Rapport du TP sur les Machines à Vecteurs de Support (SVM)

Qingjian ZHU

Table des matières

0.1 Introduction

Ce rapport présente les résultats du travail pratique sur les Machines à Vecteurs de Support (SVM). L'objectif est d'explorer les principes fondamentaux des SVM, de comparer différents noyaux sur un jeu de données simple (Iris), puis d'appliquer ces techniques à une tâche plus complexe de classification de visages. Nous étudierons en particulier l'influence du paramètre de régularisation `C`, l'impact de l'ajout de variables non pertinentes (bruit), et l'amélioration des performances grâce à la réduction de dimension par Analyse en Composantes Principales (ACP/PCA).

1 Classification sur le jeu de données Iris

Nous commençons par une tâche de classification simple sur le jeu de données Iris. Nous ne conserverons que les classes 1 et 2, ainsi que les deux premiers attributs pour permettre une visualisation en 2D. Nous comparons les performances d'un noyau linéaire et d'un noyau polynomial.

```
##################################################
#               Iris Dataset
##################################################
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 (say 25% for the test)
# using train_test_split whithout shuffling (fix the random state = 42) for
↪  reproductibility
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
↪  random_state=42)
```

1.1 Q1: Noyau Linéaire

Nous utilisons `GridSearchCV` pour trouver le meilleur hyperparamètre de régularisation `C` pour un SVM à noyau linéaire.

```
# Q1 Linear kernel

# fit the model and select the best hyperparameter C
parameters = {'kernel': ['linear'], 'C': list(np.logspace(-3, 3, 200))}
clf1 = SVC()
clf_linear = GridSearchCV(clf1, parameters, n_jobs=-1)
clf_linear.fit(X_train, y_train)

# compute the score
print('Generalization score for linear kernel: %s, %s' %
      (clf_linear.score(X_train, y_train),
       clf_linear.score(X_test, y_test)))
```

```
Generalization score for linear kernel: 0.7466666666666667, 0.68
```

1.2 Q2: Noyau Polynomial

Nous étendons la recherche pour un noyau polynomial, en optimisant `C`, `gamma`, et le degré du polynôme `degree`.

```
# Q2 polynomial kernel
Cs = list(np.logspace(-3, 3, 5))
gammas = 10. ** np.arange(1, 2)
degrees = np.r_[1, 2, 3]

# fit the model and select the best set of hyperparameters
parameters = {'kernel': ['poly'], 'C': Cs, 'gamma': gammas, 'degree': degrees}
clf2 = SVC()
clf_poly = GridSearchCV(clf2, parameters, n_jobs=-1)
clf_poly.fit(X_train, y_train)

print(clf_poly.best_params_)
print('Generalization score for polynomial kernel: %s, %s' %
      (clf_poly.score(X_train, y_train),
       clf_poly.score(X_test, y_test)))
```

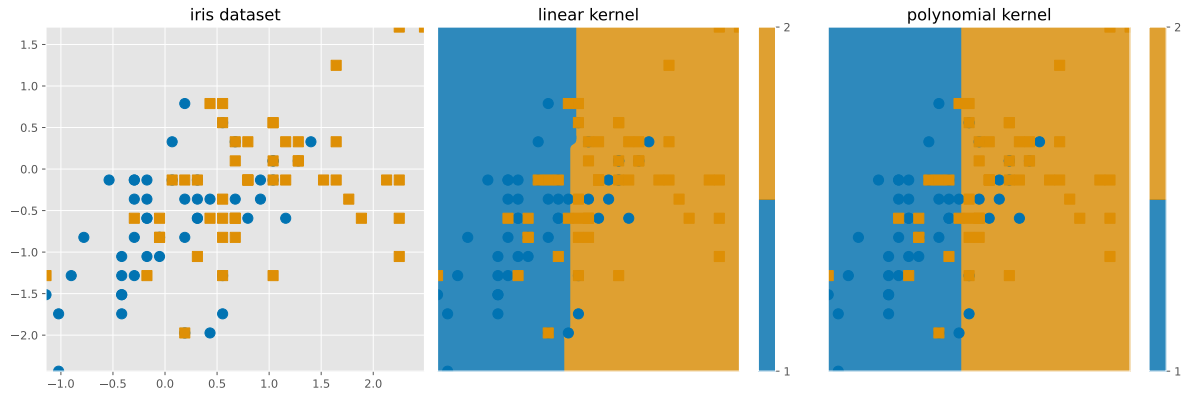
```
{'C': np.float64(0.03162277660168379), 'degree': np.int64(1), 'gamma': np.float64(10.0), 'ker
Generalization score for polynomial kernel: 0.7466666666666667, 0.68
```

1.3 Comparaison et Visualisation des Frontières

Visualisons les frontières de décision apprises par les deux modèles sur l'ensemble d'entraînement.

```
# display the results using frontiere
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):
    return clf_poly.predict(xx.reshape(1, -1))

# display the frontiere
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()
```



Analyse : Nous remarquons qu'en entraînant les modèles avec un noyau linéaire et un noyau polynomial, les meilleurs scores finaux sont quasiment identiques, s'élevant respectivement à 0,75 sur l'ensemble d'entraînement et 0,68 sur l'ensemble de test. Les deux modèles possèdent des frontières de décision très similaires. En réalité, cela s'explique par le fait que pour le noyau polynomial, le meilleur score est atteint avec un degré égal à 1('degree': np.int64(1)), ce qui le rend, en substance, équivalent à un noyau linéaire.

2 Tâche de Classification de Visages

Nous abordons maintenant une tâche plus complexe : distinguer les visages de deux personnalités publiques, Tony Blair et Colin Powell, à partir du jeu de données "Labeled Faces in the Wild" (LFW).

```
##################################################
#               Face Recognition Task
##################################################


###############################################################################
# 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)

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


```
########################################################################
# Extract features

# 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, :, :, :]

#############################################################################
# Quantitative evaluation of the model quality on the test set
```

2.1 Q4: Influence du paramètre de régularisation `C`

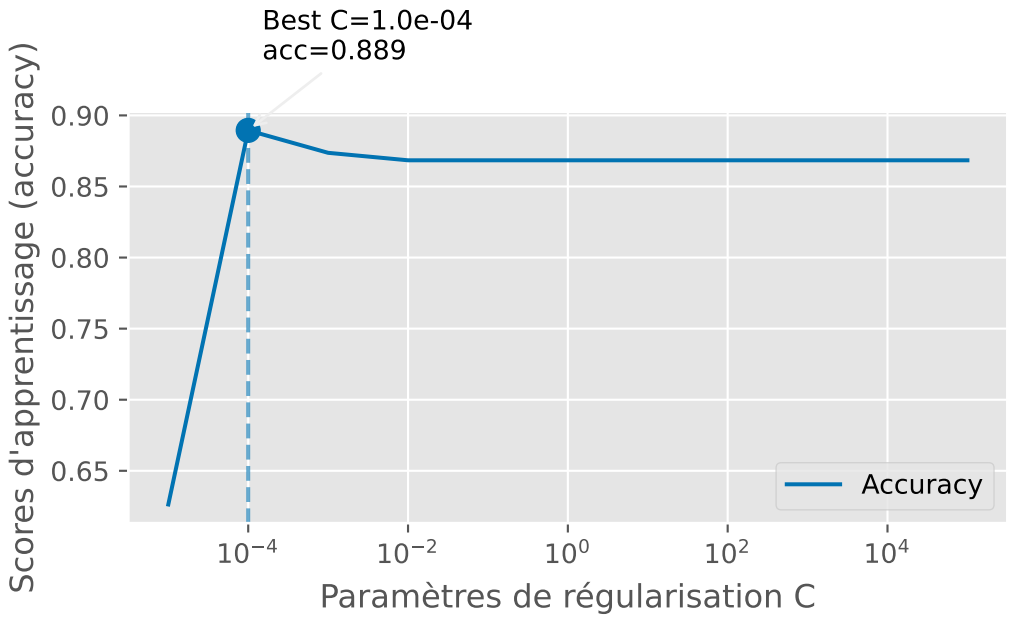
Le paramètre `C` contrôle le compromis entre la complexité du modèle (maximiser la marge) et l'erreur de classification sur l'ensemble d'entraînement. Un `C` faible favorise une grande marge au détriment de quelques erreurs (modèle simple, potentiellement sous-ajusté). Un `C` élevé pénalise fortement les erreurs, ce qui peut conduire à une marge plus petite et à un modèle complexe, risquant le sur-ajustement.

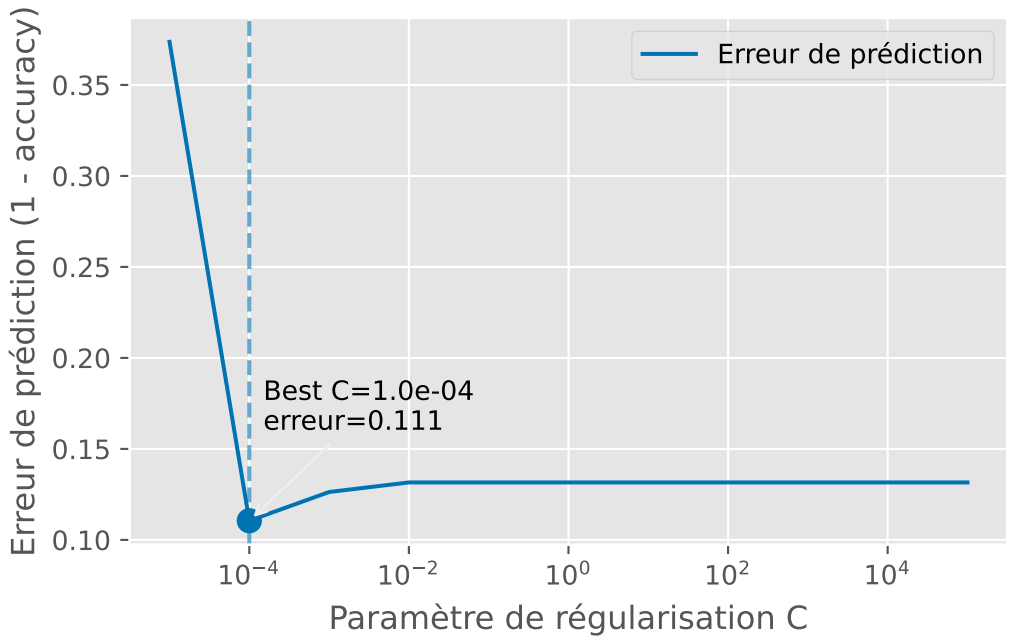
Nous entraînons un SVM linéaire pour une plage de valeurs de `C` et observons son score sur l'ensemble de test.

```
# 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_tmp = SVC(kernel='linear', C=C)
    clf_tmp.fit(X_train, y_train)
    scores.append(clf_tmp.score(X_test, y_test))

ind = int(np.argmax(scores))
best_C = Cs[ind]
best_acc = float(scores[ind])
best_err = 1.0 - best_acc

print("Best C: {}".format(best_C))

plt.figure()
plt.plot(Cs, scores, label="Accuracy")
plt.scatter([best_C], [best_acc], s=80, zorder=3)
plt.axvline(best_C, linestyle="--", alpha=0.6)
plt.annotate(
```

```
    "Best C={:.1e}\nacc={:.3f}".format(best_C, best_acc),
    xy=(best_C, best_acc),
    xytext=(1.5*best_C, min(1.0, best_acc + 0.05)),
    arrowprops=dict(arrowstyle="->", lw=1),
    fontsize=10
)
plt.xlabel("Paramètres de régularisation C")
plt.ylabel("Scores d'apprentissage (accuracy)")
plt.xscale("log")
plt.legend()
plt.tight_layout()
plt.show()

print("Best score (accuracy): {}".format(best_acc))

# Erreur de prédiction
errors = 1.0 - np.array(scores)

plt.figure()
plt.plot(Cs, errors, label="Erreur de prédiction")
plt.scatter([best_C], [best_err], s=80, zorder=3)
plt.axvline(best_C, linestyle="--", alpha=0.6)
plt.annotate(
    "Best C={:.1e}\nerreur={:.3f}".format(best_C, best_err),
    xy=(best_C, best_err),
    xytext=(1.5*best_C, min(1.0, best_err + 0.05)),
    arrowprops=dict(arrowstyle="->", lw=1),
    fontsize=10
)
plt.xlabel("Paramètre de régularisation C")
plt.ylabel("Erreur de prédiction (1 - accuracy)")
plt.xscale("log")
plt.legend()
plt.tight_layout()
plt.show()


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

```
--- Linear kernel ---
Fitting the classifier to the training set
Best C: 0.0001
Best score (accuracy): 0.8894736842105263
Predicting the people names on the testing set
```





Analyse : Le graphique illustre parfaitement le compromis biais-variance. Pour des valeurs de C très faibles, le modèle est sous-ajusté et ses performances sont médiocres sur les deux ensembles. À mesure que C augmente, le modèle s'adapte mieux aux données, et le score de

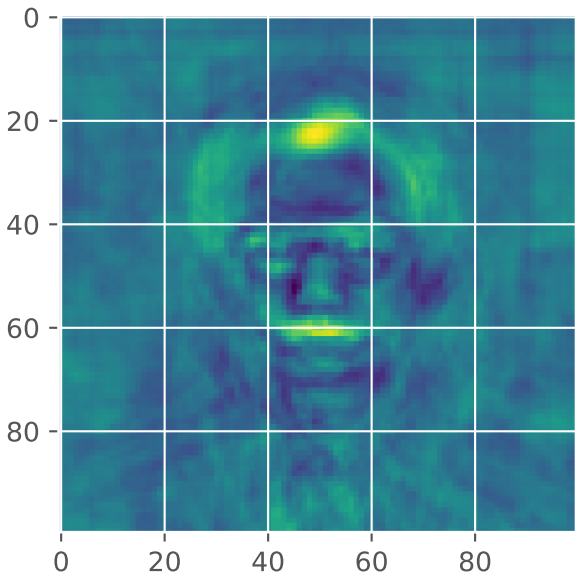
test s'améliore, atteignant un pic. Si C continue d'augmenter au-delà de ce point, le modèle commence à sur-apprendre les spécificités du jeu d'entraînement ; sa précision sur l'ensemble de test connaît alors une légère baisse avant de finalement stagner.

```
# predict labels for the X_test images with the best classifier
clf = SVC(kernel='linear', C=Cs[ind])
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))
```

```
done in 0.665s
Chance level : 0.6210526315789474
Accuracy : 0.8894736842105263
```

```
########################################################################
# 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()
```


predicted: Powell
true: Powell
predicted: Powell
true: Powell
predicted: Blair
true: Blair
predicted: Blair
true: Blair
predicted: Powell
true: Powell
predicted: Powell
true: Blair
predicted: Powell
true: Powell
predicted: Blair
true: Blair
predicted: Powell
true: Powell
predicted: Powell
true: Powell
predicted: Powell
true: Powell
predicted: Powell
true: Powell



Nous pouvons voir à travers cette partie du code que :

Le niveau de chance (chance level) est de 0,62. C'est la précision que nous obtiendrions en prédisant systématiquement la classe majoritaire (notre score de référence). Cette ligne de base sert à évaluer si le modèle a réellement appris une règle plus performante que la simple prédiction de la classe la plus fréquente.

De plus, la précision que nous avons obtenue en test atteint environ 0,9, ce qui est nettement supérieur au niveau de chance, prouvant ainsi que l'entraînement du modèle a bien été efficace.

Enfin, on peut observer que la dernière figure est une carte de chaleur des poids (weight heatmap). Les pixels ayant les poids les plus élevés correspondent aux zones les plus sensibles pour la distinction entre les deux classes. Cela nous donne une vue de l'interprétabilité du modèle, à la fois brute mais intuitive : certaines zones du visage (les yeux, la bouche, les cheveux, etc.) pourraient être plus « significatives » que d'autres.

2.2 Q5: Impact de l'ajout de variables de nuisance

Nous allons maintenant évaluer l'effet de l'ajout de variables non informatives (bruit gaussien) sur les performances du classifieur.

La procédure concrète est la suivante :

1. **`run_svm_cv(X, y)`** : Appliquer un `GridSearchCV` sur les caractéristiques faciales originales (ici, les pixels moyens en niveaux de gris de chaque visage, préalablement standardisés) afin d'établir une ligne de base sans bruit.
2. **Construire le bruit (`noise`)** : Générer 300 caractéristiques de bruit gaussien indépendant de forme `(n_samples, 300)` avec une variance de 1. Ensuite, concaténer ce bruit avec les caractéristiques originales pour créer `X_noisy` (la dimensionnalité augmente de 300, mais le nombre d'échantillons reste inchangé).
3. **`run_svm_cv(X_noisy, y)`** : Exécuter à nouveau le même processus SVM sur l'ensemble de données bruité.

```python
# Q5
from sklearn import svm
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)))

print("Score sans variable de nuisance")
# use run_svm_cv on original data
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, )
#with gaussian coefficients of std sigma
X_noisy = np.concatenate((X, noise), axis=1)
X_noisy = X_noisy[np.random.permutation(X.shape[0])]
# use run_svm_cv on noisy data
run_svm_cv(X_noisy, y)
```

```
Score sans variable de nuisance
Generalization score for linear kernel: 1.0, 0.9

Score avec variable de nuisance
Generalization score for linear kernel: 1.0, 0.5368421052631579
```

Analyse : Comme attendu, l'ajout de 300 variables de bruit aléatoire a considérablement dégradé les performances du classifieur.

Pour un nombre d'échantillons fixe, l'ajout massif de caractéristiques non informatives réduit la marge effective d'un SVM linéaire (une « séparabilité fallacieuse » pouvant apparaître dans les directions du bruit). Le modèle devient ainsi plus sensible aux perturbations aléatoires, et son erreur de généralisation augmente.

C'est également une manifestation intuitive du « fléau de la dimensionnalité » dans un cadre de haute dimension avec peu d'échantillons : plus le nombre de dimensions de bruit est élevé, plus il est difficile de trouver un hyperplan de séparation stable.

2.3 Q6: Amélioration via la réduction de dimension (PCA)

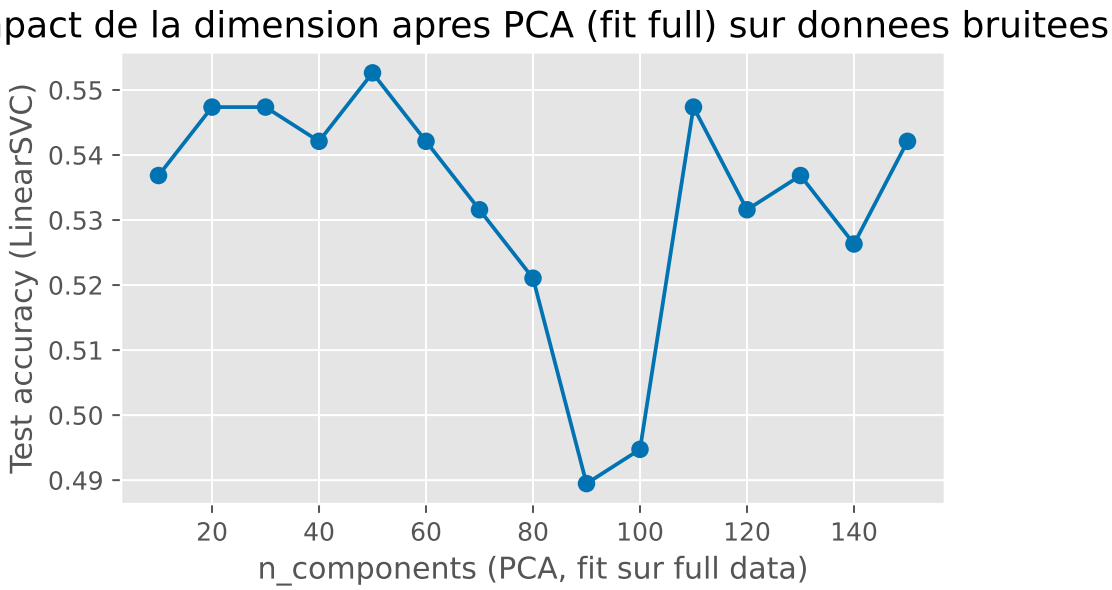
Pour contrer l'effet négatif des variables de nuisance, nous appliquons une Analyse en Composantes Principales (PCA) sur les données bruitées avant de les fournir au SVM. La PCA va identifier les axes de plus grande variance, qui correspondent (on l'espère) aux caractéristiques originales et non au bruit, nous permettant ainsi de "nettoyer" les données.

Afin d'améliorer l'efficacité de l'exécution, nous utilisons ici `LinearSVC` en remplacement de `SVC(kernel='linear')`. Bien que les résultats numériques puissent légèrement différer, cette substitution n'affecte de manière significative ni le principe sous-jacent, ni les conclusions tirées des résultats.

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

C_fixed = Cs[ind]
Xn_all = X_noisy
yn_all = y

# n_components from 10 to 150 with step 10; clip by data limits
max_nc = min(Xn_all.shape[0], Xn_all.shape[1])
grid_n_components = list(range(10, 151, 10))
grid_n_components = [k for k in grid_n_components if k <= max_nc]
if not grid_n_components:
    grid_n_components = [min(20, max_nc)]

# Fit PCA only once with the largest K later loop just slices the first k PCs
Kmax = max(grid_n_components)
pca = PCA(n_components=Kmax, svd_solver='randomized', iterated_power=1)


Z_all = pca.fit_transform(Xn_all)
test_scores = []
train_scores = []
best_records = []  # (k, C_fixed, train_score, test_score)
```

```python
for k in grid_n_components:
    # Take the first k principal components and restore original train/test split
    Ztr = Z_all[train_idx, :k]
    Zte = Z_all[test_idx, :k]
    ytr = yn_all[train_idx]
    yte = yn_all[test_idx]

    # Faster linear SVM: LinearSVC (reuse the best C from Q4)
    clf = LinearSVC(C=C_fixed, dual="auto", max_iter=5000)
    clf.fit(Ztr, ytr)

    tr = clf.score(Ztr, ytr)
    te = clf.score(Zte, yte)

    train_scores.append(tr)
    test_scores.append(te)
    best_records.append((k, C_fixed, tr, te))

# Print the best n_components and corresponding scores
best_idx = int(np.argmax(test_scores))
best_k, best_C, best_tr, best_te = best_records[best_idx]
print(f"Best n_components = {best_k}, best C = {best_C}")
print(f"Train score = {best_tr:.3f}, Test score = {best_te:.3f}")

# Visualization: test accuracy vs n_components
plt.figure()
plt.plot(grid_n_components, test_scores, marker='o')
plt.xlabel("n_components (PCA, fit sur full data)")
plt.ylabel("Test accuracy (LinearSVC)")
plt.title("Impact de la dimension apres PCA (fit full) sur donnees bruitees")
plt.tight_layout()
plt.show()
```

```
Score apres reduction de dimension
Best n_components = 50, best C = 0.0001
Train score = 0.732, Test score = 0.553
```



Analyse : En utilisant l'ACP pour réduire la dimensionnalité des données bruitées, nous pouvons observer qu'au début, la précision connaît une légère baisse. Cependant, à mesure que le nombre de `n_components` augmente, elle amorce une remontée. Un nombre optimal de composantes principales (autour de 100-150) permet de capturer l'essentiel de l'information utile tout en filtrant une grande partie du bruit. Cela démontre l'efficacité de l'ACP en tant que technique de pré-traitement pour les données de grande dimension.

2.4 Q7: Biais dans le prétraitement des données (Script Original)

La question 7 du TP demande d'identifier un biais dans le prétraitement des données du script `svm_script.py`. Ce biais est un problème classique de **fuite de données (data leakage)**.

Dans le script, pour la tâche sur les visages (LFW), la standardisation est effectuée sur l'ensemble des données *avant* la séparation en ensembles d'entraînement et de test :

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

Ce n'est qu'ensuite que les données sont divisées en deux moitiés (entraînement et test) après une permutation aléatoire. Cela signifie que la moyenne et l'écart-type ont été calculés en "voyant" les données de test. Il s'agit d'un cas classique de fuite de données: des informations de l'ensemble de test s'infiltrent prématurément dans le processus d'entraînement, ce qui conduit à des scores de test artificiellement élevés et à une évaluation trop optimiste.

3 Conclusion

Ce travail pratique nous a permis de mettre en œuvre et d'évaluer les SVM sur différentes tâches. Nous avons constaté l'importance du choix du noyau et des hyperparamètres, qui peuvent être optimisés par validation croisée. L'étude sur les données de visages a illustré de manière concrète l'impact du paramètre de régularisation `C` sur le compromis biais-variance. Enfin, nous avons démontré expérimentalement la malédiction de la dimensionnalité et l'efficacité de la PCA pour y remédier, tout en soulignant l'importance cruciale d'éviter la fuite de données lors du prétraitement pour obtenir une évaluation fiable des performances du modèle.