LISV Laboratoire d'ingénierie des systèmes de Versailles
UVSQ université PARIS-SACLAY

Apprentissage automatique
TP 2 - Régression et Classification

Halim Djerroud

révision 1.0

— Exercices avec solutions —

Objectifs du TP
- Comprendre et appliquer la régression linéaire simple et multiple
- Maîtriser la régression logistique pour les problèmes de classification binaire
- Expérimenter avec différents classifieurs (K-NN, arbres de décision, SVM)
- Évaluer les performances des modèles avec des métriques appropriées
- Comparer différents algorithmes d'apprentissage supervisé

Outils et Environnement Le TP a été réalisé à l'aide des outils et environnements suivants :
- **Langage** : Python 3.x
- **Bibliothèques** : Pandas, NumPy, Matplotlib, Seaborn, Scikit-learn
- **IDE** : Jupyter Notebook / VS Code / Google Colab

Configuration de l'environnement :

```
conda activate ml
pip install scikit-learn pandas numpy matplotlib seaborn
```

1 Partie 1 : Régression Linéaire

1.1 Exercice 1.1 : Régression linéaire simple

Utilisez le dataset **California Housing** pour prédire le prix médian des maisons en fonction d'une seule variable.

1. Charger le dataset California Housing
2. Choisir la variable `MedInc` (revenu médian) comme variable explicative
3. Séparer les données en ensemble d'entraînement (80%) et de test (20%)
4. Entraîner un modèle de régression linéaire simple
5. Visualiser la droite de régression et les points de données
6. Calculer les métriques de performance : MSE, RMSE, MAE, R^2
7. Interpréter les coefficients obtenus (pente et ordonnée à l'origine)

Solution :

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

# 1. Charger le dataset
data = fetch_california_housing(as_frame=True)
df = data.frame
```

```python
# 2. Sélectionner une seule variable
X = df[['MedInc']].values
y = df['MedHouseVal'].values

# 3. Séparer les données
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 4. Entraîner le modèle
model = LinearRegression()
model.fit(X_train, y_train)

# 5. Prédictions
y_pred = model.predict(X_test)

# 6. Métriques
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f"Coefficients:")
print(f"  Pente (a): {model.coef_[0]:.4f}")
print(f"  Ordonnée à l'origine (b): {model.intercept_:.4f}")
print(f"\nMétriques de performance:")
print(f"  MSE: {mse:.4f}")
print(f"  RMSE: {rmse:.4f}")
print(f"  MAE: {mae:.4f}")
print(f"  R²: {r2:.4f}")

# 7. Visualisation
plt.figure(figsize=(10, 6))
plt.scatter(X_test, y_test, alpha=0.5, label='Données réelles')
plt.plot(X_test, y_pred, color='red', linewidth=2, label='Droite de régression')
plt.xlabel('Revenu médian (MedInc)')
plt.ylabel('Prix médian des maisons (MedHouseVal)')
plt.title('Régression linéaire simple')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

Interprétation des résultats :

- Le coefficient (pente) indique l'augmentation du prix des maisons pour chaque unité d'augmentation du revenu médian
- L'ordonnée à l'origine représente le prix de base théorique quand le revenu est nul
- Le R^2 proche de 0.47-0.50 indique que le revenu médian explique environ la moitié de la variance des prix

1.2 Exercice 1.2 : Régression linéaire multiple

Étendez l'exercice précédent en utilisant plusieurs variables explicatives.

1. Utiliser toutes les variables du dataset sauf la variable cible
2. Entraîner un modèle de régression linéaire multiple
3. Comparer les performances avec la régression simple
4. Analyser l'importance relative de chaque variable (coefficients)
5. Créer un graphique des valeurs prédites vs valeurs réelles
6. Analyser les résidus (erreurs de prédiction)

Solution :

```python
# 1. Préparer les données avec toutes les variables
X_multi = df.drop('MedHouseVal', axis=1).values
y = df['MedHouseVal'].values
```

```python
# 2. Séparer les données
X_train_multi, X_test_multi, y_train_multi, y_test_multi = train_test_split(
    X_multi, y, test_size=0.2, random_state=42
)

# 3. Entraîner le modèle
model_multi = LinearRegression()
model_multi.fit(X_train_multi, y_train_multi)

# 4. Prédictions
y_pred_multi = model_multi.predict(X_test_multi)

# 5. Métriques
mse_multi = mean_squared_error(y_test_multi, y_pred_multi)
rmse_multi = np.sqrt(mse_multi)
mae_multi = mean_absolute_error(y_test_multi, y_pred_multi)
r2_multi = r2_score(y_test_multi, y_pred_multi)

print(f"Régression Multiple:")
print(f"  MSE: {mse_multi:.4f}")
print(f"  RMSE: {rmse_multi:.4f}")
print(f"  MAE: {mae_multi:.4f}")
print(f"  R²: {r2_multi:.4f}")

# 6. Importance des variables
feature_names = df.drop('MedHouseVal', axis=1).columns
coefficients = pd.DataFrame({
    'Variable': feature_names,
    'Coefficient': model_multi.coef_
}).sort_values('Coefficient', key=abs, ascending=False)

print("\nImportance des variables (coefficients):")
print(coefficients)

# 7. Visualisation : Prédictions vs Valeurs réelles
plt.figure(figsize=(10, 6))
plt.scatter(y_test_multi, y_pred_multi, alpha=0.5)
plt.plot([y_test_multi.min(), y_test_multi.max()],
         [y_test_multi.min(), y_test_multi.max()],
         'r--', lw=2)
plt.xlabel('Valeurs réelles')
plt.ylabel('Valeurs prédites')
plt.title('Prédictions vs Valeurs réelles (Régression Multiple)')
plt.grid(True, alpha=0.3)
plt.show()

# 8. Analyse des résidus
residus = y_test_multi - y_pred_multi

fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Distribution des résidus
axes[0].hist(residus, bins=50, edgecolor='black')
axes[0].set_xlabel('Résidus')
axes[0].set_ylabel('Fréquence')
axes[0].set_title('Distribution des résidus')
axes[0].axvline(x=0, color='r', linestyle='--')

# Résidus vs Valeurs prédites
axes[1].scatter(y_pred_multi, residus, alpha=0.5)
axes[1].axhline(y=0, color='r', linestyle='--')
axes[1].set_xlabel('Valeurs prédites')
axes[1].set_ylabel('Résidus')
axes[1].set_title('Résidus vs Valeurs prédites')
axes[1].grid(True, alpha=0.3)

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

1.3 Exercice 1.3 : Descente de gradient

Implémentez une régression linéaire simple en utilisant la descente de gradient.

1.3 *Exercice 1.3 : Descente de gradient*

1. Implémenter la fonction de coût (MSE)
2. Implémenter la descente de gradient
3. Tester avec différents taux d'apprentissage (0.001, 0.01, 0.1)
4. Visualiser l'évolution du coût au fil des itérations
5. Comparer les résultats avec la régression linéaire de scikit-learn

Solution :

```
def compute_cost(X, y, theta):
    """Calcule la fonction de coût MSE"""
    m = len(y)
    predictions = X.dot(theta)
    cost = (1/(2*m)) * np.sum((predictions - y)**2)
    return cost

def gradient_descent(X, y, theta, learning_rate, iterations):
    """Effectue la descente de gradient"""
    m = len(y)
    cost_history = []

    for i in range(iterations):
        predictions = X.dot(theta)
        errors = predictions - y
        theta = theta - (learning_rate/m) * X.T.dot(errors)
        cost = compute_cost(X, y, theta)
        cost_history.append(cost)

    return theta, cost_history

# Préparer les données
X_simple = df[['MedInc']].values
y_simple = df['MedHouseVal'].values

# Normaliser les données
X_mean = X_simple.mean()
X_std = X_simple.std()
X_norm = (X_simple - X_mean) / X_std

# Ajouter une colonne de 1 pour l'intercept
X_norm = np.c_[np.ones(len(X_norm)), X_norm]

# Initialiser theta
theta = np.zeros(2)

# Tester différents taux d'apprentissage
learning_rates = [0.001, 0.01, 0.1]
iterations = 1000

plt.figure(figsize=(15, 5))

for idx, lr in enumerate(learning_rates, 1):
    theta_init = np.zeros(2)
    theta_final, cost_history = gradient_descent(
        X_norm, y_simple, theta_init, lr, iterations
    )

    plt.subplot(1, 3, idx)
    plt.plot(range(iterations), cost_history)
    plt.xlabel('Itérations')
    plt.ylabel('Coût (MSE)')
    plt.title(f'Learning rate = {lr}')
    plt.grid(True, alpha=0.3)

    print(f"\nLearning rate = {lr}")
    print(f"  Theta final: {theta_final}")
    print(f"  Coût final: {cost_history[-1]:.4f}")

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

```python
# Comparaison avec scikit-learn
model_sklearn = LinearRegression()
model_sklearn.fit(X_simple, y_simple)
print(f"\nScikit-learn:")
print(f"  Coefficient: {model_sklearn.coef_[0]:.4f}")
print(f"  Intercept: {model_sklearn.intercept_:.4f}")
```

2 Partie 2 : Régression Logistique

2.1 Exercice 2.1 : Classification binaire avec Iris

Utilisez le dataset Iris pour créer un classifieur binaire qui distingue deux espèces.

1. Charger le dataset Iris
2. Créer un problème de classification binaire (Setosa vs non-Setosa)
3. Séparer les données en ensemble d'entraînement (70%) et de test (30%)
4. Entraîner un modèle de régression logistique
5. Calculer les métriques : accuracy, précision, rappel, F1-score
6. Créer la matrice de confusion
7. Tracer la courbe ROC et calculer l'AUC

Solution :

```python
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, confusion_matrix, roc_curve, auc)
import seaborn as sns

# 1. Charger les données
iris = load_iris()
X = iris.data
y = iris.target

# 2. Créer un problème binaire : Setosa (0) vs non-Setosa (1)
y_binary = (y != 0).astype(int)

# 3. Séparer les données
X_train, X_test, y_train, y_test = train_test_split(
    X, y_binary, test_size=0.3, random_state=42, stratify=y_binary
)

# 4. Entraîner le modèle
log_reg = LogisticRegression(max_iter=200)
log_reg.fit(X_train, y_train)

# 5. Prédictions
y_pred = log_reg.predict(X_test)
y_pred_proba = log_reg.predict_proba(X_test)[:, 1]

# 6. Métriques
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print("Métriques de classification:")
print(f"  Accuracy: {accuracy:.4f}")
print(f"  Précision: {precision:.4f}")
print(f"  Rappel: {recall:.4f}")
print(f"  F1-Score: {f1:.4f}")

# 7. Matrice de confusion
cm = confusion_matrix(y_test, y_pred)

plt.figure(figsize=(12, 5))
```

```
# Matrice de confusion
plt.subplot(1, 2, 1)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Matrice de Confusion')
plt.ylabel('Valeur réelle')
plt.xlabel('Valeur prédite')

# 8. Courbe ROC
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
roc_auc = auc(fpr, tpr)

plt.subplot(1, 2, 2)
plt.plot(fpr, tpr, color='darkorange', lw=2,
         label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Taux de faux positifs')
plt.ylabel('Taux de vrais positifs')
plt.title('Courbe ROC')
plt.legend(loc="lower right")
plt.grid(True, alpha=0.3)

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

2.2 Exercice 2.2 : Classification multiclasse

Étendez la régression logistique pour classifier les trois espèces d'Iris.

1. Utiliser toutes les classes du dataset Iris
2. Entraîner un modèle de régression logistique multiclasse
3. Visualiser la matrice de confusion
4. Calculer le rapport de classification complet
5. Comparer les performances pour chaque classe

Solution :

```
from sklearn.metrics import classification_report

# 1. Utiliser toutes les classes
y_multi = iris.target

# 2. Séparer les données
X_train_m, X_test_m, y_train_m, y_test_m = train_test_split(
    X, y_multi, test_size=0.3, random_state=42, stratify=y_multi
)

# 3. Entraîner le modèle multiclasse
log_reg_multi = LogisticRegression(max_iter=200, multi_class='multinomial')
log_reg_multi.fit(X_train_m, y_train_m)

# 4. Prédictions
y_pred_multi = log_reg_multi.predict(X_test_m)

# 5. Métriques globales
accuracy_multi = accuracy_score(y_test_m, y_pred_multi)
print(f"Accuracy globale: {accuracy_multi:.4f}\n")

# 6. Rapport de classification détaillé
print("Rapport de classification:")
print(classification_report(y_test_m, y_pred_multi,
                            target_names=iris.target_names))

# 7. Matrice de confusion
cm_multi = confusion_matrix(y_test_m, y_pred_multi)
```

```
plt.figure(figsize=(8, 6))
sns.heatmap(cm_multi, annot=True, fmt='d', cmap='Blues',
            xticklabels=iris.target_names,
            yticklabels=iris.target_names)
plt.title('Matrice de Confusion - Classification Multiclasse')
plt.ylabel('Valeur réelle')
plt.xlabel('Valeur prédite')
plt.show()

# 8. Analyse par classe
print("\nAnalyse par classe:")
for i, name in enumerate(iris.target_names):
    class_pred = (y_pred_multi == i)
    class_true = (y_test_m == i)
    class_accuracy = accuracy_score(class_true, class_pred)
    print(f"  {name}: {class_accuracy:.4f}")
```

3 Partie 3 : Classifieurs

3.1 Exercice 3.1 : K-Nearest Neighbors (K-NN)

Implémentez et testez le classifieur K-NN sur le dataset Iris.

1. Tester différentes valeurs de K (1, 3, 5, 7, 9, 11)
2. Pour chaque K, calculer l'accuracy sur l'ensemble de test
3. Tracer la courbe de l'accuracy en fonction de K
4. Déterminer la valeur optimale de K
5. Analyser l'impact de la normalisation des données
6. Comparer avec et sans normalisation

Solution :

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler

# Préparer les données
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.3, random_state=42
)

# Tester différentes valeurs de K
k_values = [1, 3, 5, 7, 9, 11, 15, 19]
accuracies_without_scaling = []
accuracies_with_scaling = []

# Sans normalisation
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    accuracies_without_scaling.append(acc)

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

for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train_scaled, y_train)
    y_pred = knn.predict(X_test_scaled)
    acc = accuracy_score(y_test, y_pred)
    accuracies_with_scaling.append(acc)

# Visualisation
plt.figure(figsize=(10, 6))
```

```
plt.plot(k_values, accuracies_without_scaling, 'o-',
         label='Sans normalisation', linewidth=2)
plt.plot(k_values, accuracies_with_scaling, 's-',
         label='Avec normalisation', linewidth=2)
plt.xlabel('Valeur de K')
plt.ylabel('Accuracy')
plt.title('Performance du K-NN en fonction de K')
plt.legend()
plt.grid(True, alpha=0.3)
plt.xticks(k_values)
plt.show()

# Trouver le meilleur K
best_k_without = k_values[np.argmax(accuracies_without_scaling)]
best_k_with = k_values[np.argmax(accuracies_with_scaling)]

print(f"Meilleur K sans normalisation: {best_k_without} "
      f"(accuracy: {max(accuracies_without_scaling):.4f})")
print(f"Meilleur K avec normalisation: {best_k_with} "
      f"(accuracy: {max(accuracies_with_scaling):.4f})")
```

3.2 Exercice 3.2 : Arbres de Décision

Utilisez un arbre de décision pour classifier les données Iris.

1. Entraîner un arbre de décision avec profondeur maximale = 3
2. Visualiser l'arbre de décision
3. Tester différentes profondeurs (2, 3, 4, 5, 10)
4. Identifier le risque de sur-apprentissage
5. Utiliser la validation croisée pour choisir la meilleure profondeur

Solution :

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import cross_val_score

# Préparer les données
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.3, random_state=42
)

# 1. Entraîner un arbre simple
tree = DecisionTreeClassifier(max_depth=3, random_state=42)
tree.fit(X_train, y_train)

# 2. Visualiser l'arbre
plt.figure(figsize=(20, 10))
plot_tree(tree, feature_names=iris.feature_names,
          class_names=iris.target_names, filled=True, fontsize=10)
plt.title('Arbre de Décision (profondeur = 3)')
plt.show()

# 3. Tester différentes profondeurs
depths = range(1, 15)
train_accuracies = []
test_accuracies = []
cv_accuracies = []

for depth in depths:
    tree = DecisionTreeClassifier(max_depth=depth, random_state=42)
    tree.fit(X_train, y_train)

    # Accuracy sur train
    train_acc = tree.score(X_train, y_train)
    train_accuracies.append(train_acc)

    # Accuracy sur test
    test_acc = tree.score(X_test, y_test)
```

```
    test_accuracies.append(test_acc)

    # Validation croisée
    cv_scores = cross_val_score(tree, X_train, y_train, cv=5)
    cv_accuracies.append(cv_scores.mean())

# Visualisation
plt.figure(figsize=(12, 6))
plt.plot(depths, train_accuracies, 'o-', label='Train', linewidth=2)
plt.plot(depths, test_accuracies, 's-', label='Test', linewidth=2)
plt.plot(depths, cv_accuracies, '^-', label='Validation Croisée', linewidth=2)
plt.xlabel('Profondeur de l\'arbre')
plt.ylabel('Accuracy')
plt.title('Performance en fonction de la profondeur')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Identifier le sur-apprentissage
best_depth_cv = depths[np.argmax(cv_accuracies)]
print(f"\nMeilleure profondeur (validation croisée): {best_depth_cv}")
print(f"Accuracy CV: {max(cv_accuracies):.4f}")

# Analyser le sur-apprentissage
overfitting_gap = []
for i, depth in enumerate(depths):
    gap = train_accuracies[i] - test_accuracies[i]
    overfitting_gap.append(gap)
    if depth in [3, 5, 10]:
        print(f"\nProfondeur {depth}:")
        print(f"  Train accuracy: {train_accuracies[i]:.4f}")
        print(f"  Test accuracy: {test_accuracies[i]:.4f}")
        print(f"  Écart (sur-apprentissage): {gap:.4f}")
```

3.3 Exercice 3.3 : Support Vector Machine (SVM)

Appliquez le SVM pour la classification des données Iris.

1. Tester un SVM linéaire
2. Tester un SVM avec noyau RBF
3. Tester un SVM avec noyau polynomial
4. Comparer les performances des trois variantes
5. Optimiser le paramètre C (régularisation) pour le SVM RBF
6. Visualiser la frontière de décision (2D, en utilisant 2 features)

Solution :

```
from sklearn.svm import SVC

# Préparer les données avec normalisation (important pour SVM)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 1. Tester différents noyaux
kernels = ['linear', 'rbf', 'poly']
results = {}

for kernel in kernels:
    svm = SVC(kernel=kernel, random_state=42)
    svm.fit(X_train_scaled, y_train)

    train_acc = svm.score(X_train_scaled, y_train)
    test_acc = svm.score(X_test_scaled, y_test)

    results[kernel] = {
        'train_accuracy': train_acc,
        'test_accuracy': test_acc
```

```
    }

    print(f"\nSVM avec noyau {kernel}:")
    print(f"  Train accuracy: {train_acc:.4f}")
    print(f"  Test accuracy: {test_acc:.4f}")

# 2. Optimiser le paramètre C pour RBF
C_values = [0.01, 0.1, 1, 10, 100]
accuracies = []

for C in C_values:
    svm = SVC(kernel='rbf', C=C, random_state=42)
    scores = cross_val_score(svm, X_train_scaled, y_train, cv=5)
    accuracies.append(scores.mean())

plt.figure(figsize=(10, 6))
plt.plot(C_values, accuracies, 'o-', linewidth=2)
plt.xscale('log')
plt.xlabel('Paramètre C')
plt.ylabel('Accuracy (Validation Croisée)')
plt.title('Optimisation du paramètre C pour SVM RBF')
plt.grid(True, alpha=0.3)
plt.show()

best_C = C_values[np.argmax(accuracies)]
print(f"\nMeilleur C: {best_C}")
print(f"Accuracy CV: {max(accuracies):.4f}")

# 3. Visualiser la frontière de décision (2D)
# Utiliser seulement 2 features pour la visualisation
X_2d = iris.data[:, :2]  # Prendre les 2 premières features
y_2d = iris.target

X_train_2d, X_test_2d, y_train_2d, y_test_2d = train_test_split(
    X_2d, y_2d, test_size=0.3, random_state=42
)

# Normaliser
scaler_2d = StandardScaler()
X_train_2d_scaled = scaler_2d.fit_transform(X_train_2d)
X_test_2d_scaled = scaler_2d.transform(X_test_2d)

# Entraîner SVM
svm_2d = SVC(kernel='rbf', C=best_C, random_state=42)
svm_2d.fit(X_train_2d_scaled, y_train_2d)

# Créer une grille pour la visualisation
h = 0.02  # pas de la grille
x_min, x_max = X_train_2d_scaled[:, 0].min() - 1, X_train_2d_scaled[:, 0].max() + 1
y_min, y_max = X_train_2d_scaled[:, 1].min() - 1, X_train_2d_scaled[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

# Prédire sur la grille
Z = svm_2d.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Visualiser
plt.figure(figsize=(10, 8))
plt.contourf(xx, yy, Z, alpha=0.3, cmap='viridis')
plt.scatter(X_train_2d_scaled[:, 0], X_train_2d_scaled[:, 1],
            c=y_train_2d, cmap='viridis', edgecolors='black', s=50)
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title('Frontière de décision SVM (RBF)')
plt.colorbar()
plt.show()
```

3.4 Exercice 3.4 : Comparaison des classifieurs

Comparez tous les classifieurs étudiés sur le dataset Iris.

1. Créer une fonction pour évaluer chaque classifieur

3.4 Exercice 3.4 : Comparaison des classifieurs

2. Tester : Régression Logistique, K-NN, Arbre de Décision, SVM
3. Utiliser la validation croisée (5-fold) pour chaque modèle
4. Créer un tableau comparatif avec les métriques
5. Visualiser les résultats avec un graphique en barres
6. Analyser les temps d'exécution de chaque algorithme

Solution :

```
from sklearn.ensemble import RandomForestClassifier
import time

# Préparer les données
X_scaled = StandardScaler().fit_transform(iris.data)
y = iris.target

# Définir les classifieurs
classifiers = {
    'Régression Logistique': LogisticRegression(max_iter=200),
    'K-NN (k=5)': KNeighborsClassifier(n_neighbors=5),
    'Arbre de Décision': DecisionTreeClassifier(max_depth=5, random_state=42),
    'SVM (RBF)': SVC(kernel='rbf', C=1.0, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42)
}

# Évaluer chaque classifieur
results = []

for name, clf in classifiers.items():
    # Validation croisée
    start_time = time.time()
    cv_scores = cross_val_score(clf, X_scaled, y, cv=5)
    elapsed_time = time.time() - start_time

    # Entraîner sur toutes les données pour obtenir d'autres métriques
    X_train, X_test, y_train, y_test = train_test_split(
        X_scaled, y, test_size=0.3, random_state=42
    )
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)

    results.append({
        'Classifieur': name,
        'Accuracy (CV)': cv_scores.mean(),
        'Std (CV)': cv_scores.std(),
        'Accuracy (Test)': accuracy_score(y_test, y_pred),
        'Précision': precision_score(y_test, y_pred, average='macro'),
        'Rappel': recall_score(y_test, y_pred, average='macro'),
        'F1-Score': f1_score(y_test, y_pred, average='macro'),
        'Temps (s)': elapsed_time
    })

# Créer un DataFrame pour l'affichage
results_df = pd.DataFrame(results)
print("Comparaison des classifieurs:")
print(results_df.to_string(index=False))

# Visualisations
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# 1. Accuracy
axes[0, 0].bar(results_df['Classifieur'], results_df['Accuracy (CV)'])
axes[0, 0].set_ylabel('Accuracy')
axes[0, 0].set_title('Accuracy (Validation Croisée)')
axes[0, 0].tick_params(axis='x', rotation=45)
axes[0, 0].grid(axis='y', alpha=0.3)

# 2. F1-Score
axes[0, 1].bar(results_df['Classifieur'], results_df['F1-Score'], color='orange')
axes[0, 1].set_ylabel('F1-Score')
```

```
axes[0, 1].set_title('F1-Score')
axes[0, 1].tick_params(axis='x', rotation=45)
axes[0, 1].grid(axis='y', alpha=0.3)

# 3. Temps d'exécution
axes[1, 0].bar(results_df['Classifieur'], results_df['Temps (s)'], color='green')
axes[1, 0].set_ylabel('Temps (secondes)')
axes[1, 0].set_title('Temps d\'exécution')
axes[1, 0].tick_params(axis='x', rotation=45)
axes[1, 0].grid(axis='y', alpha=0.3)

# 4. Comparaison Précision vs Rappel
axes[1, 1].scatter(results_df['Précision'], results_df['Rappel'], s=100)
for i, txt in enumerate(results_df['Classifieur']):
    axes[1, 1].annotate(txt, (results_df['Précision'].iloc[i],
                              results_df['Rappel'].iloc[i]),
                        fontsize=8, ha='right')
axes[1, 1].set_xlabel('Précision')
axes[1, 1].set_ylabel('Rappel')
axes[1, 1].set_title('Précision vs Rappel')
axes[1, 1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Identifier le meilleur classifieur
best_classifier = results_df.loc[results_df['Accuracy (CV)'].idxmax(), 'Classifieur']
best_accuracy = results_df['Accuracy (CV)'].max()
print(f"\n{'='*50}")
print(f"Meilleur classifieur: {best_classifier}")
print(f"Accuracy: {best_accuracy:.4f}")
print(f"{'='*50}")
```

4 Partie 4 : Étude de Cas Complète

4.1 Exercice 4.1 : Dataset Titanic

Appliquez l'ensemble des techniques apprises sur le dataset Titanic.

1. Télécharger le dataset Titanic depuis Kaggle ou scikit-learn
2. Effectuer l'analyse exploratoire des données (EDA)
3. Nettoyer les données (valeurs manquantes, encodage)
4. Créer de nouvelles features pertinentes (feature engineering)
5. Tester au moins 5 algorithmes différents
6. Utiliser la validation croisée et l'optimisation des hyperparamètres
7. Présenter un rapport complet avec visualisations

Commentaire :
Cet exercice est un projet complet qui nécessite plusieurs heures de travail. Il permet d'appliquer toutes les compétences acquises dans ce TP. Utilisez les notebooks Jupyter pour documenter votre démarche et vos résultats.

5 Questions de Réflexion

5.1 Questions théoriques

1. Expliquez la différence fondamentale entre régression et classification.
2. Quand doit-on utiliser la régression logistique plutôt qu'une régression linéaire ?
3. Quels sont les avantages et inconvénients du K-NN par rapport aux arbres de décision ?
4. Expliquez le phénomène de sur-apprentissage (overfitting) et comment le détecter.
5. Pourquoi est-il important de normaliser les données pour certains algorithmes (SVM, K-NN) mais pas pour d'autres (arbres de décision) ?

6. Quelle est la différence entre la validation croisée et un simple split train/test ?
7. Expliquez les métriques : précision, rappel, F1-score. Quand privilégier l'une plutôt que l'autre ?
8. Comment choisir la valeur de K dans l'algorithme K-NN ?

Solution :
Réponses aux questions théoriques :

1. **Régression vs Classification :**
   — Régression : prédit une valeur continue (ex : prix, température)
   — Classification : prédit une catégorie discrète (ex : spam/non-spam, espèce)
2. **Régression Logistique :** Utiliser la régression logistique quand :
   — La variable cible est catégorielle (binaire ou multiclasse)
   — On veut estimer des probabilités de classe
   — On a besoin d'un modèle interprétable
3. **K-NN vs Arbres de Décision :**
   K-NN avantages : simple, non-paramétrique, pas d'entraînement
   K-NN inconvénients : lent en prédiction, sensible aux features non pertinentes, nécessite normalisation
   Arbres avantages : rapides, interprétables, gèrent les features non-linéaires
   Arbres inconvénients : tendance au sur-apprentissage, instables
4. **Sur-apprentissage :** Le modèle apprend trop bien les données d'entraînement, incluant le bruit. Détection : écart important entre performance train et test.
5. **Normalisation :**
   — SVM et K-NN : utilisent des distances → sensibles à l'échelle
   — Arbres : utilisent des seuils → insensibles à l'échelle
6. **Validation croisée :**
   — Train/test : une seule division, peut être biaisée
   — Validation croisée : multiple divisions, estimation plus robuste
7. **Métriques :**
   — Précision : parmi les prédictions positives, combien sont correctes
   — Rappel : parmi les vrais positifs, combien sont détectés
   — F1 : moyenne harmonique, équilibre précision/rappel
   — Privilégier le rappel pour détecter les maladies (faux négatif grave)
   — Privilégier la précision pour les filtres anti-spam (faux positif gênant)
8. **Choix de K :**
   — K trop petit : sensible au bruit, sur-apprentissage
   — K trop grand : sous-apprentissage, frontières trop lisses
   — Méthode : tester plusieurs valeurs avec validation croisée
   — Règle empirique : $K \approx \sqrt{n}$ où n est le nombre d'exemples

6 Pour aller plus loin

6.1 Exercices avancés

1. **Régression Polynomiale :** Implémenter une régression polynomiale et identifier le degré optimal pour éviter le sur-apprentissage.
2. **Régularisation :** Comparer Ridge, Lasso et ElasticNet sur le dataset California Housing.
3. **Ensemble Methods :** Implémenter et comparer Random Forest, AdaBoost et Gradient Boosting.
4. **Optimisation des hyperparamètres :** Utiliser GridSearchCV et RandomizedSearchCV pour optimiser les hyperparamètres du SVM.
5. **Pipeline :** Créer un pipeline complet incluant preprocessing, feature selection et classification.
6. **Courbes d'apprentissage :** Tracer et analyser les courbes d'apprentissage pour diagnostiquer les problèmes de biais/variance.

6.2 Ressources complémentaires

— **Livre :** "Hands-On Machine Learning with Scikit-Learn" - Aurélien Géron
— **Documentation :** https://scikit-learn.org/stable/
— **Cours :** Andrew Ng - Machine Learning (Coursera)
— **Datasets :** UCI Machine Learning Repository, Kaggle