Université de Montpellier

Master 2 Statistiques et Sciences des Données

TP : Support Vector Machine (SVM)

Auteur : Maxence Lamure

2024 - 2025

Table des matières

1 Introduction aux SVM

Les Support Vector Machines (SVM) représentent une famille d'algorithmes d'apprentissage supervisé, largement utilisés pour les problèmes de classification et, dans une moindre mesure, de régression. Le principe des SVM repose sur la recherche d'un hyperplan séparateur optimal entre différentes classes de données. Cet hyperplan est défini de manière à maximiser la marge entre les points de chaque classe les plus proches de cette frontière, appelés vecteurs de support. La fonction de classification pour un SVM est donnée par :

$$f(x) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + b\right)$$

où α_i sont les coefficients du modèle, y_i les labels, $K(x_i, x)$ est le noyau choisi, et b est le biais.

Dans le cas où les données sont linéairement séparables, le SVM détermine un hyperplan qui maximise la distance entre les vecteurs de support des différentes classes. Cependant, toutes les données ne sont pas toujours linéairement séparables. Pour répondre à cette problématique, les SVM utilisent la méthode du noyau (kernel) qui permet de transformer les données en un espace de dimension supérieure où une séparation linéaire devient possible.

Dans ce TP, nous allons principalement travailler avec deux types de noyaux :

- Cas linéaire, adapté lorsque les données peuvent être séparées par un hyperplan.
- Cas polynomial, utile pour capturer des relations non linéaires en augmentant la complexité de la frontière.

L'objectif de ce travail est de comparer les performances des SVM en fonction des noyaux utilisés et de l'ajustement de leurs hyperparamètres. En modifiant des paramètres comme le paramètre de régularisation ou les paramètres des noyaux, nous chercherons à comprendre comment ces manipulations influencent la qualité de la classification.

2 Application

2.1 Question 1 : Noyau linéaire

À partir du code donnée en annexe, nous classifions la classe 1 contre la classe 2 en utilisant uniquement les deux premières variables caractéristiques de l'ensemble Iris et un noyau linéaire. Nous séparons équitablement les données d'entraînement et de test en deux échantillons distincts. Dans le cas d'un noyau linéaire, la fonction de noyau est définie comme suit :

$$K(x, x') = \langle x, x' \rangle$$

L'analyse des résultats obtenus avec les paramètres suivants :

```
parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3, 200))}
clf_linear = GridSearchCV(SVC(), parameters, cv=5)
clf_linear.fit(X_train, y_train)
```

montre que le modèle SVC avec noyau linéaire atteint un score de 72% sur les données d'entraînement, mais ce score chute à 68% sur les données de test, ce qui suggère un problème de généralisation. Cela pourrait indiquer un léger surentraînement ou une incapacité du modèle à capturer toute la complexité des données.

2.2 Question 2 : Noyau polynomial

Nous utilisons un noyau polynomial pour comparer ses performances au noyau linéaire. Le noyau polynomial, permettant de capturer des relations non linéaires, est donné par :

$$K(x, x') = (\alpha + \beta\langle x, x' \rangle)^{\delta}, \ \delta > 0$$

où α, β, et δ sont des paramètres ajustables.

Le code suivant :

```
parameters = {'kernel': ['poly'], 'C': Cs, 'gamma': gammas, 'degree': degrees}
clf_poly = GridSearchCV(SVC(), param_grid=parameters, n_jobs=-1)
clf_poly.fit(X_train, y_train)
```

permet de trouver les paramètres optimaux avec la validation croisée :

```
{'C': 0.03162277660168379, 'degree': 1, 'gamma': 10.0, 'kernel': 'poly'\}
```

Le modèle SVC avec un noyau polynomial atteint un score de 74% sur les données d'entraînement et 72% sur les données de test, ce qui montre une performance légèrement meilleure sur l'ensemble d'entraînement par rapport au modèle avec noyau linéaire, mais une performance de test comparable.

La Figure 1 illustre les frontières de décision obtenues avec un noyau linéaire et un noyau polynomial, ainsi que la distribution des observations selon ces deux modèles.

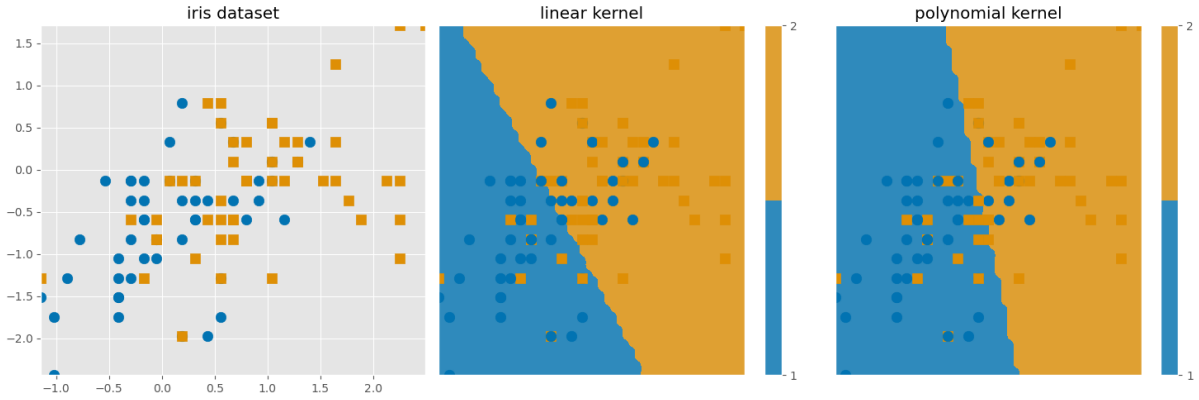

Figure 1: Performances des SVM appliqués au jeu de données iris pour un noyau linéaire et polynomial

2.3 Question 3 : SVM GUI

En diminuant le paramètre C dans un SVM avec un noyau linéaire, on observe que le modèle devient plus tolérant aux erreurs sur les données d'entraînement. Concrètement, cela se traduit par une marge plus large entre les classes, au prix de certaines mauvaises classifications. Le modèle privilégie ainsi une meilleure généralisation, en évitant de se concentrer trop strictement sur chaque point d'entraînement. Ce compromis est représenté par le problème d'optimisation suivant :

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i$$

sujet aux contraintes $y_i(\langle w, x_i\rangle + b) \geq 1 - \xi_i$ et $\xi_i \geq 0$.

Dans le cadre d'un jeu de données déséquilibré, comme celui utilisé ici (90% des points dans une classe et 10% dans l'autre), une valeur plus faible de C tend à favoriser la classe majoritaire. En effet, la frontière de décision devient moins précise et plus influencée par la majorité des points, ce qui peut entraîner une dégradation de la performance sur la classe minoritaire, avec davantage de points mal classifiés dans cette catégorie.

3 Classification de visages

Pour classifier les visages, nous utilisons l'ensemble de données LFW (Labeled Faces in the Wild) puis nous sélectionnons deux personnes spécifiques (Tony Blair et Colin Powell) dont nous disposons au moins 70 photos et réduisons la taille des images par optimisation d'espace. Nous préparons ensuite les données dont un échantillon se trouve dans la Figure 2 pour une tâche de classification binaire.

Figure 2: Echantillon des données LFW

3.1 Question 4 : Paramètre de régularisation

Nous étudions l'influence du paramètre de régularisation C en ajustant les valeurs sur une échelle logarithmique entre 10^{-5} et 10^{5} et en observant les scores :

```
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))
```

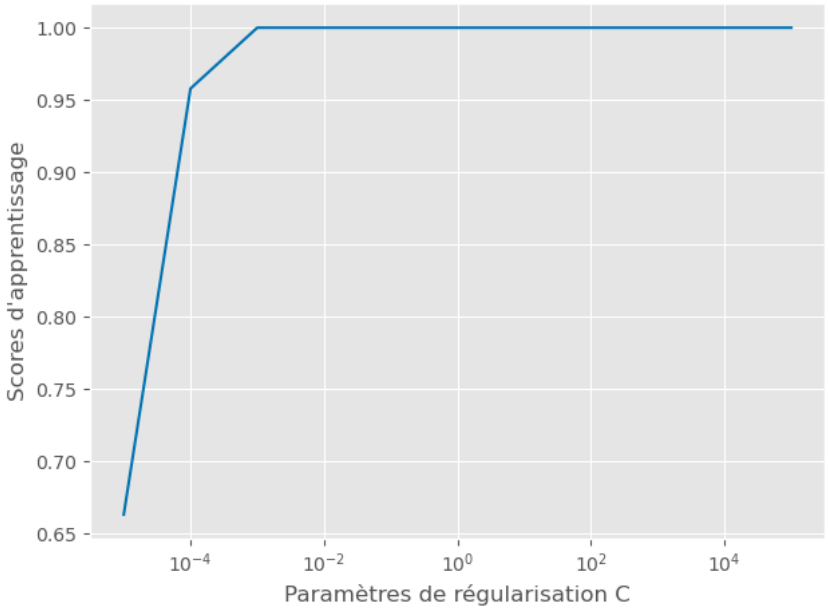

Figure 3: Graphe montrant le score d'apprentissage en fonction du paramètre de régularisation C

Comme montré dans la Figure 3, le meilleur paramètre de régularisation est $C = 10^{-3}$, pour un score de 1. Ce score reste le même pour un C plus grand, ce qui montre la stabilité de la classification.

Nous réentraînons à présent le modèle avec ce C optimal et réalisons les prédictions finales sur les données de test. Nous comparons ensuite la précision obtenue à celle attendue d'un modèle prédictif aléatoire, servant de référence pour évaluer les performances.

```
clf = SVC(kernel='linear', C=best_C)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print("Chance level : %s" % max(np.mean(y), 1. - np.mean(y)))
print("Accuracy : %s" % clf.score(X_test, y_test))
```

Nous obtenons alors une précision de 91% et un niveau de chance de 62%. Le modèle affiche donc une performance nettement meilleure que l'aléatoire, ce qui montre qu'il a bien appris à différencier les classes à partir des données fournies.

Les résultats sont illustrés dans la Figure 4 par le code suivant :

```
prediction_titles = [title(y_pred[i], y_test[i], names)
                     for i in range(y_pred.shape[0])]

plot_gallery(images_test, prediction_titles)
```

Figure 4: Echantillon de prédiction du modèle pour les données LFW

La Figure 5 présente quant à elle les coefficients du modèle linéaire appris sous forme de carte de chaleur sur une grille bidimensionnelle :

```
plt.imshow(np.reshape(clf.coef_, (h, w)))
```

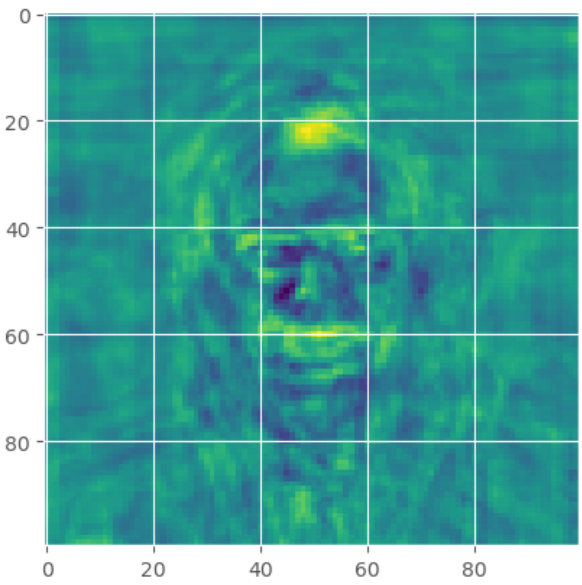

Figure 5: Coefficients du modèle linéaire appris sous forme de carte de chaleur

Les variations de couleurs représentent les différences d'intensité des coefficients ou des données dans cette grille. Typiquement, des zones plus claires indiquent des valeurs plus élevées, tandis que des zones plus sombres signalent des valeurs plus faibles.

Nous constatons alors que le classifieur se base principalement sur certains éléments tels que l'implantation des cheveux au-dessus du crâne et sur les côtés, les sourcils/yeux, le nez et la bouche pour sa prédiction.

3.2 Question 5 : Variables de nuisance

Nous ajoutons ici des variables de nuisance pour évaluer l'impact sur les performances du modèle dont le score est calculé par la fonction suivante :

```
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 (train), %s (test) \n' %
          (_clf_linear.score(_X_train, _y_train), _clf_linear.score(_X_test,
              _y_test)))
```

Sans variables de nuisance, le modèle atteint un score de 1 pour les données d'entraînement et de 93% pour les données de test, montrant une très bonne performance et sa capacité à généraliser sur de nouvelles données.

Avec variables de nuisance en revanche, calculé selon :

```
sigma = 1
noise = sigma * np.random.randn(n_samples, 300, )
X_noisy = np.concatenate((X, noise), axis=1)
X_noisy = X_noisy[np.random.permutation(X.shape[0])]
```

le modèle atteint certes un score de 1 pour les données d'entraînement mais chute significativement à 47% pour les données de test (inférieur au niveau de chance à 62%), indiquant une mauvaise robustesse face à de nouvelles données malgré une bonne performance pour les données d'entraînement.

3.3 Question 6 : Réduction de dimensions ACP

Après avoir analysé l'impact des variables de nuisance sur les performances du modèle, il peut être utile d'étudier une autre technique de prétraitement des données : la réduction de dimensions. Grâce à l'Analyse en Composantes Principales (ACP), l'objectif est de diminuer le nombre de variables tout en conservant un maximum d'information. Cette approche peut améliorer la performance du modèle en éliminant le bruit et en simplifiant la structure des données. Nous appliquerons donc l'ACP sur les données bruitées pour évaluer son effet sur la précision du modèle SVM, en testant trois scénarios : 40, 80 et 120 composantes. Le code correspondant à l'application de l'ACP est le suivant :

```
pca = PCA(n_components=n_components).fit(X_noisy)
X_pca = pca.transform(X_noisy)
```

Ainsi, après un temps de calcul relativement long, on obtient les résultats suivants :

- 40 composantes :

```
    0.7473684210526316 (train), 0.5684210526315789 (test)
```

- 80 composantes :

```
    0.868421052631579 (train), 0.4789473684210526 (test)
```

- 120 composantes :

```
    0.8631578947368421 (train), 0.5421052631578948 (test)
```

On observe une tendance au sur-apprentissage à mesure que le nombre de composantes augmente. En effet, avec 80 et 120 composantes, le score d'entraînement reste élevé, mais le score de test diminue. Cela suggère que le modèle devient trop complexe et s'adapte trop étroitement aux données d'entraînement, au détriment de la généralisation sur des données non vues.

La réduction du nombre de composantes à 40 permet d'obtenir un meilleur équilibre entre les scores d'entraînement et de test, bien que les scores globaux soient plus faibles que dans les cas avec plus de composantes. Cela montre que l'ACP aide à réduire le bruit et à améliorer la généralisation jusqu'à un certain point. Au-delà d'un certain nombre de composantes, l'ajout de plus de dimensions semble réintroduire du bruit ou rendre le modèle plus sujet au sur-apprentissage.

Annexes

Introduction

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import SVC
import sys
from pathlib import Path

from svm_source import *
from sklearn import svm
from sklearn import datasets
from sklearn.utils import shuffle
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.datasets import fetch_lfw_people
from sklearn.decomposition import PCA
from time import time

scaler = StandardScaler()

import warnings
warnings.filterwarnings("ignore")

plt.style.use('ggplot')
```

Application

Question 1 : Noyau linéaire

```python
iris = datasets.load_iris()
X = iris.data
X = scaler.fit_transform(X)
y = iris.target
X = X[y != 0, :2]
y = y[y != 0]

# split train test
X, y = shuffle(X, y)
# Separation des donnees en ensemble d'entrainement et de test (50% pour chaque )
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=42)

# Q1 Linear kernel

# Definition des parametres pour le noyau lineaire
parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3, 200))}

# Recherche sur grille pour trouver le meilleur parametre C
clf_linear = GridSearchCV(SVC(), parameters, cv=5)
clf_linear.fit(X_train, y_train)

# Affichage du score
print('Generalization score for linear kernel: %s (train), %s (test)' %
      (clf_linear.score(X_train, y_train),
       clf_linear.score(X_test, y_test)))
```

Question 2 : Noyau polynomial

```
# Q2 polynomial kernel
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}

# Recherche sur grille pour le noyau polynomial
clf_poly = GridSearchCV(SVC(), param_grid=parameters, n_jobs=-1)
clf_poly.fit(X_train, y_train)

# Affichage du meilleur parametre trouve
print('Best parameters for polynomial kernel:', clf_poly.best_params_)

# Affichage du score
print('Generalization score for polynomial kernel: %s (train), %s (test)' %
      (clf_poly.score(X_train, y_train),
       clf_poly.score(X_test, y_test)))

def f_linear(xx):
    """Classifier: needed to avoid warning due to shape issues"""
    return clf_linear.predict(xx.reshape(1, -1))

def f_poly(xx):
    """Classifier: needed to avoid warning due to shape issues"""
    return clf_poly.predict(xx.reshape(1, -1))

plt.ion()
plt.figure(figsize=(15, 5))
plt.subplot(131)
plot_2d(X, y)
plt.title("iris dataset")

plt.subplot(132)
frontiere(f_linear, X, y)
plt.title("linear kernel")

plt.subplot(133)
frontiere(f_poly, X, y)

plt.title("polynomial kernel")
plt.tight_layout()
plt.draw()
```

Question 3 : Noyau linéaire

```
# please open a terminal and run python svm_gui.py
# Then, play with the applet : generate various datasets and observe the
# different classifiers you can obtain by varying the kernel

# Q3
# python svm_gui.py
```

Classification des visages

Question 4 : Paramètre de régularisation

```
# Download the data and unzip; then load it as numpy arrays
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4,
                              color=True, funneled=False, slice_=None,
                              download_if_missing=True)
# data_home='.'

# introspect the images arrays to find the shapes (for plotting)
images = lfw_people.images
n_samples, h, w, n_colors = images.shape

# the label to predict is the id of the person
target_names = lfw_people.target_names.tolist()

# Pick a pair to classify such as
names = ['Tony Blair', 'Colin Powell']
# names = ['Donald Rumsfeld', 'Colin Powell']

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)

# Verification des tailles apres filtrage
print("Taille des images :", images.shape)
print("Taille de y :", y.shape)

# plot a sample set of the data
plot_gallery(images, np.arange(12))
plt.show()

# features using only illuminations
X = (np.mean(images, axis=3)).reshape(n_samples, -1)

# # or compute features using colors (3 times more features)
# X = images.copy().reshape(n_samples, -1)

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

# Split data into a half training and half test set
# X_train, X_test, y_train, y_test, images_train, images_test = \
#     train_test_split(X, y, images, test_size=0.5, random_state=0)
# X_train, X_test, y_train, y_test = \
#     train_test_split(X, y, test_size=0.5, random_state=0)

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, :, :, :]
```

```
# Q4
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 = []
```

```
for C in Cs:
    clf = SVC(kernel='linear', C=C)
    clf.fit(X_train, y_train)
    scores.append(clf.score(X_train, y_train))

ind = np.argmax(scores)
best_C = Cs[ind]
print("Best C: {}".format(Cs[ind]))

plt.figure()
plt.plot(Cs, scores)
plt.xlabel("Parametres de regularisation C")
plt.ylabel("Scores d'apprentissage")
plt.xscale("log")
plt.tight_layout()
plt.show()
print("Best score: {}".format(np.max(scores)))

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

```
# predict labels for the X_test images with the best classifier
clf = SVC(kernel='linear', C=best_C)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

print("done in %0.3fs" % (time() - t0))
# The chance level is the accuracy that will be reached when constantly
    predicting the majority class.
print("Chance level : %s" % max(np.mean(y), 1. - np.mean(y)))
print("Accuracy : %s" % clf.score(X_test, y_test))

# 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])]

plot_gallery(images_test, prediction_titles)
plt.show()

# Look at the coefficients
plt.figure()
plt.imshow(np.reshape(clf.coef_, (h, w)))
plt.show()
```

Question 5 : Variables de nuisance

```
# Q5
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 (train), %s (test) \n' %
```

```
          (_clf_linear.score(_X_train, _y_train), _clf_linear.score(_X_test,
              _y_test)))

print("Score sans variable de nuisance")
# Utilisation de la fonction run_svm_cv sur les donnees d'origine
run_svm_cv(X, y)

print("Score avec variable de nuisance")
n_features = X.shape[1]
# On rajoute des variables de nuisances
sigma = 1
noise = sigma * np.random.randn(n_samples, 300, )
X_noisy = np.concatenate((X, noise), axis=1)
X_noisy = X_noisy[np.random.permutation(X.shape[0])]
# Utilisation de la fonction run_svm_cv sur les donnees avec bruit
run_svm_cv(X_noisy, y)
```

Question 6 : Réduction de dimensions ACP

```
# Q6
print("Score apres reduction de dimension")

n_components = 40  # jouer avec ce parametre
pca = PCA(n_components=n_components).fit(X_noisy)
X_pca = pca.transform(X_noisy)

# Perform SVM with PCA-transformed data
print("Score avec PCA")
run_svm_cv(X_pca, y)

n_components = 80  # jouer avec ce parametre
pca = PCA(n_components=n_components).fit(X_noisy)
X_pca = pca.transform(X_noisy)

# Perform SVM with PCA-transformed data
print("Score avec PCA")
run_svm_cv(X_pca, y)

n_components = 120  # jouer avec ce parametre
pca = PCA(n_components=n_components).fit(X_noisy)
X_pca = pca.transform(X_noisy)

# Perform SVM with PCA-transformed data
print("Score avec PCA")
run_svm_cv(X_pca, y)
```