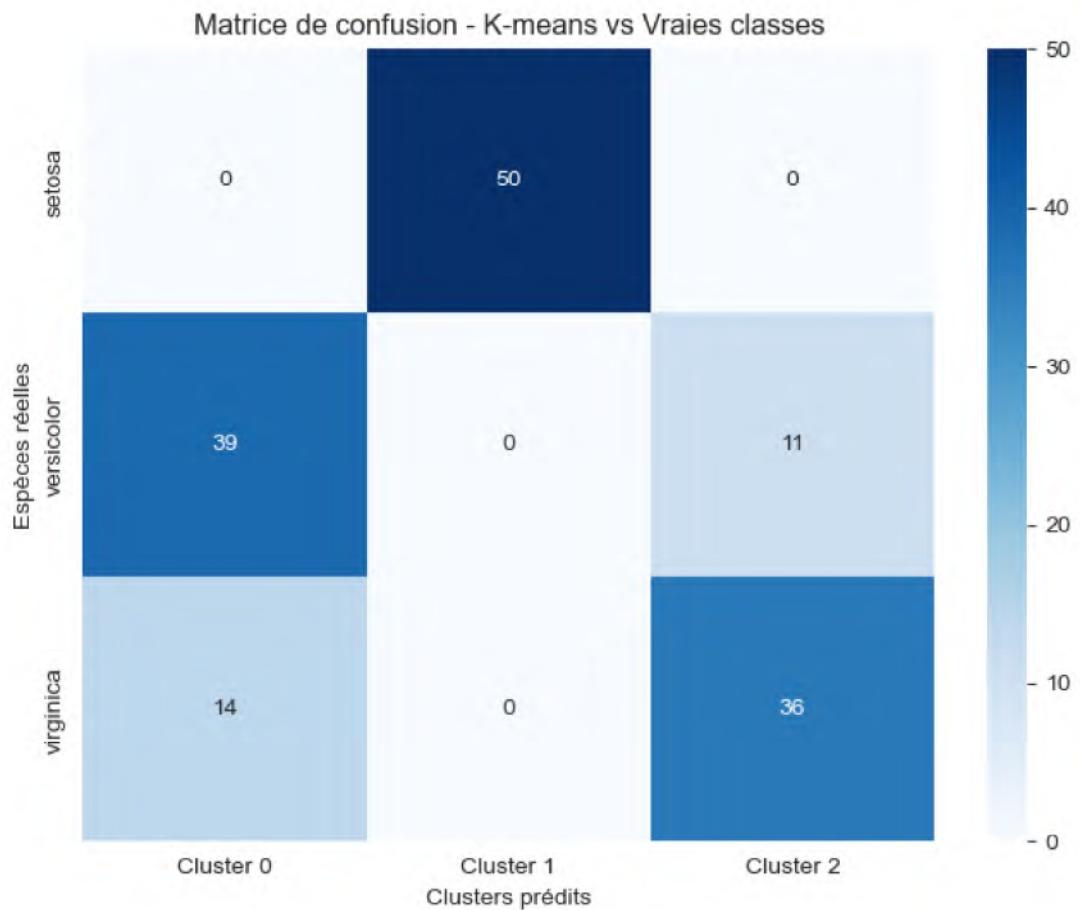
Features: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']



COMPARAISON AVEC LES VRAIES CLASSES:

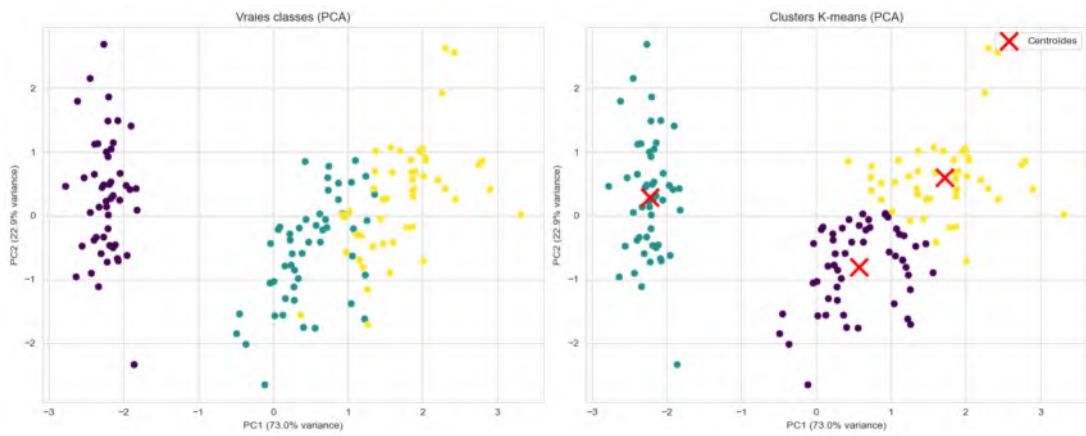
ARI (Adjusted Rand Index): 0.620

Score silhouette: 0.460



CENTRES DES CLUSTERS (échelle originale):

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
Cluster 0	5.80	2.67	4.37	
Cluster 1	5.01	3.43	1.46	
Cluster 2	6.78	3.10	5.51	
Cluster 0		1.41		
Cluster 1		0.25		
Cluster 2		1.97		



Variance expliquée par les 2 premières composantes: 95.8%

1.8 Ressources complémentaires

1.8.1 Datasets pour pratiquer

```
[141]: print("== DATASETS RECOMMANDÉS POUR LA PRATIQUE ==")

practice_datasets = {
    "Débutant": [
        {"nom": "Iris", "description": "Classification de fleurs", "source": "sklearn.datasets"}, 
        {"nom": "Boston Housing", "description": "Prédiction prix immobilier", "source": "sklearn.datasets"}, 
        {"nom": "Wine", "description": "Classification de vins", "source": "sklearn.datasets"}, 
        {"nom": "Titanic", "description": "Survie des passagers", "source": "Kaggle"}],
    "Intermédiaire": [
        {"nom": "California Housing", "description": "Prix immobilier Californie", "source": "sklearn.datasets"}, 
        {"nom": "Diabetes", "description": "Progression du diabète", "source": "sklearn.datasets"}, 
        {"nom": "Credit Card Fraud", "description": "Détection de fraude", "source": "Kaggle"}, 
        {"nom": "Movie Lens", "description": "Système de recommandation", "source": "GroupLens"}],
    "Avancé": []
}
```

```

        {"nom": "ImageNet", "description": "Classification d'images", "source": "Stanford"},  

        {"nom": "IMDB Reviews", "description": "Analyse de sentiment", "source": "Stanford"},  

        {"nom": "Stock Prices", "description": "Prédiction financière", "source": "Yahoo Finance"},  

        {"nom": "OpenML", "description": "Centaines de datasets", "source": "OpenML.org"}  

    ]  

}  
  

for level, datasets in practice_datasets.items():  

    print(f"\n{level.upper()}:")  

    for dataset in datasets:  

        print(f"  • {dataset['nom']}: {dataset['description']}")  

        print(f"  • {dataset['source']}")

```

==== DATASETS RECOMMANDÉS POUR LA PRATIQUE ===

DÉBUTANT:

- Iris: Classification de fleurs (sklearn.datasets)
- Boston Housing: Prédiction prix immobilier (sklearn.datasets)
- Wine: Classification de vins (sklearn.datasets)
- Titanic: Survie des passagers (Kaggle)

INTERMÉDIAIRE:

- California Housing: Prix immobilier Californie (sklearn.datasets)
- Diabetes: Progression du diabète (sklearn.datasets)
- Credit Card Fraud: Détection de fraude (Kaggle)
- Movie Lens: Système de recommandation (GroupLens)

AVANCÉ:

- ImageNet: Classification d'images (Stanford)
- IMDB Reviews: Analyse de sentiment (Stanford)
- Stock Prices: Prédiction financière (Yahoo Finance)
- OpenML: Centaines de datasets (OpenML.org)

1.8.2 Bibliothèques et outils avancés

[143]: `print("\n==== BIBLIOTHÈQUES ET OUTILS AVANCÉS ===")`

```

advanced_tools = {
    "Machine Learning": [
        "XGBoost - Gradient boosting optimisé",
        "LightGBM - Gradient boosting rapide",
        "CatBoost - Gestion automatique variables catégorielles",
        "Optuna - Optimisation hyperparamètres",
    ]
}

```

```

        "SHAP - Explainable AI",
        "MLflow - Gestion expériences ML"
    ],
    "Deep Learning": [
        "TensorFlow/Keras - Framework complet",
        "PyTorch - Recherche et prototypage",
        "Hugging Face - Modèles pré-entraînés NLP",
        "OpenCV - Computer Vision",
        "Pytorch Lightning - PyTorch simplifié"
    ],
    "Données": [
        "Dask - Pandas à grande échelle",
        "Polars - Alternative rapide à Pandas",
        "Apache Spark - Big Data processing",
        "Great Expectations - Qualité des données",
        "DVC - Versioning des données"
    ],
    "Visualisation": [
        "Plotly - Graphiques interactifs",
        "Bokeh - Visualisations web",
        "Altair - Grammar of graphics",
        "Streamlit - Applications web rapides",
        "Dash - Dashboards interactifs"
    ],
    "Déploiement": [
        "FastAPI - APIs ML rapides",
        "Docker - Containerisation",
        "Kubernetes - Orchestration",
        "AWS SageMaker - ML cloud",
        "Google Cloud AI - Services ML",
        "Azure ML - Platform Microsoft"
    ]
}

for category, tools in advanced_tools.items():
    print(f"\n{category.upper()}:")
    for tool in tools:
        print(f"    • {tool}")

```

==== BIBLIOTHÈQUES ET Outils AVANCÉS ===

MACHINE LEARNING:

- XGBoost - Gradient boosting optimisé
- LightGBM - Gradient boosting rapide
- CatBoost - Gestion automatique variables catégorielles
- Optuna - Optimisation hyperparamètres
- SHAP - Explainable AI

- MLflow - Gestion expériences ML

DEEP LEARNING:

- TensorFlow/Keras - Framework complet
- PyTorch - Recherche et prototypage
- Hugging Face - Modèles pré-entraînés NLP
- OpenCV - Computer Vision
- Pytorch Lightning - PyTorch simplifié

DONNÉES:

- Dask - Pandas à grande échelle
- Polars - Alternative rapide à Pandas
- Apache Spark - Big Data processing
- Great Expectations - Qualité des données
- DVC - Versioning des données

VISUALISATION:

- Plotly - Graphiques interactifs
- Bokeh - Visualisations web
- Altair - Grammar of graphics
- Streamlit - Applications web rapides
- Dash - Dashboards interactifs

DÉPLOIEMENT:

- FastAPI - APIs ML rapides
- Docker - Containerisation
- Kubernetes - Orchestration
- AWS SageMaker - ML cloud
- Google Cloud AI - Services ML
- Azure ML - Platform Microsoft

1.8.3 Ressources d'apprentissage

```
[148]: print("\n==== RESSOURCES D'APPRENTISSAGE RECOMMANDÉES ===")
```

```
resources = {
    "Livres": [
        "Hands-On Machine Learning - Aurélien Géron",
        "Pattern Recognition and ML - Christopher Bishop",
        "The Elements of Statistical Learning - Hastie, Tibshirani, Friedman",
        "Python Data Science Handbook - Jake VanderPlas",
        "Feature Engineering for ML - Alice Zheng, Amanda Casari"
    ],
    "Cours en ligne": [
        "Coursera - Machine Learning par Andrew Ng",
        "edX - MIT Introduction to ML",
        "Udacity - Machine Learning Engineer",
    ]
}
```

```

        "Fast.ai - Practical Deep Learning",
        "Kaggle Learn - Micro-cours gratuits"
    ],
    "Communautés": [
        "Kaggle - Compétitions et datasets",
        "GitHub - Code et projets open source",
        "Stack Overflow - Questions techniques",
        "Reddit r/MachineLearning - Discussions",
        "Papers With Code - Articles avec implémentations"
    ],
    "Conférences": [
        "NeurIPS - Recherche de pointe",
        "ICML - International Conference on ML",
        "PyData - Conférences Python data science",
        "Strata Data - Conférence O'Reilly",
        "MLConf - Conférences ML pratiques"
    ]
}

for category, items in resources.items():
    print(f"\n{category.upper()}:")
    for item in items:
        print(f"  • {item}")

```

==== RESSOURCES D'APPRENTISSAGE RECOMMANDÉES ===

LIVRES:

- Hands-On Machine Learning - Aurélien Géron
- Pattern Recognition and ML - Christopher Bishop
- The Elements of Statistical Learning - Hastie, Tibshirani, Friedman
- Python Data Science Handbook - Jake VanderPlas
- Feature Engineering for ML - Alice Zheng, Amanda Casari

COURS EN LIGNE:

- Coursera - Machine Learning par Andrew Ng
- edX - MIT Introduction to ML
- Udacity - Machine Learning Engineer
- Fast.ai - Practical Deep Learning
- Kaggle Learn - Micro-cours gratuits

COMMUNAUTÉS:

- Kaggle - Compétitions et datasets
- GitHub - Code et projets open source
- Stack Overflow - Questions techniques
- Reddit r/MachineLearning - Discussions
- Papers With Code - Articles avec implémentations

CONFÉRENCES :

- NeurIPS - Recherche de pointe
- ICML - International Conference on ML
- PyData - Conférences Python data science
- Strata Data - Conférence O'Reilly
- MLConf - Conférences ML pratiques

1.8.4 Evaluation finale

```
[151]: print("== ÉVALUATION FINALE ET SYNTHÈSE ==")
```

```
print("""
```

RÉCAPITULATIF DES ALGORITHMES ÉTUDIÉS:

1. RÉGRESSION LINÉAIRE

- + Modélise relations linéaires
- + Très interprétable
- Limité aux relations linéaires

2. RÉGRESSION LOGISTIQUE

- + Classification probabiliste
- + Interprétable (odds ratios)
- Assume relations linéaires en logit

3. K-NEAREST NEIGHBORS (KNN)

- + Simple, pas d'hypothèses sur les données
- + Fonctionne avec données complexes
- Coûteux, sensible à la dimension

4. ARBRES DE DÉCISION

- + Très interprétables
- + Gère interactions automatiquement
- Tendance au surapprentissage

5. FORêTS ALÉATOIRES

- + Réduit surapprentissage des arbres
- + Importance des features
- Moins interprétable qu'un arbre unique

6. SUPPORT VECTOR MACHINES (SVM)

- + Efficace en haute dimension
- + Kernel trick pour non-linéarité
- Lent sur gros datasets

7. K-MEANS CLUSTERING

- + Simple clustering non-supervisé
- + Efficace pour clusters sphériques

- Besoin de choisir k , assume formes sphériques

GUIDE DE CHOIX D'ALGORITHME:

""")

```
# Tableau de comparaison final
#### Tableau de comparaison des algorithmes
```

```
comparison_data = {
    'Algorithme': ['Régression Linéaire', 'Régression Logistique', 'KNN',
                    'Arbre Décision', 'Forêt Aléatoire', 'SVM', 'K-Means'],
    'Type': ['Régression', 'Classification', 'Les deux', 'Les deux', 'Les deux',
             'Les deux', 'Clustering'],
    'Interprétabilité': [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    'Performance': [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    'Vitesse': [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
    'Gros datasets': [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
}
```

```
df_comparaison = pd.DataFrame(comparison_data)
print(df_comparaison)
```

```
df_comparaison = pd.DataFrame(comparison_data)
print(df_comparaison.to_string(index=False))
```

```
print(f"""
```

CONCEPTS CLÉS MAÎTRISÉS:

- Workflow Machine Learning complet
- Validation croisée et métriques d'évaluation
- Compromis biais-variance
- Préparation des données et feature engineering
- Hyperparamètres et optimisation
- Visualisation des résultats
- Interprétation des modèles
- Clustering non-supervisé

OUTILS ET BIBLIOTHÈQUES:

- Scikit-learn pour tous les algorithmes ML
- Pandas pour manipulation des données
- NumPy pour calculs numériques
- Matplotlib/Seaborn pour visualisation
- Pipeline pour flux de traitement
- GridSearchCV pour optimisation

PROCHAINES ÉTAPES RECOMMANDÉES:

1. Deep Learning avec TensorFlow/PyTorch

2. Traitement du langage naturel (NLP)
3. Computer Vision
4. Séries temporelles
5. MLOps et déploiement de modèles
6. Explainable AI (XAI)

PROJET FINAL SUGGÉRÉ:

Créer un projet complet de bout en bout:

- Collecte et nettoyage des données
 - Analyse exploratoire approfondie
 - Comparaison de plusieurs algorithmes
 - Optimisation des hyperparamètres
 - Validation rigoureuse
 - Interprétation des résultats
 - Présentation des conclusions business
- """)

```
print("\n" + "="*80)
print(" FÉLICITATIONS ! FORMATION TERMINÉE AVEC SUCCÈS ! ")
print("=*80)
```

==== ÉVALUATION FINALE ET SYNTHÈSE ===

RÉCAPITULATIF DES ALGORITHMES ÉTUDIÉS:

1. RÉGRESSION LINÉAIRE
 - + Modélise relations linéaires
 - + Très interprétable
 - Limité aux relations linéaires
2. RÉGRESSION LOGISTIQUE
 - + Classification probabiliste
 - + Interprétable (odds ratios)
 - Assume relations linéaires en logit
3. K-NEAREST NEIGHBORS (KNN)
 - + Simple, pas d'hypothèses sur les données
 - + Fonctionne avec données complexes
 - Coûteux, sensible à la dimension
4. ARBRES DE DÉCISION
 - + Très interprétables
 - + Gère interactions automatiquement
 - Tendance au surapprentissage
5. FORÊTS ALÉATOIRES
 - + Réduit surapprentissage des arbres
 - + Importance des features

- Moins interprétable qu'arbre unique
6. SUPPORT VECTOR MACHINES (SVM)
- + Efficace en haute dimension
 - + Kernel trick pour non-linéarité
 - Lent sur gros datasets
7. K-MEANS CLUSTERING
- + Simple clustering non-supervisé
 - + Efficace pour clusters sphériques
 - Besoin de choisir k , assume formes sphériques

GUIDE DE CHOIX D'ALGORITHME:

	Algorithme	Type	Interprétabilité	Performance	Vitesse	\
0	Régression Linéaire	Régression				
1	Régression Logistique	Classification				
2	KNN	Les deux				
3	Arbre Décision	Les deux				
4	Forêt Aléatoire	Les deux				
5	SVM	Les deux				
6	K-Means	Clustering				

Gros datasets	Algorithme	Type	Interprétabilité	Performance	Vitesse	Gros
0	Régression Linéaire	Régression				
1						
2						
3						
4						
5						
6						

Algorithme	Type	Interprétabilité	Performance	Vitesse	Gros
Régression Linéaire	Régression				
Régression Logistique	Classification				
KNN	Les deux				
Arbre Décision	Les deux				
Forêt Aléatoire	Les deux				
SVM	Les deux				
K-Means	Clustering				

CONCEPTS CLÉS MAÎTRISÉS:

- Workflow Machine Learning complet
- Validation croisée et métriques d'évaluation
- Compromis biais-variance
- Préparation des données et feature engineering
- Hyperparamètres et optimisation
- Visualisation des résultats
- Interprétation des modèles
- Clustering non-supervisé

OUTILS ET BIBLIOTHÈQUES:

- Scikit-learn pour tous les algorithmes ML
- Pandas pour manipulation des données
- NumPy pour calculs numériques
- Matplotlib/Seaborn pour visualisation
- Pipeline pour flux de traitement
- GridSearchCV pour optimisation

PROCHAINES ÉTAPES RECOMMANDÉES:

1. Deep Learning avec TensorFlow/PyTorch
2. Traitement du langage naturel (NLP)
3. Computer Vision
4. Séries temporelles
5. MLOps et déploiement de modèles
6. Explainable AI (XAI)

PROJET FINAL SUGGÉRÉ:

Créer un projet complet de bout en bout:

- Collecte et nettoyage des données
- Analyse exploratoire approfondie
- Comparaison de plusieurs algorithmes
- Optimisation des hyperparamètres
- Validation rigoureuse
- Interprétation des résultats
- Présentation des conclusions business

=====

FÉLICITATIONS ! FORMATION TERMINÉE AVEC SUCCÈS !

=====