TP2

November 18, 2024

Exercice 1 : SVM à marge douce

0.0.1 Objectifs

- Comprendre le rôle des variables de slack $ $ dans les SVM.
- Manipuler le paramètre (C) pour ajuster la tolérance aux erreurs de classification.
- Appliquer un SVM à un jeu de données non linéairement séparable et analyser les résultats.

0.1 Données synthétiques

On s'intéresse ici au cas de données presque linéairement séparable. Et on cherche à résoudre le problème du SVM correspondant

[ ]:
```
import matplotlib
```

[ ]:
```
def generate_nearly_separable_data(num_points_per_class=8, seed=None,␣
↪noise_level=0.3):
    """
    Génère des données presque linéairement séparables en ajoutant du bruit␣
↪autour de la frontière de décision.

    Parameters:
    - num_points_per_class: int, nombre de points par classe.
    - seed: int, valeur pour initialiser le générateur aléatoire (pour␣
↪reproductibilité).
    - noise_level: float, niveau de bruit ajouté pour rendre les données␣
↪presque linéairement séparables.

    Returns:
    - X: numpy.ndarray, coordonnées des points (2D).
    - y: numpy.ndarray, classes des points (+1 ou -1).
    """
    if seed is not None:
        np.random.seed(seed)

    # Génération des points pour la classe +1 (séparée)
```

```
    class1 = np.random.multivariate_normal(mean=[-2, -2], cov=[[1, 0.5], [0.5,␣
↪1]], size=num_points_per_class)

    # Génération des points pour la classe -1 (séparée)
    class2 = np.random.multivariate_normal(mean=[2, 2], cov=[[1, -0.3], [-0.3,␣
↪1]], size=num_points_per_class)

    # Combinaison des deux classes
    X = np.vstack((class1, class2))
    y = np.hstack((np.ones(num_points_per_class), -np.
↪ones(num_points_per_class)))

    # Ajout de bruit pour rendre les données presque linéairement séparables
    noise = np.random.randn(X.shape[0], X.shape[1]) * noise_level
    X += noise

    return X, y




def plot_points(X, y):
    """
    Affiche les points de données avec différentes couleurs pour chaque classe.

    Parameters:
    - X: numpy.ndarray, les données avec les caractéristiques (n_samples,␣
↪n_features).
    - y: numpy.ndarray, les étiquettes des classes associées aux données␣
↪(n_samples).
    """
    unique_classes = np.unique(y)  # Trouve toutes les classes uniques


    # Tracer les points de chaque classe
    for i, label in enumerate(unique_classes):
        plt.scatter(X[y == label][:, 0], X[y == label][:, 1], label=f'Classe␣
↪{label}')

    # Ajouter une légende et un titre
    plt.legend()
    plt.title("Données avec classes multiples")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.grid(True)
    #plt.show()
```

```python
def plot_decision_boundary_with_margins(w, b, X,y, X_range=(-5, 5),␣
↪Y_range=(-5, 5), title="Frontière de décision et marges SVM"):
    """
    Affiche la frontière de décision et les marges d'un SVM linéaire.

    Parameters:
    - w: numpy.ndarray, vecteur des poids appris.
    - b: float, biais appris.
    - X_range: tuple, plage des valeurs pour l'axe x (min, max).
    - Y_range: tuple, plage des valeurs pour l'axe y (min, max).
    - title: str, titre du graphique.
    """
    plt.figure(figsize=(8, 6))
    plot_points(X, y)

    # Plage de valeurs x pour afficher la frontière et les marges
    x1 = np.linspace(X_range[0], X_range[1], 100)

    # Calcul de x2 pour les trois lignes : frontière, marge positive et marge␣
↪négative
    decision_boundary = -(w[0] * x1 + b) / w[1]  # Frontière de décision
    margin_positive = -(w[0] * x1 + b - 1) / w[1]  # Marge positive
    margin_negative = -(w[0] * x1 + b + 1) / w[1]  # Marge négative

    # Tracer les trois lignes
    plt.plot(x1, decision_boundary, color='green', label='Frontière de␣
↪décision')
    plt.plot(x1, margin_positive, '--', color='m', label='Marge positive')
    plt.plot(x1, margin_negative, '--', color='m', label='Marge négative')

    # Définir les limites du graphique
    plt.xlim(X_range)
    plt.ylim(Y_range)

    # Ajouter des labels et une légende
    plt.title(title)
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Question: Test de la fonction

[ ]:
```
X, y = generate_nearly_separable_data(num_points_per_class=16, seed=42,␣
 ↪noise_level=1.5)
plot_points(X,y)
```

Question : En vous inspirant de la première partie, complèter la fonction suivant

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
```

[ ]:
```
def solve_svm2(X, y, C=1.0):
    n, f = X.shape
    w = cp.Variable(f)
    b = cp.Variable()
    slack = cp.Variable(n)

    objective = cp.Minimize(# a completer)
    constraints = []
    for i in range(n):
        constraint = # a completer  # Contraintes de marge
        constraints.append(constraint)
        constraints.append(# a completer)  # Les slack doivent être >= 0

    problem = cp.Problem(objective, constraints)
    problem.solve()

    return w.value, b.value
```

Question : Fonction pour tester différentes valeurs de C

[ ]:
```
def test_svm_with_different_C(X, y,C_values=[0.01, 0.1, 3, 10, 100.0]):
    for C in C_values:
        print(f"Résolution pour C = {C}")

        # Résolution du problème SVM
        w, b = solve_svm2(X, y, C)


        # Affichage de la frontière de décision et des marges
        plot_decision_boundary_with_margins(w, b, X,y, X_range=(-5, 5),␣
 ↪Y_range=(-5, 5), title="Frontière de décision et marges SVM")

# Génération de données de test

X, y = generate_nearly_separable_data(num_points_per_class=16, seed=42,␣
 ↪noise_level=1.5)
```

```
# Tester avec différentes valeurs de C
test_svm_with_different_C(X, y, C_values)
```

Question : Que se passe t-il quand $C = 0.001$?

[ ]:

Question : Que se passe t-il quand $C = 100$?

0.2 Données réelles

[ ]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
```

0.3 Données iris

Le jeu de données Iris est un ensemble classique en apprentissage automatique introduit par Ronald A. Fisher en 1936. Il comprend 150 échantillons de fleurs répartis en trois espèces (Iris setosa, Iris versicolor, Iris virginica), avec quatre caractéristiques : longueur et largeur du sépale, longueur et largeur du pétale (en cm). Utilisé principalement pour des tâches de classification, ce jeu de données est apprécié pour sa simplicité, sa taille réduite, et la diversité des classes, certaines étant facilement séparables (Iris setosa), tandis que d'autres (Iris versicolor et Iris virginica) sont plus proches.

Questions 1. Charger le dataset Iris 2. Garder les deux premières caractéristiques pour la visualisation 3. Filtrer pour conserver seulement deux classes (e.g., setosa et versicolor) 4. Séparer les données en train et test 5. Visusaliser les données

[ ]:
```
# Charger le dataset Iris
iris = load_iris()
X_iris = iris.data[:, :2]  # Garder les deux premières caractéristiques pour la␣
 ↪visualisation
y_iris = iris.target

# Filtrer pour conserver seulement deux classes (e.g., setosa et versicolor)
mask = y_iris < 2
X_iris= X_iris[mask]
y_iris = y_iris [mask]

# Séparer les données en train et test
X_train, X_test, y_train, y_test = train_test_split(X_iris, y_iris ,␣
 ↪test_size=0.3, random_state=42)
```

```
# Visualisation des données
plt.scatter(X_iris[:, 0], X_iris[:, 1], c=y_iris , cmap='viridis',␣
 ↪edgecolors='k')
plt.xlabel('Longueur des sépales')
plt.ylabel('Largeur des sépales')
plt.title('Dataset Iris : Deux premières caractéristiques')
plt.show()
```

0.4 Utilisation de l'algorithme de sklearn sur les données Iris

[ ]:
```
from sklearn.metrics import accuracy_score
```

On utilisera la fonction ci-dessous pour afficher la frontière de décision ainsi que les marges

[ ]:
```
def plot_decision_boundary_iris(model, X, y, title="Frontière de décision et␣
 ↪marges SVM"):
    """
    Trace la frontière de décision et les marges pour un SVM linéaire.

    Parameters:
    - model: SVM entraîné (scikit-learn SVC avec kernel='linear').
    - X: numpy.ndarray, données d'entrée (n_samples, n_features).
    - y: numpy.ndarray, étiquettes des classes (n_samples,).
    - title: str, titre du graphique.
    """
    plt.figure(figsize=(8, 6))

    # Tracer les points des deux classes
    unique_classes = np.unique(y)
    colors = ['r', 'b']
    for i, cls in enumerate(unique_classes):
        plt.scatter(
            X[y == cls, 0], X[y == cls, 1],
            label=f"Classe {cls}", color=colors[i], edgecolor='k'
        )

    # Récupérer les limites des axes
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    # Générer un maillage pour calculer la frontière
    xx, yy = np.meshgrid(
        np.linspace(x_min, x_max, 100),
        np.linspace(y_min, y_max, 100)
    )
    Z = model.decision_function(np.c_[xx.ravel(), yy.ravel()])
```

```
    Z = Z.reshape(xx.shape)

    # Tracer la frontière de décision et les marges
    plt.contour(xx, yy, Z, levels=[-1, 0, 1], linestyles=['--', '-', '--'],␣
↪colors='k')
    plt.title(title)
    plt.xlabel('Caractéristique 1')
    plt.ylabel('Caractéristique 2')
    plt.legend()
    plt.grid(True)
    plt.show()
```

La fonction `accuracy_score` de `sklearn.metrics` calcule l'exactitude (*accuracy*) d'un modèle de classification.

L'exactitude est définie comme le rapport entre le nombre de prédictions correctes et le nombre total de prédictions effectuées. Elle est donnée par la formule suivante :

$$\text{Accuracy} = \frac{\text{Nombre de prédictions correctes}}{\text{Nombre total d'échantillons}}$$

1. Entraîner un SVM avec un noyau linéaire

[ ]:

2. Tracer la frontière de décision et les marges

[ ]:

3. Faire la prédiction sur les donées de test

[ ]:

4. Calculer et afficher le score d'exactitude

[ ]:

1 SVM non linéairement séparable

Nous allosn reprendre la classification sur les données iris mais avec un noyau Gaussien

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
from sklearn.model_selection import train_test_split

# Chargement des données
iris = datasets.load_iris()
```

```
X, y = iris.data, iris.target
# On conserve 50% du jeu de données pour l'évaluation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
```

Testez l'effet du paramètre d'échelle du noyau (gamma) et du paramètre de régularisation C.

[ ]:
```
clf = svm.SVC(C=1.0, kernel='rbf', gamma=0.25)
clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

afficher la frontière de décision

[ ]:
```
X, y = iris.data[:,:2], iris.target
# On conserve 50% du jeu de données pour l'évaluation
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)
```

[ ]:
```
clf = svm.SVC(C=0.1, kernel='rbf', gamma=0.5)
clf.fit(X_train, y_train)

# Pour afficher la surface de décision on va discrétiser l'espace avec un pas h
h = .02
# Créer la surface de décision discretisée
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))

# Surface de décision
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, cmap=plt.cm.coolwarm, alpha=0.8)
# Afficher aussi les points d'apprentissage
plt.scatter(X_train[:, 0], X_train[:, 1], label="train", edgecolors='k',␣
 ↪c=y_train, cmap=plt.cm.coolwarm)
plt.scatter(X_test[:, 0], X_test[:, 1], label="test", marker='*', c=y_test,␣
 ↪cmap=plt.cm.coolwarm)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.title("SVM RBF")
```

Que constatez vous ?

L'utilisation du noyau gaussien permet d'obtenir des frontières de décision non linéaires. Le paramètre gamma correspond au rayon d'influence de chaque observation : plus est élevé, plus le rayon d'influence de chaque observation est réduit. Les observations plus proches de la frontière ont donc plus de poids et la frontière aura tendance à « coller » aux observations.

2 TP à rendre

Reproduisez pour les datasets suivants… - Digits (en utilisant les données complètes)

- Iris

… les expérimentations suivantes

- Mise au point de deux types des classifieurs : Arbre de décision et SVM. Pour chacun de ces types de classifieurs vous devrez :
- Définir les hyper-paramètres à faire varier.
- Evaluer et selectionner par Grid-Search l'ensemble des configurations possibles, en utilisant la Validation Croisée à 3 plis pour l'évaluation de la performance en généralisation. Vous pourrez vous inspirer d'un code tel que celui-ci pour boucler sur les datasets et/ou les classifieurs.
- Ecrire sous forme d'un tableau récapitulatif les performances respectives (les meilleures obtenues) par chacun des modèles sur chacun des jeux de données (sur le test set).
- Donner des conclusions sur les résultats obtenus quand à la performance, la stabilité, la robustesse des familles de classifieurs utilisées, et les paramètres optimaux de chaque type de modèle.

[ ]: