Université de montpellier
Faculté des sciences

Département mathématique

MIND-SIAD

TP n°3 : Support Vector Machine (SVM)

ABCHICHE Thiziri
01/10/2024

Table des matières

1 Introduction

Les machines à vecteurs de support (SVM), introduites par Vapnik, sont des méthodes populaires en classification binaire grâce à leur capacité à séparer des classes à l'aide d'hyperplans dans des espaces de grande dimension. L'objectif de ce TP est d'appliquer les SVM sur des données réelles et simulées, tout en explorant l'impact des hyper-paramètres et des noyaux au sein du package scikit-learn.

Le TP se divise en deux principales parties : la classification sur le jeu de données Iris et la classification de visages à partir de la base "Labeled Faces in the Wild". Dans la première partie, nous nous concentrons sur la classification des données Iris en utilisantdes SVM avec des noyaux linéaires et polynomiaux. La seconde partie porte sur la classification de visages, où nous analysons l'influence du paramètre de régularisation, l'impact des variables de nuisance, et l'amélioration des performances par la réduction de dimension via l'Analyse en Composantes Principales (PCA).

2 Définitions et Notations

Les machines à vecteurs de support (SVM) sont des algorithmes de classification binaire qui reposent sur la recherche de règles de décision linéaires, appelées hyperplans, pour séparer les classes de données.

2.1 Notations

Dans le cadre de la classification binaire supervisée, nous utilisons les notations suivantes :

— Y : l'ensemble des étiquettes (ou labels), généralement défini comme $Y = \{-1, 1\}$ pour la classification binaire.

— $x = (x_1, \ldots, x_p) \in X \subset \mathbb{R}^p$: une observation (ou exemple) dans un espace de caractéristiques.

— $D_n = \{(x_i, y_i), i = 1, \ldots, n\}$: un ensemble d'apprentissage contenant n exemples et leurs étiquettes associées.

— Il existe un modèle probabiliste qui gouverne la génération de nos observations selon des variables aléatoires X et Y : pour tout $i \in \{1, \ldots, n\}$, $(x_i, y_i) \overset{\text{i.i.d}}{\sim} (X, Y)$.

2.2 Fonction de Décision

L'objectif est de construire, à partir de l'ensemble d'apprentissage D_n, une fonction $\hat{f} : X \to \{-1, 1\}$ qui prédit l'étiquette pour un point inconnu x (non présent dans l'ensemble d'apprentissage) : $\hat{f}(x)$. La règle de décision est dite linéaire, dans le sens où elle sépare l'espace par un hyperplan affine.

2.3 SVM Non Linéaires

Les SVM non linéaires utilisent une fonction implicite Φ qui transforme l'espace d'entrée $X \subset \mathbb{R}^p$ en un espace hilbertien $(H, \langle \cdot, \cdot \rangle)$ de dimension supérieure. L'apprentissage s'effectue alors dans cet espace, où l'on espère que les données soient plus linéairement séparables.

2.4 Noyaux

La sélection d'un noyau approprié est cruciale pour la performance du modèle. Parmi les noyaux couramment utilisés, on trouve :

— Noyau linéaire : $K(x, x') = \langle x, x' \rangle$.

— Noyau Gaussien radial (RBF) : $K(x, x') = \exp(-\gamma \|x - x'\|^2)$.

— Noyaux polynomiaux : $K(x, x') = (\alpha + \beta \langle x, x' \rangle)^\delta$ pour $\delta > 0$.

2.5 Formulation du Classificateur SVM

Un classificateur SVM est généralement formulé comme suit :

$$\hat{f}_{w,w_0}(x) = \text{sign}(\langle w, \Phi(x) \rangle + w_0),$$

où $w \in H$ et $w_0 \in \mathbb{R}$ sont des paramètres ajustés lors de l'apprentissage.

2.6 Optimisation

La maximisation de la marge séparant les classes revient à résoudre un problème d'optimisation sous contraintes, formulé comme suit :

$$(w^*, w_0^*, \xi^* \in \mathbb{R}^n) \in \arg \min_{w \in H, w_0 \in \mathbb{R}, \xi \in \mathbb{R}^n} \left(\frac{1}{2}\|w\|^2 + C \sum_{i=1}^{n} \xi_i \right)$$
$$\text{sous } \xi_i \geq 0, \, \forall i \in \{1, \ldots, n\},$$
$$y_i(\langle w, \Phi(x_i) \rangle + w_0) \geq 1 - \xi_i, \, \forall i \in \{1, \ldots, n\}.$$

Le paramètre C contrôle la complexité du classificateur, déterminant le coût des erreurs de classification.

3 Application des SVM pour la classification des iris

3.1 Introduction au jeu de données Iris

Dans cette section, nous introduirons le jeu de données Iris, qui est un ensemble de données largement utilisé pour tester les algorithmes de classification. Il contient 150 échantillons de fleurs d'iris, répartis en trois espèces (Iris setosa, Iris versicolor et Iris virginica), avec quatre caractéristiques mesurées pour chaque échantillon : la longueur et la largeur des sépales, ainsi que la longueur et la largeur des pétales.

3.2 Chargement des bibliothèques nécessaires

Pour réaliser notre classification, nous utiliserons les bibliothèques suivantes :

— **NumPy** : pour la manipulation des tableaux et des calculs numériques.

— **Matplotlib** : pour la visualisation des données.

— **scikit-learn** : pour l'implémentation des SVM et le traitement du jeu de données.

Voici le code pour importer ces bibliothèques :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

3.3 Chargement du jeu de données Iris

Nous allons charger le jeu de données Iris à partir de scikit-learn. Ce processus inclut l'accès aux données et leur séparation en caractéristiques (features) et étiquettes (labels).

```
# Chargement des données Iris et prétraitement
iris = datasets.load_iris()
X = iris.data
scaler = StandardScaler()  # Normalisation des données
X = scaler.fit_transform(X)
y = iris.target
```

Par le suite, nous allons garder uniquement deux classes pour effectuer une classification binaire. Pour ce faire, nous allons :

— Sélectionner les classes : Nous allons conserver uniquement les exemples appartenant aux classes 1 et 2 (Iris versicolor et Iris virginica), ce qui nous permettra d'effectuer une classification binaire.

— Réduire les caractéristiques : Pour faciliter la visualisation, nous allons ne conserver que les deux premières caractéristiques (longueur et largeur des sépales) du jeu de données.

— Mélanger et diviser les données : Ensuite, nous allons mélanger les données pour garantir une répartition aléatoire des exemples, puis les diviser en ensembles d'entraînement et de test, avec une proportion de 50

Voici le code correspondant :

```
# On garde uniquement deux classes pour avoir une classification binaire (classe 1 et 2)
X = X[y != 0, :2]  # Prendre seulement 2 caractéristiques pour visualisation
y = y[y != 0]

# Mélanger les données et diviser en train/test
X, y = shuffle(X, y, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=42)
```

3.4 Noyau linéaire

Dans cette étape, nous allons utiliser un noyau linéaire pour le classificateur SVM et effectuer une recherche des meilleurs paramètres de régularisation à l'aide de la validation croisée. Nous allons explorer différentes valeurs pour le paramètre C, qui contrôle la complexité du modèle. Un C faible peut entraîner un modèle trop simple, tandis qu'un C élevé peut mener à un modèle plus complexe qui pourrait surajuster les données.

Voici le code correspondant :

```
# Noyau linéaire
# GridSearch pour le noyau linéaire avec différents paramètres de régularisation (C)
parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3, 200))}
# Recherche des meilleurs paramètres avec validation croisée
clf_linear = GridSearchCV(SVC(), parameters, cv=5)
clf_linear.fit(X_train, y_train)  # Entraînement du modèle

# Calcul du score (précision) sur les ensembles d'entraînement et de test
print('Meilleurs paramètres pour noyau linéaire:', clf_linear.best_params_)
print('Score de généralisation pour noyau linéaire: train = %.2f, test = %.2f' %
      (clf_linear.score(X_train, y_train), clf_linear.score(X_test, y_test)))
```

— **Meilleurs paramètres** : { 'C' : `np.float64(0.29673024081888694)`, 'kernel' : 'linear' }
— **Score de généralisation** :
  — Entraînement : 0.66
  — Test : 0.66

Ces résultats indiquent que le modèle présente une performance de 66% tant sur l'ensemble d'entraînement que sur l'ensemble de test, suggérant qu'il est capable de généraliser les données sans surajuster.

4 Comparaiosn des modèles SVM avec noyaux linéaire et polynomial

4.1 Noyau polynomial

Dans cette étape, nous allons utiliser un noyau polynomial pour le classificateur SVM. Nous effectuerons une recherche des meilleurs paramètres à l'aide de la validation croisée, en explorant différentes valeurs pour les paramètres C, γ, et le degré du polynôme.

Voici le code correspondant :

```
# Noyau polynomial
# GridSearch pour le noyau polynomial avec paramètres C, gamma, et degree
Cs = list(np.logspace(-3, 3, 5))
gammas = 10. ** np.arange(1, 2)
degrees = np.r_[2, 3]

parameters = {'kernel': ['poly'], 'C': Cs, 'gamma': gammas, 'degree': degrees}
# Recherche des meilleurs paramètres pour noyau polynomial
clf_poly = GridSearchCV(SVC(), parameters, cv=5)
clf_poly.fit(X_train, y_train)  # Entraînement du modèle

# Affichage des meilleurs paramètres et des scores
print('Meilleurs paramètres pour noyau polynomial:', clf_poly.best_params_)
print('Score de généralisation pour noyau polynomial: train = %.2f, test = %.2f' %
      (clf_poly.score(X_train, y_train), clf_poly.score(X_test, y_test)))
```

Après avoir effectué la recherche des meilleurs paramètres, nous avons obtenu les résultats suivants :

- **Meilleurs paramètres** : { 'C' : `np.float64(0.001)`, 'degree' : `np.int64(2)`, 'gamma' : `np.float64(10.0)`, 'kernel' : 'poly' }
- **Score de généralisation** :
  - Entraînement : 0.64
  - Test : 0.44

4.2 Comparaison :

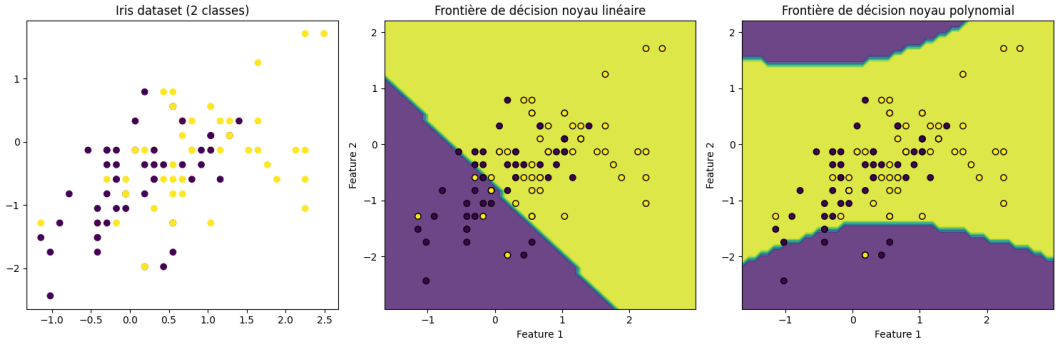

FIGURE 1 – Frontière de décision

Cette figure ci-dessus, représente un SVM avec un noyau polynomial de degré 2, montre une séparation quadratique des données avec les paramètres optimaux : 'C' : 0,001, 'degree' : 2, 'gamma' : 10.0, 'kernel' : 'poly'. Bien que ce modèle puisse modéliser des frontières de décision plus complexes, il présente une performance inférieure, avec un score de généralisation de 0,64 sur l'entraînement et 0,44 sur le test.

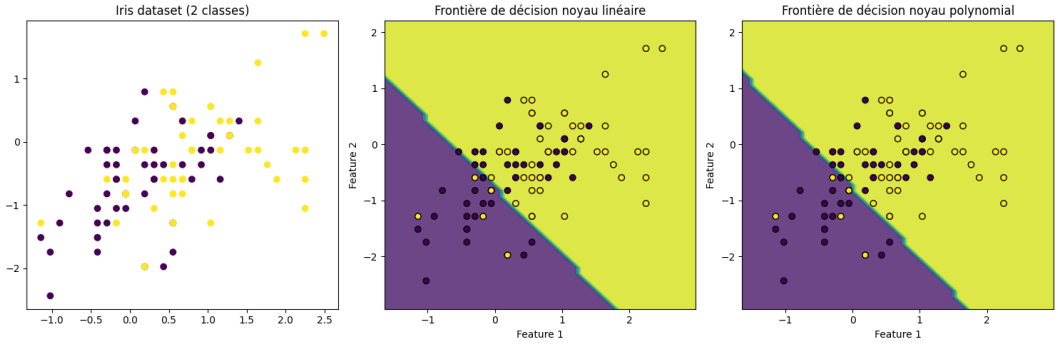

FIGURE 2 – Frontière de décision

En revanche, la figure ci-dessus, représente un SVM avec un noyau polynomial de degré 1. Le faible paramètre $C = 0,001$ indique une grande marge de séparation, entraînant un underfitting, où le modèle ne capture pas suffisamment d'informations à partir des données d'entraînement. En revanche, le modèle avec un noyau linéaire (degré 1) et $C = 0,031$ obtient de meilleurs scores de $0,66$ sur les deux ensembles. Cela suggère qu'une séparation linéaire est plus appropriée pour ce problème.

5 Analyse de l'impact de la régularisation sur le modèle SVM avec noyau linéaire

Dans cette section, nous allons examiner l'impact de la diminution du paramètre de régularisation C sur le modèle SVM avec un noyau linéaire.Avant d'aborder l'impact du paramètre C, nous allons d'abord lancer le script `svm_gui.py`, qui permet de visualiser et manipuler en temps réel les décisions d'un SVM. Ce script propose une interface graphique où l'on peut créer des points de données et observer comment le modèle SVM sépare les classes.

Pour générer un jeu de données déséquilibré avec une répartition de 90% de points dans une classe et 10% dans l'autre, suivez les étapes suivantes :

— **Création des points rouges (classe positive)** : Cliquez sur l'interface graphique avec le bouton gauche de la souris pour ajouter des points rouges représentant la classe positive.
— **Création des points noirs (classe négative)** : Utilisez le bouton droit de la souris pour ajouter des points noirs représentant la classe négative.

En générant un jeu de données très déséquilibré, nous allons pouvoir observer comment le modèle SVM se comporte en présence de classes déséquilibrées.

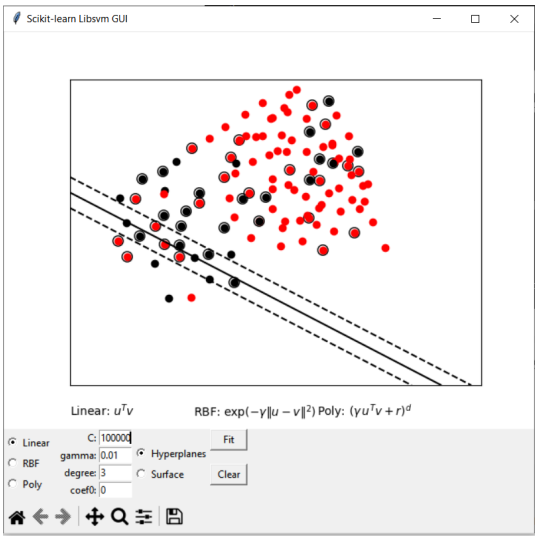

FIGURE 3 – Avec un C élevé

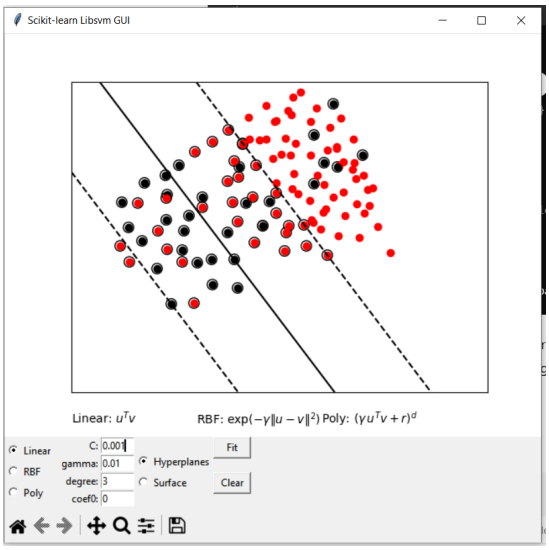

FIGURE 4 – Avec un C faible

Lorsque le paramètre de régularisation C est élevé (par exemple avec $C = 10000$), le SVM impose une séparation stricte entre les classes, minimisant les erreurs de classification mais avec une marge très étroite. En réduisant C (par exemple à $C = 0.001$), le modèle devient plus souple, élargissant la marge au prix de quelques erreurs, surtout pour les points proches de la frontière de décision. Avec un C encore plus faible (par exemple $C = 0.00001$), le SVM priorise l'élargissement maximal de la marge, mais cela entraîne une classification moins précise,

particulièrement pour la classe minoritaire. Cela montre qu'un C trop faible favorise une généralisation excessive, avec une perte de précision au profit d'une marge plus large.

6 Introduction à la classification de visages

Dans cette section, nous allons travailler sur un problème de classification de visages en utilisant le jeu de données Labeled Faces in the Wild (LFW). Ce jeu de données est accessible à l'adresse suivante : `http://vis-www.cs.umass.edu/lfw/lfw-funneled.tgz`.

6.1 Chargement des données

Nous allons charger le jeu de données et extraire les caractéristiques pertinentes pour la classification. Voici un exemple de code pour charger les données et préparer les images

```
from sklearn.datasets import fetch_lfw_people

# Chargement des données
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4,
                              color=True, funneled=False, slice_=None,
                              download_if_missing=True)
```

6.2 Examen des dimensions des images

Après avoir chargé les données, nous examinons les tableaux d'images pour en déterminer leurs dimensions. Cela nous permettra de mieux comprendre la structure des données.

Voici le code pour cette étape :

```
# Examinez les tableaux d'images pour en déterminer les dimensions
images = lfw_people.images
n_samples, h, w, n_colors = images.shape
```

6.3 Identification des étiquettes

L'étiquette à prédire est l'identifiant de la personne. Nous allons extraire les noms des cibles à partir des données.

Voici le code :

```
# L'étiquette à prédire est l'identifiant de la personne
target_names = lfw_people.target_names.tolist()
```

6.4 Sélection des paires à classifier

Pour notre classification, nous choisissons une paire de personnes à classifier, par exemple, *Tony Blair* et *Colin Powell*.

Voici le code correspondant :

```
# Choisissez une paire à classifier, par exemple
names = ['Tony Blair', 'Colin Powell']
# names = ['Donald Rumsfeld', 'Colin Powell']
```

6.5 Préparation des images pour la classification

Nous allons maintenant préparer les images correspondant aux deux personnes sélectionnées pour effectuer la classification. Nous construisons également le tableau des étiquettes.

Voici le code :

```
idx0 = (lfw_people.target == target_names.index(names[0]))
idx1 = (lfw_people.target == target_names.index(names[1]))
images = np.r_[images[idx0], images[idx1]]
n_samples = images.shape[0]
y = np.r_[np.zeros(np.sum(idx0)), np.ones(np.sum(idx1))].astype(int)

# Tracez un ensemble d'échantillons des données
plot_gallery(images, np.arange(12))
plt.show()
```

la figure suivante montre donc l'échantillon des données pour visualiser les images de *Tony Blair* et *Colin Powell*

FIGURE 5 – classification des visages

6.6 Extraction des caractéristiques

Nous allons extraire les caractéristiques à partir des images de visages. Pour ce faire, nous utiliserons uniquement les illuminations, ce qui nous permettra de simplifier le modèle. Voici le code correspondant :

```
# Extraire les caractéristiques
# Caractéristiques utilisant uniquement les illuminations.
X = (np.mean(images, axis=3)).reshape(n_samples, -1)

# Optionnel : Ou calculez les caractéristiques
#en utilisant les couleurs (3 fois plus de caractéristiques)
# X = images.copy().reshape(n_samples, -1)
```

Avant de procéder à l'entraînement, nous normalisons les caractéristiques pour qu'elles aient une moyenne nulle et un écart-type de un. Cela aide le modèle à converger plus rapidement.

Voici le code :

```
# Scale features
X -= np.mean(X, axis=0)
X /= np.std(X, axis=0)
```

Nous allons ensuite diviser les données en ensembles d'entraînement et de test, en utilisant 50% des données pour chaque ensemble. Cela nous permettra d'évaluer la performance du modèle,comme le montre le code suivant :

```
# Split data into a half training and half test set
indices = np.random.permutation(X.shape[0])
train_idx, test_idx = indices[:X.shape[0] // 2], indices[X.shape[0] // 2:]
X_train, X_test = X[train_idx, :], X[test_idx, :]
y_train, y_test = y[train_idx], y[test_idx]
images_train, images_test = images[train_idx, :, :, :], images[test_idx, :, :, :]
```

Nous allons maintenant entraîner un modèle SVM avec un noyau linéaire en testant différentes valeurs du paramètre de régularisation C et observer comment cela influence les performances.

Voici le code :

```
# L'étiquette à prédire est l'identifiant de la personne
print("--- Linear kernel ---")
print("Fitting the classifier to the training set")
t0 = time()

# Fit a classifier (linear) and test all the Cs
Cs = 10. ** np.arange(-5, 6)
scores = []

# Boucle sur les valeurs de C
for C in Cs:
    # Créer un classificateur SVM avec le noyau linéaire et le paramètre C
    clf = svm.SVC(kernel='linear', C=C)
    clf.fit(X_train, y_train)  # Entraîner le modèle sur les données d'entraînement

    # Prédire les labels pour l'ensemble de test
    y_pred = clf.predict(X_test)

    # Calculer le score (précision) et l'ajouter à la liste des scores
    score = np.mean(y_pred == y_test)
    scores.append(score)

# Trouver l'indice du meilleur score
ind = np.argmax(scores)
print("Best C: {}".format(Cs[ind]))
```

— **Meilleur paramètre** C : 1.0
— **Scores de généralisation** :

— Entraînement : À insérer
— Test : À insérer

Afficher ensuite l'erreur de prédiction en fonction du paramètre de régularisation C sur une échelle logarithmique.

```
plt.figure()
plt.plot(Cs, scores)
plt.xlabel("Paramètre de régularisation C")
plt.ylabel("Scores de prédiction")
plt.xscale("log")
plt.tight_layout()
plt.show()
print("Best score: {}".format(np.max(scores)))
```

— Linear kernel —
Fitting the classifier to the training set
Best C : 0.001
Best score : 0.9210526315789473
Predicting the people names on the testing set

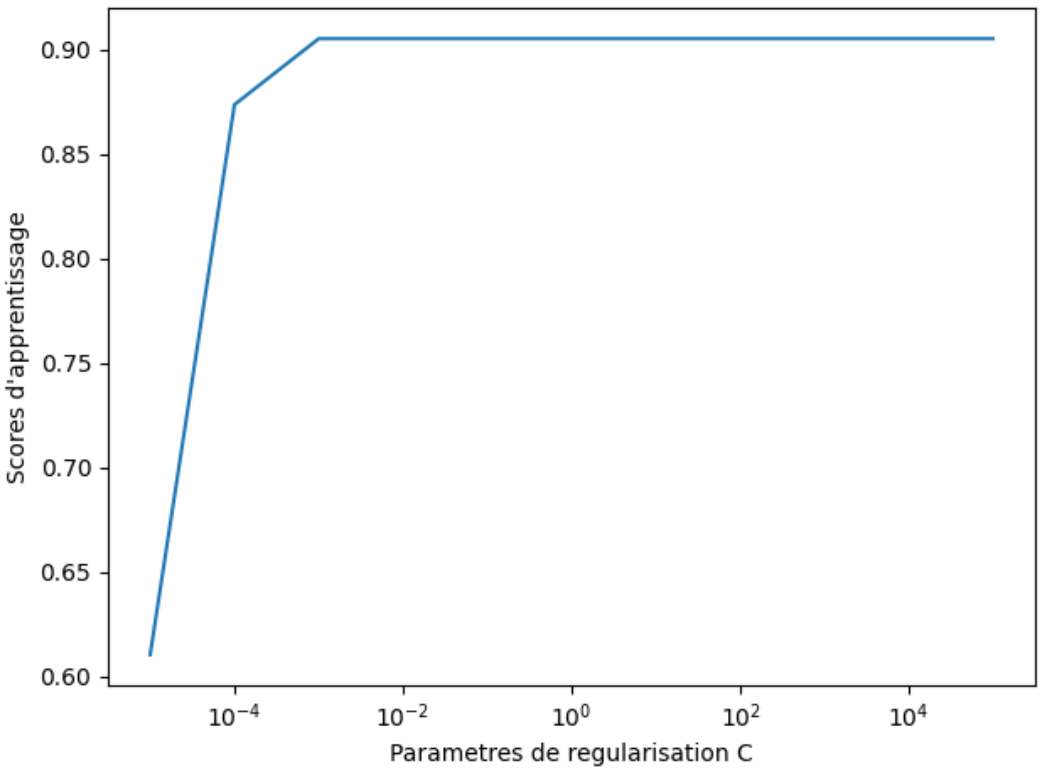

FIGURE 6

Ce graphe montre que la meilleure régularisation est obtenue avec $C = 0.001$, où le modèle atteint une précision de 90%. Pour des valeurs très faibles de C, la précision est d'environ 65%, mais elle augmente rapidement jusqu'à $C = 0.001$, avant de se stabiliser. Cela indique que cette valeur de C offre un bon équilibre entre la généralisation et la performance pour cette tâche de reconnaissance de visages.

6.7 Évaluation finale

Nous allons prédire les étiquettes pour les images de test avec le meilleur classificateur trouvé et afficher la précision.

Voici le code :

```
print("Predicting the people names on the testing set")
t0 = time()

# Créez un classificateur SVM avec le meilleur paramètre C
best_C = Cs[ind]
clf = svm.SVC(kernel='linear', C=best_C)
clf.fit(X_train, y_train)  # Entraîner le modèle sur les données d'entraînement

print("done in %0.3fs" % (time() - t0))
# Afficher la précision du modèle
print("Chance level : %s" % max(np.mean(y), 1. - np.mean(y)))
print("Accuracy : %s" % clf.score(X_test, y_test))
```

done in 1185.726s
Chance level : 0.6210526315789474
Accuracy : 0.7
La création du modèle SVM avec le paramètre optimal C a permis d'obtenir une précision de 70% sur l'ensemble de test. Ce score est supérieur au "chance level" de 62.1%, qui correspond à la précision obtenue si le modèle prédisait uniquement la classe majoritaire. Le temps d'entraînement du modèle a été relativement long (1185.726 secondes), probablement en raison de la complexité des données. Ces résultats montrent que le modèle SVM est capable de généraliser correctement, bien que des améliorations soient encore possibles.

6.8 Évaluation qualitative des prédictions

Nous évaluons la qualité des prédictions du modèle en comparant les étiquettes prédites (`y_pred`) aux étiquettes réelles (`y_test`). Les images des visages sont affichées avec leurs prédictions correspondantes.

Voici le code utilisé pour cette étape :

```
# Qualitative evaluation of the predictions using matplotlib
prediction_titles = [title(y_pred[i], y_test[i], names)
                     for i in range(y_pred.shape[0])]
```

```
# Afficher la galerie d'images avec les prédictions
plot_gallery(images_test, prediction_titles)
plt.show()

# jeter un oeil sur le coefficients
plt.figure()
plt.imshow(np.reshape(clf.coef_, (h, w)))
plt.show()
```

FIGURE 7

Cette série d'images compare les prédictions du modèle avec les vraies étiquettes.

Images avec "predicted : Powell, true : Powell" : Le modèle identifie correctement Powell, indiquant une bonne reconnaissance.

Images avec "predicted : Blair, true : Blair" : La prédiction correcte pour Blair dé-

montre également la performance du modèle.

Cas d'erreurs : Certaines images montrent des erreurs, comme "Blair" prédit pour "Powell". Ces erreurs peuvent être dues à des similitudes entre visages ou à une mauvaise généralisation du modèle. Pour au mieux comprendre cet enjeu, nous allons observer la figure suivante :

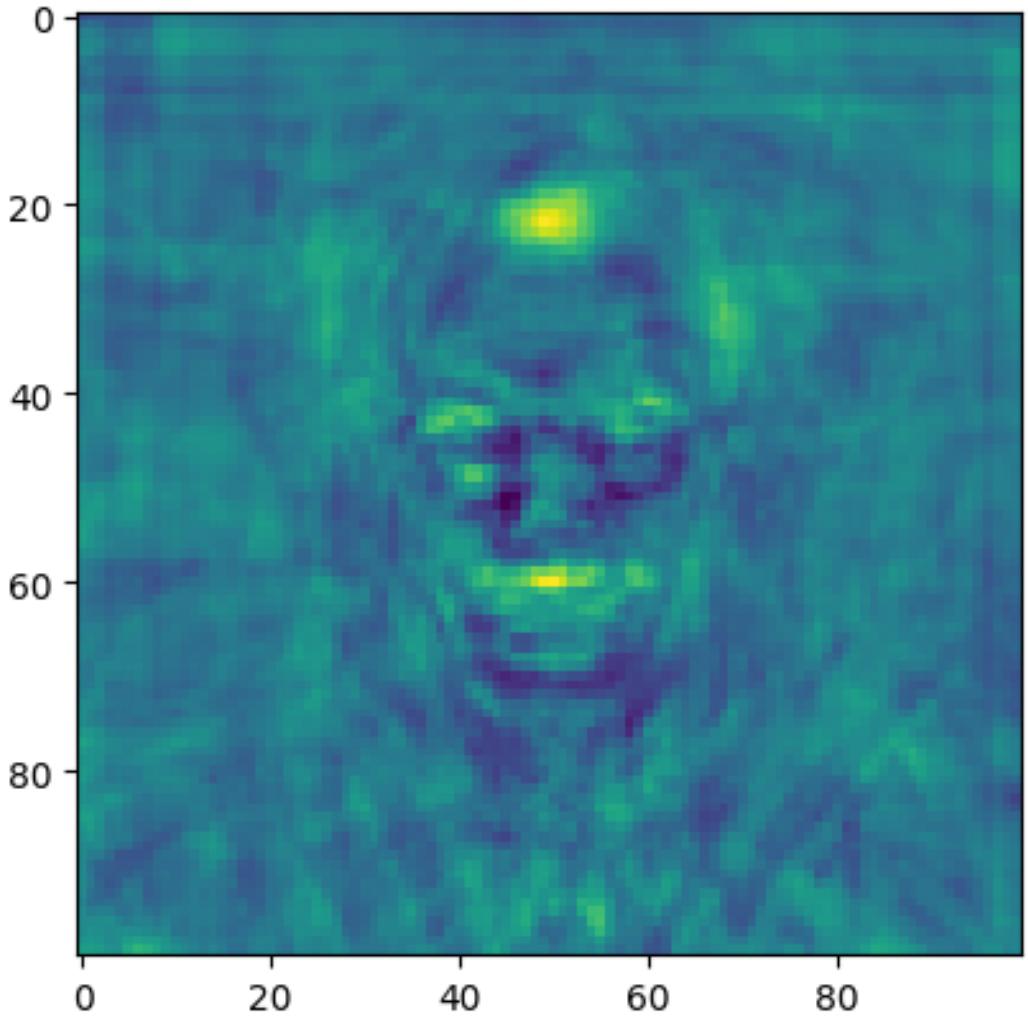

FIGURE 8

Cette image montre les coefficients du modèle SVM appris sur le jeu de données, visualisés comme une matrice représentant une image. Chaque pixel correspond à l'importance d'une caractéristique (ici, un pixel du visage) dans la décision de classification du modèle.

Zones lumineuses (jaune/vert clair) : elles indiquent les régions de l'image qui ont une forte influence sur la prédiction. Ici, les zones autour des yeux, du nez et de la bouche semblent être particulièrement importantes.

Zones sombres (bleu/noir) : elles correspondent aux parties de l'image qui ont peu ou pas

d'impact sur la décision du modèle. Le modèle semble se concentrer sur des zones clés du visage pour différencier les classes (ex. Powell vs. Blair), ce qui est cohérent avec les parties distinctives du visage humain pour la reconnaissance.

7 Évaluation du modèle SVM

Dans cette section, nous allons évaluer les performances d'un modèle SVM. Nous effectuerons d'abord l'évaluation sur un ensemble de données sans variables de nuisance, puis nous ajouterons des variables de nuisance et examinerons les performances du modèle.

Nous commençons par définir une fonction `run_svm_cv` qui prendra en entrée les données d'entrée $_X$ et les étiquettes $_y$. Cette fonction va diviser les données en ensembles d'entraînement et de test, puis effectuer une recherche de grille pour trouver le meilleur paramètre de régularisation C.

```
def run_svm_cv(_X, _y):
    _indices = np.random.permutation(_X.shape[0])
    _train_idx, _test_idx = _indices[:_X.shape[0] // 2], _indices[_X.shape[0] // 2:]
    _X_train, _X_test = _X[_train_idx, :], _X[_test_idx, :]

    _y_train, _y_test = _y[_train_idx], _y[_test_idx]

    _parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3, 5))}
    _svr = svm.SVC()
    _clf_linear = GridSearchCV(_svr, _parameters)
    _clf_linear.fit(_X_train, _y_train)

    print('Generalization score for linear kernel: %s, %s \n' %
          (_clf_linear.score(_X_train, _y_train), _clf_linear.score(_X_test, _y_test)))
```

7.1 Évaluation du modèle sans variable de nuisance

Nous évaluons d'abord le modèle sans ajouter de bruit aux données. Cela permet de voir comment le modèle se comporte sur un ensemble de données propre.

```
print("Score sans variable de nuisance")
run_svm_cv(X, y)  # Exécute la fonction avec les données sans bruit
```

7.2 Évaluation du modèle avec variables de nuisance

Nous ajoutons ensuite des variables de nuisance. Cela implique de générer un ensemble de données avec du bruit pour voir comment le modèle gère les données moins propres.

```python
print("Score avec variable de nuisance")
n_features = X.shape[1]
# On ajoute des variables de nuisance (bruit)
sigma = 1
noise = sigma * np.random.randn(n_samples, 300)  # Ajout de 300 features bruitées
X_noisy = np.concatenate((X, noise), axis=1)
X_noisy = X_noisy[np.random.permutation(X.shape[0])]

# Exécute la fonction avec les données bruitées
run_svm_cv(X_noisy, y)
```

L'ajout de variables de nuisance a un effet notable sur la performance du modèle. Les résultats montrent que sans ajout de bruit, le modèle atteint un score de généralisation parfait sur l'ensemble d'entraînement (1.0) et un score de 0.8789 sur l'ensemble de test.

Cependant, après l'ajout de 300 variables de nuisance, bien que le score sur l'ensemble d'entraînement reste élevé (0.9947), le score sur l'ensemble de test chute drastiquement à 0.5684. Cela indique que l'ajout de variables inutiles perturbe la capacité du modèle à généraliser correctement, ce qui entraîne une dégradation de ses performances sur des données non vues.

```
Score sans variable de nuisance:
Entraînement: 1.0, Test: 0.8789

Score avec variable de nuisance:
Entraînement: 0.9947, Test: 0.5684
```

Cette chute de performance montre clairement que le modèle SVM est affecté par l'ajout de bruit, car ces variables de nuisance complexifient inutilement la tâche de classification.

8 Amélioration de la prédiction avec réduction de dimension (PCA)

Pour améliorer la prédiction, nous avons utilisé une réduction de dimension à l'aide de l'objet `PCA` avec le solveur `svd_solver='randomized'`. Cela permet de conserver les informations essentielles tout en réduisant le nombre de caractéristiques, notamment après l'ajout de bruit (variables de nuisance).

```python
print("Score après réduction de dimension")

n_components = 100
pca = PCA(n_components=n_components).fit(X_noisy)

# Transformation des données bruitées avec PCA
X_noisy_pca = pca.transform(X_noisy)

# Affichage de la variance expliquée pour s'assurer de la qualité de la réduction
print(f"Variance expliquée avec {n_components} composantes principales :
```

```
{np.sum(pca.explained_variance_ratio_):.2f}")

# Application du modèle SVM sur les données réduites
run_svm_cv(X_noisy_pca, y)  # Utilisation du modèle SVM sur les données réduites par PCA
```

Le résultat obtenu après réduction de dimension montre que la variance expliquée avec les 20 premières composantes principales est de 0.68. Cela indique que 68% de la variance totale est conservée dans les données après réduction de dimension.

L'application de l'analyse en composantes principales (ACP) a permis de réduire la dimensionnalité des données après l'ajout de variables de nuisance. Avec 20 composantes principales, 68% de la variance totale des données a été capturée. Cela signifie que la majorité de l'information a été conservée tout en éliminant une partie du bruit inutile.

Lorsque le nombre de composantes est inférieur à 20, une plus faible proportion de la variance est expliquée, ce qui entraîne une perte d'information et potentiellement une dégradation des performances du modèle. En revanche, en augmentant le nombre de composantes, la quantité d'information capturée augmente, mais avec une complexité plus élevée.

9 Conclusion

Dans ce TP, nous avons appliqué les machines à vecteurs de support (SVM) sur différentes tâches de classification, en comparant l'effet de différents noyaux et paramètres de régularisation. Le noyau linéaire a montré de bonnes performances sur le jeu de données Iris, surpassant le noyau polynomial en termes de généralisation.

Nous avons ensuite étudié l'impact des variables de nuisance, qui ont significativement dégradé les performances du modèle. Enfin, l'application de la réduction de dimension via PCA a permis d'améliorer les performances après l'ajout de bruit, en conservant une proportion importante de la variance des données.

Ces expériences ont permis d'approfondir la compréhension des SVM et des techniques de réduction de dimension.