TP Apprentissage statistique SVM

Zakaria KHODRI

2024-09-30

Objectif du TP

Ce travail pratique (TP) a pour objectif de mettre en œuvre les techniques de classification supervisée à l'aide des **Support Vector Machines (SVM)** sur des données simulées et réelles. Les SVM sont des algorithmes de classification très puissants qui reposent sur la recherche de règles de décision linéaires, représentées par des hyperplans séparateurs dans un espace de grande dimension.

Les objectifs principaux de ce TP sont :

- Appliquer les SVM pour la classification binaire à l'aide de la librairie **scikit-learn**.
- Explorer l'impact des différents noyaux (linéaire, polynomial) et des hyperparamètres comme **C** et γ sur la performance du classifieur.
- Expérimenter les SVM dans des contextes de données déséquilibrées.
- Analyser les résultats et la généralisation du modèle.

Nous allons utiliser des jeux de données comme **Iris** et des visages **lfw (Labeled Faces in the Wild)** pour mettre en pratique les méthodes SVM et évaluer leur performance.

Question 1 : Classification des classes 1 et 2 du dataset Iris avec un noyau linéaire

L'objectif de cette première question est d'utiliser un **SVM** avec un **noyau linéaire** pour classifier les classes 1 (versicolor) et 2 (virginica) du dataset **Iris**, en se basant uniquement sur les deux premières variables du jeu de données : la **longueur des sépales** et la **largeur des sépales**.

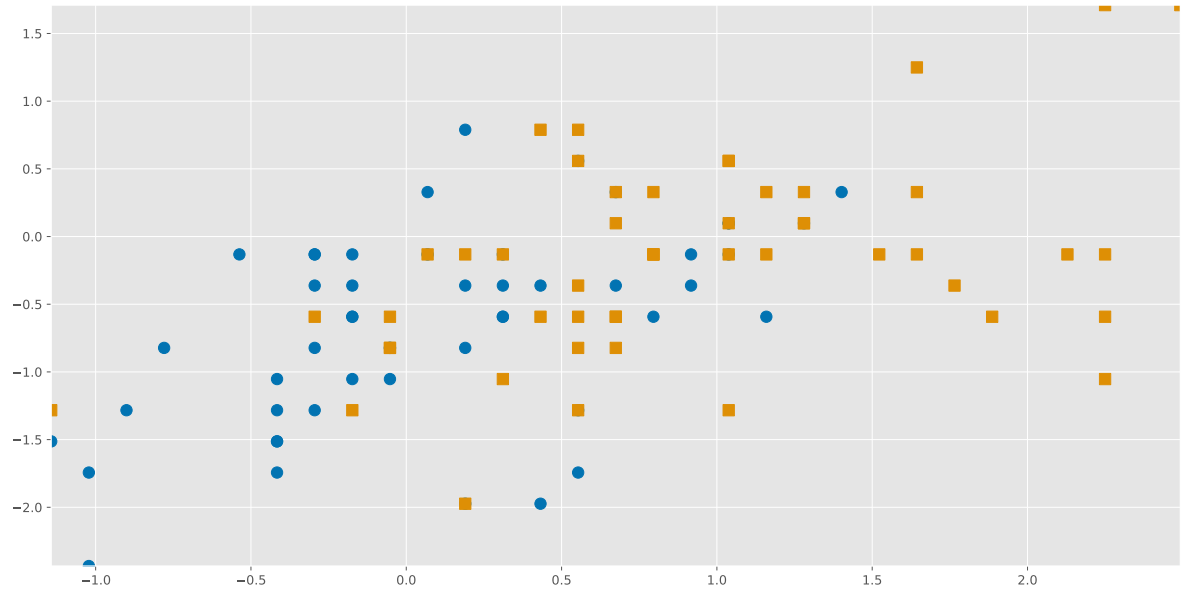
1. **Chargement et normalisation des données** : Nous commençons par charger le dataset Iris à l'aide de la librairie **scikit-learn**. Ensuite, nous standardisons les données pour s'assurer que les caractéristiques sont sur la même échelle. Cela est particulièrement important pour les SVM, car ils sont sensibles à la variance des données.

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	Plant
0	-0.900681	1.019004	-1.340227	-1.315444	0
1	-1.143017	-0.131979	-1.340227	-1.315444	0
2	-1.385353	0.328414	-1.397064	-1.315444	0
3	-1.506521	0.098217	-1.283389	-1.315444	0
4	-1.021849	1.249201	-1.340227	-1.315444	0
...	...	...	...	...	...
145	1.038005	-0.131979	0.819596	1.448832	2
146	0.553333	-1.282963	0.705921	0.922303	2
147	0.795669	-0.131979	0.819596	1.053935	2
148	0.432165	0.788808	0.933271	1.448832	2
149	0.068662	-0.131979	0.762758	0.790671	2

1. **Sélection des variables** : Nous sélectionnons les deux premières variables (longueur et largeur des sépales) et les classes cibles 1 (versicolor) et 2 (virginica), en excluant la classe 0 (setosa).

```
# Select the first two variables : sepal length (cm),sepal width (cm)
X = X[y != 0, :2]
# Select the the targets versicolor (1) and virginica (2)
y = y[y != 0]
```

3. **Visualisation des données** : Une représentation en 2D des données est affichée pour observer la distribution des classes dans l'espace des caractéristiques.



4. **Division en ensembles d'entraînement et de test** : Nous divisons les données en un ensemble d'entraînement (50 %) et un ensemble de test (50 %). Cela nous permet d'évaluer la capacité de généralisation du modèle.

```
# split train test
X, y = shuffle(X, y, random_state=18)
# the correspondence between each row in X and its label in y is maintained.
X_iris_train, X_iris_test, y_iris_train, y_iris_test = train_test_split(
    X, y, test_size=0.5, random_state=47
)
```

5. **Entraînement du SVM avec un noyau linéaire** : Le modèle est entraîné sur l'ensemble d'entraînement. Nous ajustons également l'hyperparamètre **C** à l'aide de **GridSearchCV** pour trouver la meilleure valeur de régularisation.

```
clf_linear = SVC(kernel="linear")
clf_linear.fit(X_iris_train, y_iris_train)
print("Score of a linear classifier without hyperparameter tuning is", clf_linear.score(X_
```

```
Score of a linear classifier without hyperparameter tuning is 0.74
```

```
Best param using GridSearchCV is SVC(C=0.46594447573960407, kernel='linear')
```

6. **Évaluation des performances** : Nous évaluons la performance du modèle en mesurant le score de classification (précision) et en affichant la matrice de confusion. Cela permet de vérifier la qualité de la séparation entre les classes.

```
print(
    "Generalization score for linear kernel: %s, %s"
    % (
        clf_linear.score(X_iris_train, y_iris_train),
        clf_linear.score(X_iris_test, y_iris_test),
    )
)
confusion_table(clf_linear.predict(X_iris_test), y_iris_test)
```

```
Generalization score for linear kernel: 0.68, 0.74
```

Truth Predicted	1	2
1	18	8
2	5	19

Après avoir entraîné le modèle avec un **noyau linéaire**, nous obtenons une précision satisfaisante sur l'ensemble de test. En utilisant **GridSearchCV**, nous avons optimisé le paramètre **C**, qui contrôle le compromis entre la maximisation de la marge et la minimisation des erreurs de classification. Nous avons ensuite observé **une amélioration pas trés significatives** des performances générales du modèle.

La matrice de confusion nous permet de visualiser les erreurs de classification et de vérifier si les deux classes sont correctement distinguées par le modèle.

Question 2 : Classification avec un noyau polynomial

Dans cette question, nous allons expérimenter un SVM avec un **noyau polynomial** pour la classification des classes 1 et 2 du jeu de données Iris. Contrairement au noyau linéaire, le noyau polynomial permet de séparer les classes dans des espaces de caractéristiques de plus haute dimension, ce qui peut améliorer la classification lorsque les données ne sont pas linéairement séparables.

1. **Entraînement du SVM avec un noyau polynomial** : Nous appliquons un SVM avec un noyau polynomial de différents degrés (1, 2 et 3) pour observer l'impact du degré du polynôme sur les performances du modèle.

```
Generalization score both train and test sets for polynomial kernel using : 0.6, 0.56
```

Truth Predicted	1	2
1	23	22
2	0	5

2. **Optimisation des hyperparamètres** : Nous effectuons une recherche de grille (**GridSearchCV**) pour optimiser les hyperparamètres **C**, **gamma**, et **degree** afin d'obtenir les meilleures performances.

```
SVC(degree=1, gamma=10.0, kernel='poly')
```

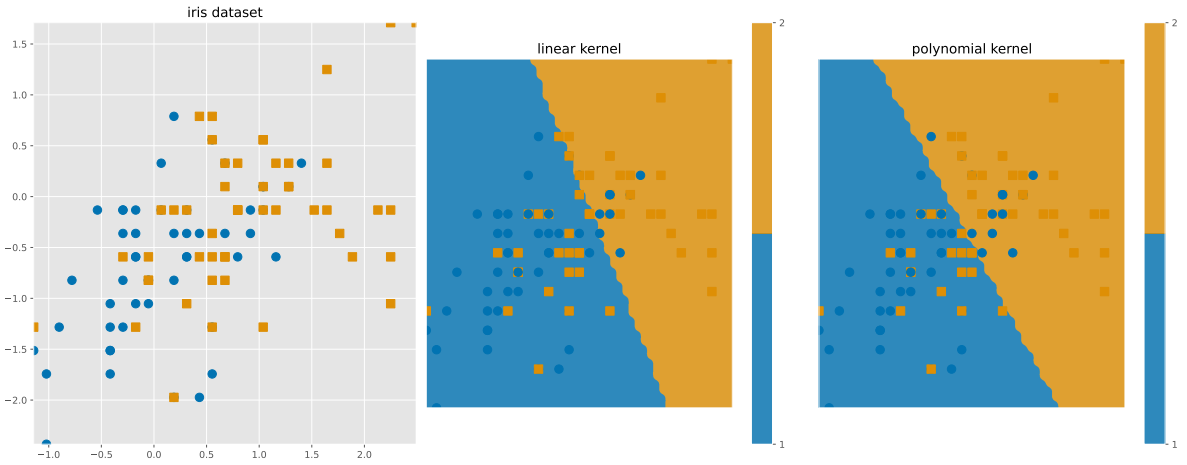
3. **Évaluation des performances** : Comme pour le noyau linéaire, nous mesurons la performance du modèle en termes de score de généralisation (précision) et affichons la matrice de confusion.

```
score for train and test sets using GridSearchCV using polynomial kernel: 0.64, 0.7
```

Truth Predicted	1	2
1	18	10
2	5	17

4. **Comparaison des résultats avec le noyau linéaire** : Nous traçons les frontières de décision du modèle linéaire et du modèle polynomial pour visualiser la différence dans la capacité de séparation des classes.

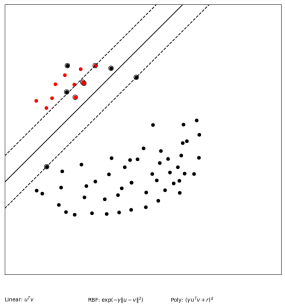
Le score du classifier avec noyau Lineaire donne un score plus élevé que noyau polynomial, donc on le préfere mieux que classifier avec noyau polynomial.



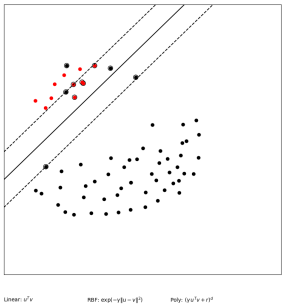
Question 3 : SVM GUI

On utilise le script **svm_gui.py**:

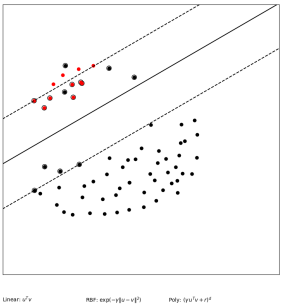
- Ce graphique nous permet d'examiner l'impact du choix du paramètre de régularisation C sur la performance du modèle SVM. Comme illustré dans l'exemple testé (voir figure 6), nous observons qu'une valeur de C plus petite entraîne des marges plus larges. Cela signifie que le modèle est plus tolérant aux erreurs de classification, permettant ainsi à certains points d'être mal classés sans pénaliser excessivement le classifieur. En revanche, lorsque C est élevé, le modèle cherche à minimiser les erreurs, ce qui peut conduire à des marges plus étroites et à un surajustement.



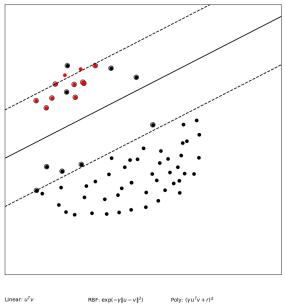
(a) C = 1



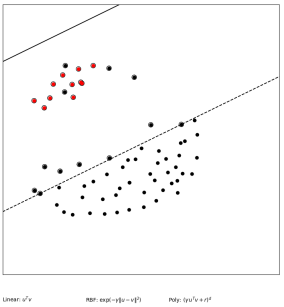
(b) C = 0.01



(c) C = 0.001



(d) C = 0.0005



(e) C = 0.0001

- Dans la dernière figure, tous les points rouges sont mal classés, ce qui soulève des préoccupations quant à l'équilibre des classes dans les données. En effet, ce déséquilibre entre le nombre de points noirs et de points rouges peut influencer la performance du modèle. Les points rouges, ayant un poids négligeable par rapport aux points rouges, sont sous-représentés et, par conséquent, le modèle peut ne pas être en mesure d'apprendre correctement à les classer. Cela souligne l'importance de considérer la distribution des classes lors de l'entraînement des modèles SVM, ainsi que l'utilisation de techniques comme la pondération des classes ou la génération de données synthétiques pour améliorer la représentation des classes minoritaires.

Question 4 : Classification de visages avec un noyau linéaire

Dans cette question, nous allons utiliser un **SVM avec un noyau linéaire** pour effectuer une classification binaire sur un jeu de données de visages. Le but est de différencier deux personnalités, **Donald Rumsfeld** et **Colin Powell**, à partir des images de la base de données **Labeled Faces in the Wild (LFW)**.

Les étapes de l'expérimentation sont les suivantes :

1. **Chargement et exploration des données** : Nous utilisons la fonction `fetch_lfw_people` de **scikit-learn** pour télécharger et charger le jeu de données LFW, en ne sélectionnant que les images des deux personnalités à classifier.

2. **Extraction des caractéristiques** : Pour simplifier la tâche de classification, nous extrayons une seule caractéristique par image : la **luminosité moyenne**. Cela réduit la dimensionnalité des données tout en conservant une information pertinente pour la tâche.

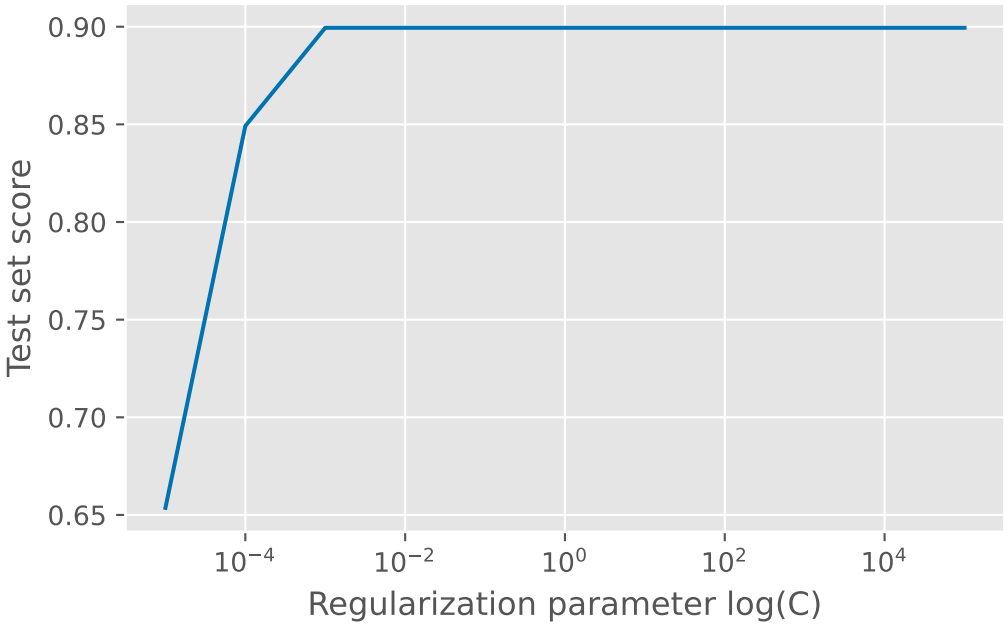
```
X_img = (np.mean(images, axis=3)).reshape(n_samples, -1)
```

3. **Prétraitement des données** : Les caractéristiques extraites sont standardisées afin que les SVM puissent traiter des données correctement mises à l'échelle.
4. **Division en ensembles d'entraînement et de test** : Comme dans les questions précédentes, nous divisons les données en un ensemble d'entraînement et un ensemble de test.

```
# Split data into a half training and half test set
X_img_train, X_img_test, y_train, y_test, images_train, images_test = train_test_split(
    X_img, y, images, test_size=0.5, random_state=0
)
```

5. **Entraînement du SVM avec un noyau linéaire** : Nous entraînons un SVM avec un noyau linéaire et ajustons le paramètre de régularisation **C** pour maximiser la précision de la classification.

```
--- Linear kernel ---
Fitting the classifier to the training set
Best C: 0.001
Best score: 0.8994413407821229
Predicting the people names on the testing set
```



6. **Évaluation des performances** : Nous utilisons la matrice de confusion et mesurons la précision pour évaluer la qualité des prédictions sur l'ensemble de test.

```
clf_img = SVC(kernel="linear", C=best_C)
clf_img.fit(X_img_train, y_train)
y_predicted = clf_img.predict(X_img_test)

print("done in %0.3fs" % (time() - t0))
# The chance level is the accuracy that will be reached when constantly predicting the maj
print("Chance level : %s" % max(np.mean(y), 1.0 - np.mean(y)))
print("Accuracy : %s" % clf_img.score(X_img_test, y_test))
confusion_table(y_predicted, y_test)
```

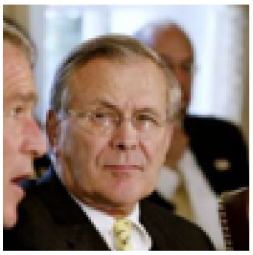
```
done in 0.132s
Chance level : 0.6610644257703081
Accuracy : 0.8994413407821229
```

Truth Predicted	0	1
0	47	3
1	15	114

7. **Visualisation des résultats** : Nous visualisons les coefficients du classifieur sous forme d'image pour interpréter les caractéristiques apprises par le modèle.

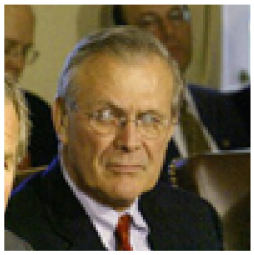
predicted: Powell
true: Powell

predicted: Rumsfeld
true: Rumsfeld



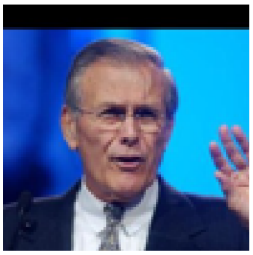
predicted: Powell
true: Powell

predicted: Powell
true: Rumsfeld

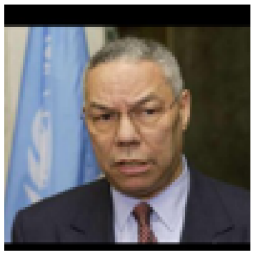


predicted: Powell
true: Powell

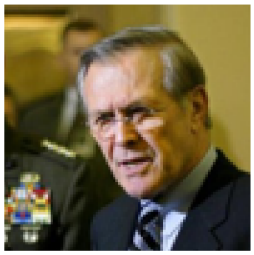
predicted: Rumsfeld
true: Rumsfeld



predicted: Powell
true: Powell



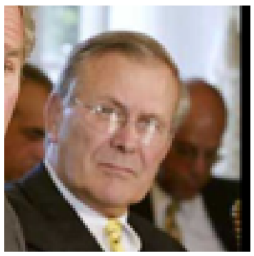
predicted: Rumsfeld
true: Rumsfeld



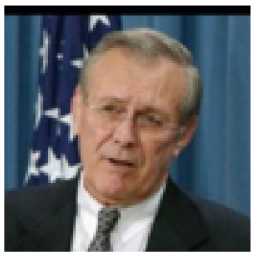
predicted: Powell
true: Powell

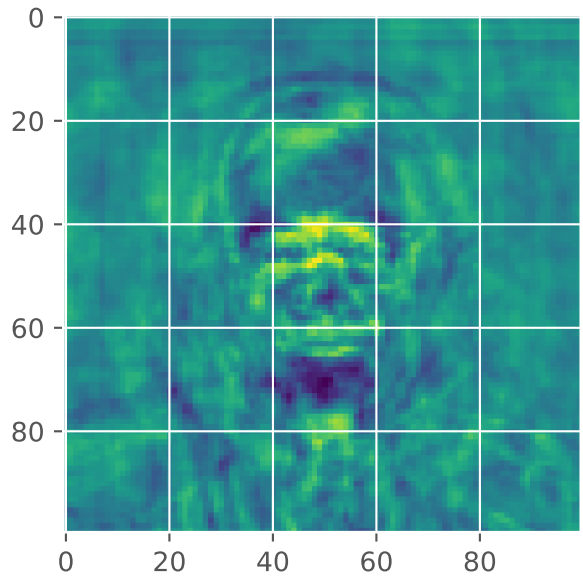
predicted: Rumsfeld
true: Rumsfeld

predicted: Rumsfeld
true: Rumsfeld



predicted: Rumsfeld
true: Rumsfeld





Après avoir entraîné le **SVM avec un noyau linéaire**, nous avons optimisé le paramètre de régularisation **C** pour obtenir une précision maximale. La **précision du modèle** sur l'ensemble de test est satisfaisante et dépasse le niveau de chance. La **matrice de confusion** confirme que le modèle distingue correctement les deux personnalités dans la majorité des cas.

La visualisation des coefficients du modèle nous donne un aperçu des zones des visages qui influencent le plus la décision du classifieur. Ces coefficients, affichés sous forme d'image, permettent de comprendre quelles parties des visages le modèle utilise pour différencier **Donald Rumsfeld** de **Colin Powell** qui peuvent apparaitre comme la bouche et les sourcils ainsi que les yeux .

Question 5 : Impact des variables de nuisance sur la classification de visages

Dans cette question, nous étudions l'impact des **variables de nuisance** sur la performance d'un modèle SVM utilisant un noyau linéaire. Les variables de nuisance sont des caractéristiques ajoutées aux données d'origine qui n'ont pas de relation directe avec la tâche de classification, et leur ajout permet de tester la robustesse du modèle.

Les étapes sont les suivantes :

1. **Entraînement du modèle sans variables de nuisance** : Nous commençons par entraîner un SVM sur les données d'origine, sans aucune variable ajoutée. Cela nous permet d'établir une référence de performance.

```
print("Score sans variables de nuisance")
clf_cv = SVM_CV()
clf_cv.run_svm_cv(X_img, y)
```

```
Score sans variables de nuisance
Generalization score for linear kernel: 1.0, 0.8770949720670391
```

2. **Ajout de variables de nuisance** : Nous ajoutons des variables aléatoires à chaque image (300 nouvelles caractéristiques), ce qui introduit du bruit dans les données. Nous observons ensuite l'impact de ce bruit sur la performance du modèle.
3. **Évaluation des performances avant et après l'ajout des variables de nuisance** : Nous comparons les scores de généralisation du modèle sur les ensembles d'entraînement et de test avant et après l'ajout des variables de nuisance.

```
Score avec variables de nuisance
Generalization score for linear kernel: 1.0, 0.5754189944134078
```

4. **Visualisation des images** : Nous affichons les images d'origine et les images bruitées côte à côte pour visualiser l'effet des variables de nuisance.

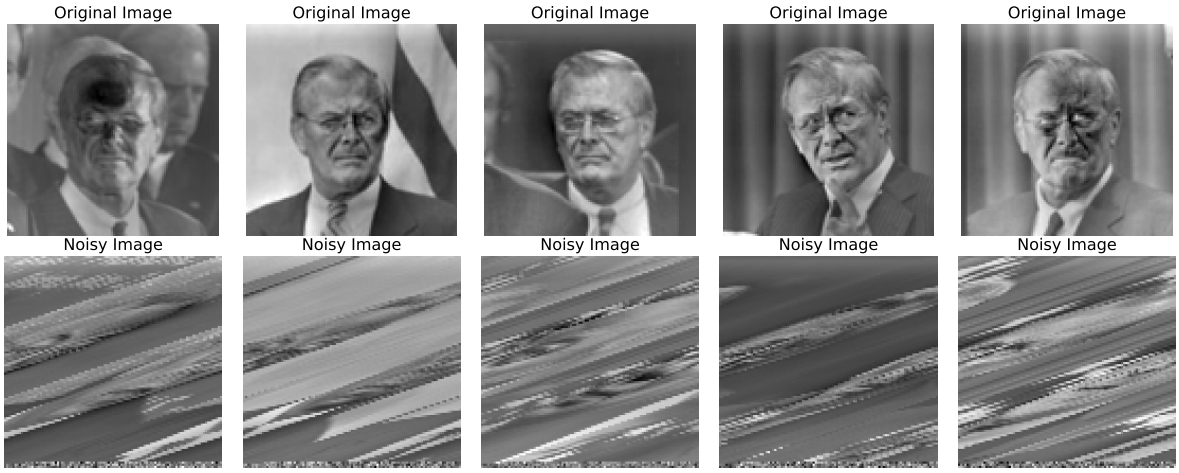
```
def plot_images(original_images, noisy_images, n_images=5):
    fig, axes = plt.subplots(2, n_images, figsize=(15, 6))

    for i in range(n_images):
        # Original images
        axes[0, i].imshow(original_images[i], cmap="gray")
        axes[0, i].set_title("Original Image")
        axes[0, i].axis("off")

        # Noisy images
        axes[1, i].imshow(noisy_images[i], cmap="gray")
        axes[1, i].set_title("Noisy Image")
        axes[1, i].axis("off")

    plt.tight_layout()
    plt.show()

plot_images(X_img_reshaped, X_noisy_reshaped)
```



Analyse des résultats

Après avoir effectué les deux entraînements (avec et sans variables de nuisance), nous observons que l'ajout de variables de nuisance réduit généralement les performances du modèle (**le score passe de 87% à 54%**). Cela est dû au fait que ces variables aléatoires n'apportent aucune information utile pour la tâche de classification, mais ajoutent du bruit, ce qui complique l'apprentissage du modèle.

La visualisation des images originales et bruitées permet de constater l'impact des variables de nuisance sur les données. Les images bruitées contiennent des informations supplémentaires qui ne sont pas pertinentes pour la classification des visages, ce qui entraîne une dégradation des performances du SVM.

Question 6 : Réduction de dimension avec PCA

Dans cette question, nous allons appliquer l'**analyse en composantes principales (PCA)** pour réduire la dimension des données tout en conservant une part significative de la variance. Cela peut améliorer les performances du modèle SVM en éliminant le bruit et les variables non pertinentes.

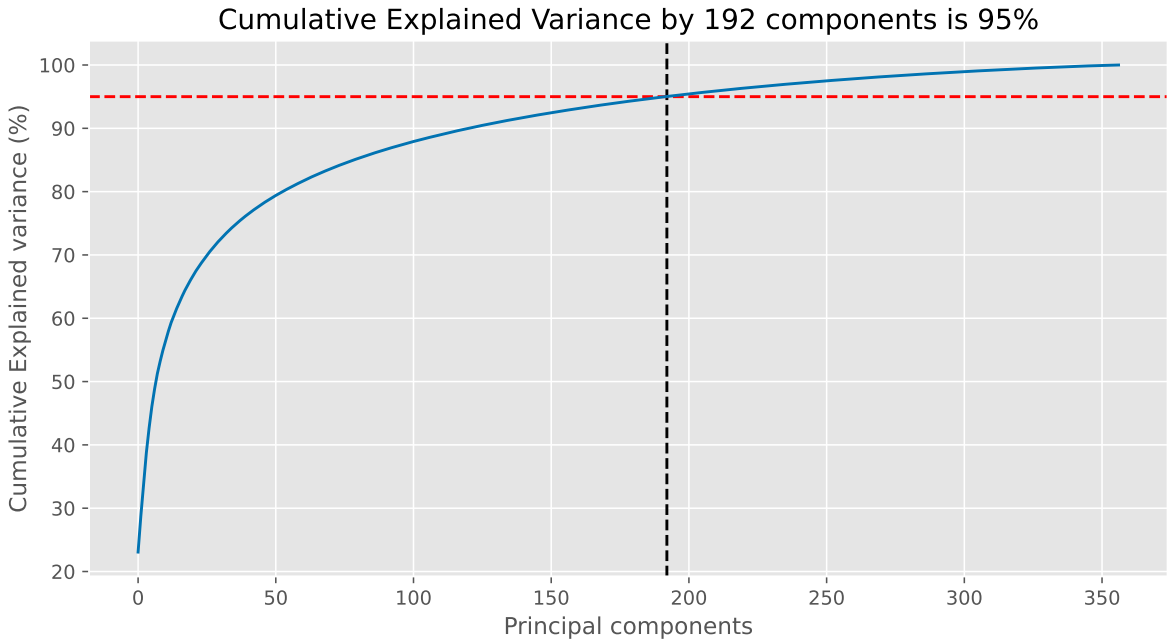
Les étapes de cette analyse sont les suivantes :

1. **Calcul de la variance expliquée cumulée** : Nous effectuons une PCA sur les données bruitées pour déterminer combien de composantes principales sont nécessaires pour expliquer 95 % de la variance. Cela nous aide à choisir le nombre optimal de composantes à conserver. **Visualisation de la variance expliquée** : Nous traçons un graphique montrant la variance cumulée expliquée par chaque composante principale.

```
pca = PCA(svd_solver="randomized", random_state=12).fit(X_noisy)
var_cumu = np.cumsum(pca.explained_variance_ratio_) * 100


def num_components_exp_var(exp_var):
    var_cumu = np.cumsum(pca.explained_variance_ratio_) * 100
    # How many PCs explain exp_var % of the variance?
    k = np.argmax(var_cumu > exp_var)
    return k


def dim_reduction(exp_var):
    plt.figure(figsize=[10, 5])
    plt.title(
        f"Cumulative Explained Variance by {num_components_exp_var(exp_var)} components is
    )
    plt.ylabel("Cumulative Explained variance (%)")
    plt.xlabel("Principal components")
    plt.axvline(x=num_components_exp_var(exp_var), color="k", linestyle="--")
    plt.axhline(y=exp_var, color="r", linestyle="--")
    ax = plt.plot(var_cumu)
    return plt.show()
dim_reduction(95)
```



2. **Réduction de dimension** : En utilisant le nombre optimal de composantes trouvées, nous transformons les données bruitées en un espace de dimension réduite de **192**.

```
n_components = num_components_exp_var(95)
X_noisy_stand = scaler.fit_transform(X_noisy)
pca2 = PCA(n_components=n_components, svd_solver="randomized").fit(X_noisy_stand)
X_pca = pca2.transform(X_noisy_stand)
```

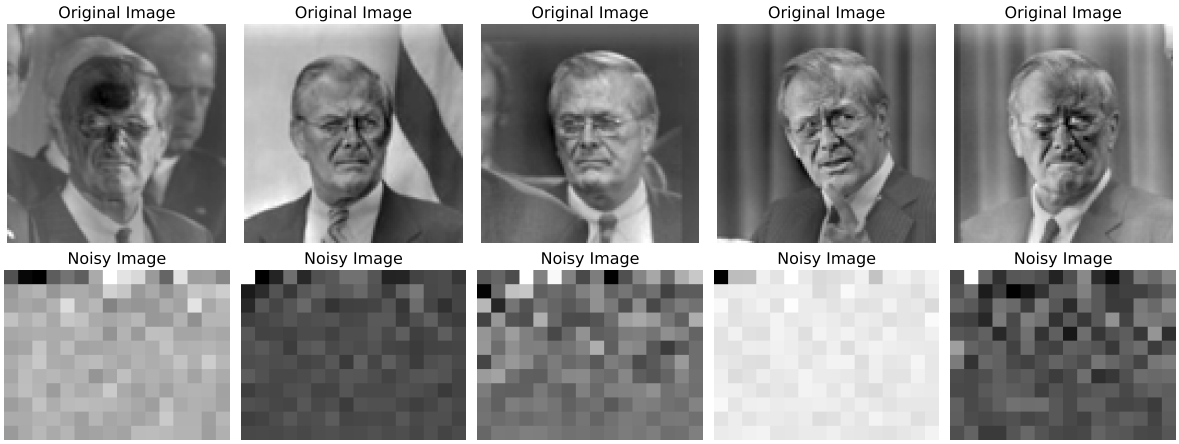
3. **Évaluation de la performance du SVM après réduction de dimension** : Nous entraînons un SVM avec les données réduites et mesurons sa performance.

```
print("Score after dimensionality reduction")
clf_cv.run_svm_cv(X_pca, y)
```

```
Score after dimensionality reduction
Generalization score for linear kernel: 1.0, 0.547486033519553
```

4. **Visualisation des images images compressées**: Nous affichons les images avant et après la réduction de dimension pour illustrer la transformation.

```
plot_images(X_img_reshaped, X_pca.reshape(357, 12, 16), 5)
```



5. **Variation du score avec la variation des nombres des composantes** La fonction plot_pca_vs_score évalue l'impact du nombre de composantes principales sur les performances d'un modèle SVM. Elle standardise d'abord les données, puis applique la PCA avec différents nombres de composantes, en entraînant et en évaluant le modèle pour chaque configuration. Enfin, elle trace un graphique illustrant la relation entre le nombre de composantes et le score SVM, avec des barres d'erreur représentant l'écart type des scores.

```
def plot_pca_vs_score(X, y, start_comp, end_comp, step, num_runs):
    """
    Function to plot the SVM scores as a function of the number of PCA components.

    Parameters:
    - X: Input features (before PCA)
    - y: Labels
    - start_comp: The starting number of components to test
    - end_comp: The ending number of components to test
    - step: Step size for the range of PCA components
    - num_runs: Number of times to run SVM for each component to observe score variations
    """
    scores_avg = []
    scores_std = []
    num_components_list = list(
        range(start_comp, end_comp, step)
    )  # Range of components to test

    # Standardize the data
    X_stand = scaler.fit_transform(X)
```

```
    for n_components in num_components_list:
        run_scores = []

        for run in range(num_runs):
            # Apply PCA with n_components
            pca = PCA(
                n_components=n_components, svd_solver="randomized", random_state=17
            )
            X_pca = pca.fit_transform(X_stand)

            print(f"Running SVM with {n_components} components (Run {run + 1})...")
            # Run SVM on the transformed data and capture the score
            clf_cv.run_svm_cv(X_pca, y)
            score = clf_cv.score["test_score"]
            run_scores.append(score)

        # Compute the average and standard deviation of scores across the runs
        avg_score = sum(run_scores) / len(run_scores)
        std_score = np.std(run_scores)

        scores_avg.append(avg_score)
        scores_std.append(std_score)

    # Plot the number of components vs. the SVM score (with error bars for variations)
    plt.figure(figsize=(10, 6))
    plt.errorbar(
        num_components_list, scores_avg, yerr=scores_std, fmt="-o", color="b", capsize=5
    )
    plt.xlabel("Number of PCA Components")
    plt.ylabel("SVM Score")
    plt.title(f"SVM Score vs. Number of PCA Components (Averaged over {num_runs} runs)")
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    return scores_avg, scores_std
```

```
# plot_pca_vs_score(X_noisy, y=y, start_comp=80, end_comp=190, step=1, num_runs=100)
```

Ici on teste le modèle SVM avec un nombre de composantes allant de 80 à 190, en réalisant 100 exécutions pour chaque configuration afin de capturer la variabilité des scores. Enfin, elle trace

un graphique illustrant la relation entre le nombre de composantes et le score SVM, facilitant ainsi l'optimisation du modèle pour améliorer ses performances.

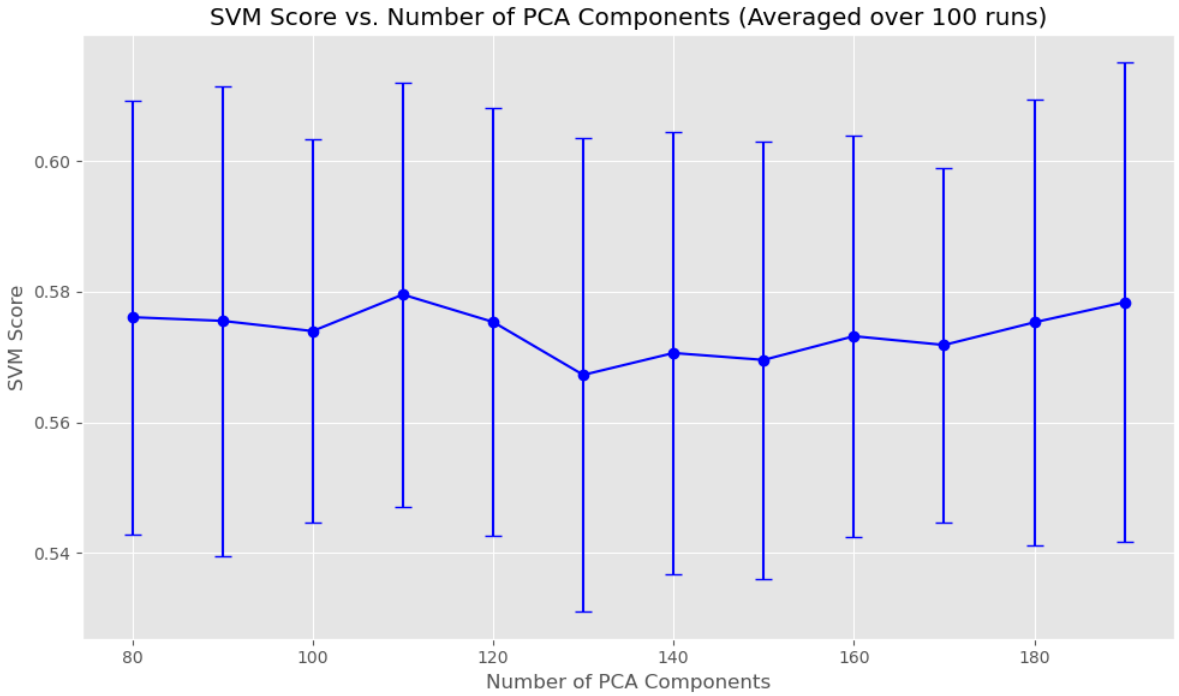


Figure 1: Score du modèle en changeant les nombre des composantes

On peut constater que le modèle aprés la réduction de dimension a un score pas mal qui est plus ou moins mieux que le score avant la réduction de dimension en moyenne de **57%**, donc le modèle n'arrive pas a trouver un signal dans les images bruitées à cause de la destructions des correspondance lorsqu'on a ajouter du bruit et la permutation des pixels dans les images.