TP : Support Vector Machine

BEX Roméo

24 Septembre 2024

Contents

Introduction **1**

Question 1 **2**

Question 2 **2**

Question 3 (bonus) **3**

Question 4 **4**

Question 5 **6**

Question 6 **6**

Introduction

Dans ce rapport, nous présentons une étude sur l'application des **SVM** (Machines à Vecteurs de Support) pour la classification de données. Nous abordons plusieurs aspects, tels que l'impact des noyaux linéaires et polynomiaux, l'ajout de variables de nuisance, et la réduction de dimension avec PCA.

Mathématiquement, le modèle SVM optimise la fonction suivante :

$$\underset{\mathbf{w},b}{\operatorname{argmin}} \left(\frac{1}{2} \|\mathbf{w}\|^2 \right) + C \sum_{i=1}^{n} \max(0, 1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b))$$

Le paramètre **C** contrôle le compromis entre la maximisation de la marge et la réduction des erreurs de classification sur les données d'entraînement.

Les algorithmes sont testés sur des jeux de données tels que **Iris** et **Labeled Faces in the Wild (LFW)**.

Question 1

Nous appliquons un SVM à noyau linéaire sur l'ensemble de données Iris avec une recherche par grille pour optimiser le paramètre de régularisation C.

```
# Définir les paramètres pour la recherche de grille
parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3, 200))}

# Utiliser GridSearchCV pour optimiser le modèle SVM avec noyau linéaire
clf_linear = GridSearchCV(SVC(), parameters, n_jobs=1)
clf_linear.fit(X_train, y_train)

# Calculer les scores de généralisation et les afficher
train_score = clf_linear.score(X_train, y_train)
test_score = clf_linear.score(X_test, y_test)
print(f'Generalization score for linear kernel: {train_score}, {test_score}')
```

Résultats :

- Précision sur l'ensemble de test : 0.62
- Précision sur l'ensemble d'entraînement : 0.72

Question 2

Nous appliquons cette fois un SVM à noyau polynomial.

```
# Définir les paramètres pour le noyau polynomial
Cs = list(np.logspace(-3, 3, 5))
gammas = 10. ** np.arange(1, 2)
degrees = np.r_[1, 2, 3]

parameters = {'kernel': ['poly'], 'C': Cs, 'gamma': gammas, 'degree': degrees}
```

```python
# Utiliser GridSearchCV pour optimiser le modèle SVM avec noyau polynomial
clf_poly = GridSearchCV(SVC(), parameters, cv=5)
clf_poly.fit(X_train, y_train)

# Afficher les meilleurs paramètres
print(clf_poly.best_params_)
```

Résultats :

- C : 0.031
- Degré (polynôme) : 1
- Gamma : 10.0

Scores de généralisation :

- Données d'entraînement : 0.71
- Données tests : 0.68

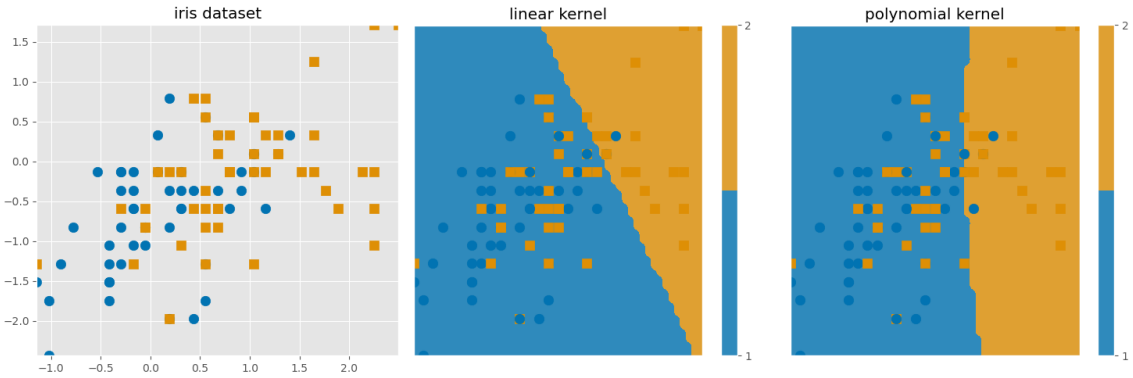

Figure 1: Frontière de décision des noyaux linéaire et polynomial

Après avoir exécuté plusieurs fois le code, on constate que la précision reste sensiblement la même pour chaque méthode. Ajouter de la complexité au modèle ne semble pas toujours améliorer les résultats.

Question 3 (bonus)

En utilisant le script `svm_gui.py` et en faisant varier le paramètre C, on obtient les graphiques suivants :

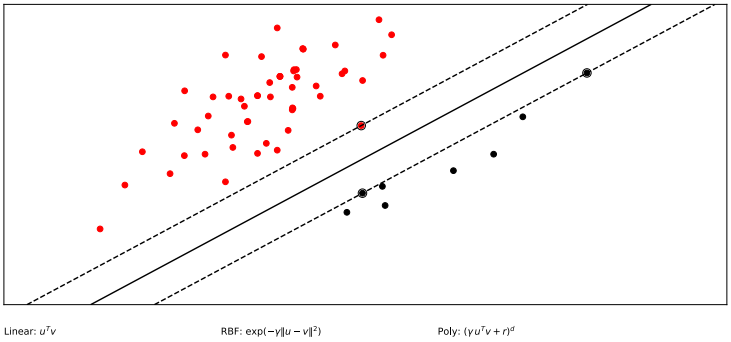

Figure 2: Frontière de décision avec $C = 1$

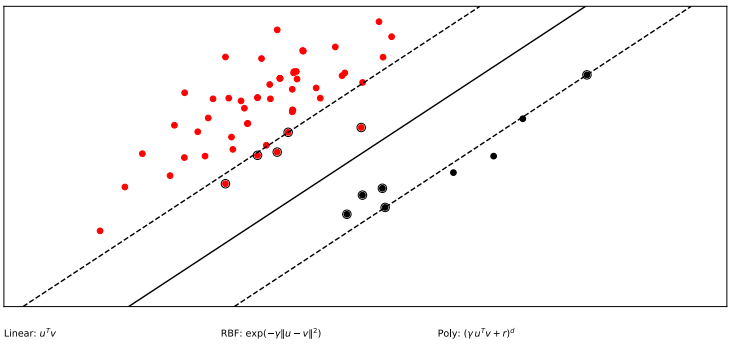

Figure 3: Frontière de décision avec $C = 0.001$

Le paramètre C contrôle le compromis entre une marge large et la minimisation des erreurs de classification. Quand C est faible, la marge est plus large mais il peut y avoir plus d'erreurs de classification. Quand C est plus élevé, la frontière est plus stricte autour des points d'entraînement, ce qui peut entraîner du surapprentissage.

Question 4

Nous allons optimiser le paramètre C avec SVM linéaire sur le jeu de données LFW.

Figure 4: Extrait du jeu de données Labeled Faces in the Wild (LFW)

```
Cs = 10. ** np.arange(-5, 6)
scores = []
for C in Cs:
    clf = SVC(kernel='linear', C=C)
    clf.fit(X_train, y_train)
    scores.append(clf.score(X_train, y_train))

# Meilleur C et graphique des scores
best_C = Cs[np.argmax(scores)]
plt.plot(Cs, scores)
plt.xscale('log')
plt.title(f"Optimisation du paramètre C")
plt.show()

print(f"Meilleur paramètre C : {best_C}")
print(f"Meilleur score: {np.max(scores)}")
```

Le modèle se comporte de manière optimale avec un C autour de 10^{-3}, où il atteint le meilleur score. Pour des valeurs de C trop petites, le modèle sous-apprend, tandis que des valeurs très élevées n'apportent pas d'amélioration notable.

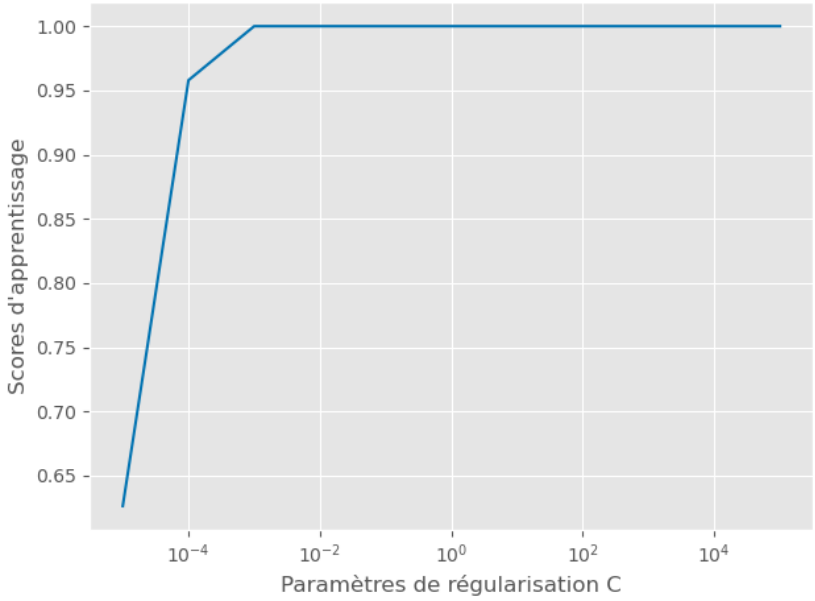

Figure 5: Graphique des scores d'apprentissage en fonction de C

Question 5

Nous avons ajouté des variables de nuisances (250) aux données pour étudier l'impact du bruit.

```
# Ajout de 250 variables de nuisance
noise = sigma * np.random.randn(n_samples, 250)
X_noisy = np.concatenate((X, noise), axis=1)
X_noisy = X_noisy[np.random.permutation(X.shape[0])]

# Validation croisée avec bruit
run_svm_cv(X_noisy, y)
```

Sans bruit, le modèle SVM obtenait une accuracy d'environ 95%. Après ajout du bruit, l'accuracy est tombée à 52%, ce qui montre l'importance d'éliminer les variables non pertinentes.

Question 6

En réduisant la dimensionnalité des données avec l'Analyse en Composantes Principales (PCA), nous avons pu observer une amélioration du score. Le

tableau suivant montre les résultats en fonction du nombre de composantes principales conservées :

Nombre de composantes	Score
3	0.5519
10	0.6388
20	0.6172

Table 1: Résultats en fonction du nombre de composantes principales

```
n_components = 20  # jouer avec ce parametre
print(f'Score après réduction de dimension avec {n_components} composantes principales')
pca = PCA(n_components=n_components).fit(X_noisy)
X_pca = pca.transform(X_noisy)
run_svm_cv(X_pca,y)
```

Nous constatons une amélioration du score de classification en prenant en compte un certain nombre de composantes principales (comme 10 dans cet exemple). Toutefois, si trop de composantes sont conservées, le score commence à diminuer.