Rapport : Machine Learning

Compte-rendu du projet d'étude

Machine Learning en Python

MAM 3
Année 2024 - 2025

Polytech Nice Sophia

Rédigé par :
Elsa Catteau, Wissal El Figuigui et Charlotte Prouzet

Sommaire

1 Introduction

2 Transformation des données

2.1 Analyse et préparation des données

Nous avons débuté ce projet en récupérant le jeu de données `Digits` à l'aide de la fonction `load_digits` de scikit-learn. Cette fonction renvoie deux tableaux : `X_digits` qui contient les données d'entrée et `y_digits` qui contient les étiquettes cibles.
Nous avons donc effectué de premiers petits codes comme :

Code Python

```
def nombre_aleatoire():  # pour afficher un nombre aleatoire
    nombre = random.randint(0, 9)  # on genere un nombre aleatoire compris
        entre 0 et 9
    print("L'image generee correspond au nombre", nombre)

    digits = load_digits()
    plt.matshow(digits.images[nombre], cmap="gray")
    plt.axis('off')
    plt.show()
```

Cela nous permet d'afficher un nombre aléatoire entre 0 et 9. En effet, la commande digits.image[i] permet d'afficher le nombre i sous la forme d'une image 8x8.

On peut par la suite afficher les 4 premières images de notre data base en écrivant :

Code Python

```
def nombre_aleatoire_i(nombre):  # pour afficher un nombre i
    print("L'image generee correspond au nombre", nombre)
    digits = load_digits()
    plt.matshow(digits.images[nombre], cmap="gray")
    plt.axis('off')
    plt.show()

for i in range(4):
    nombre_aleatoire_i(i)
```

Cela script affiche les résultats suivants :

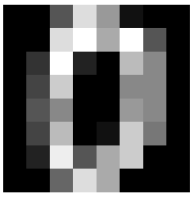


Figure 1: Visualisation de la 1ère image de notre data base : 0

Figure 2: Visualisation de la 2ème image de notre data base : 1

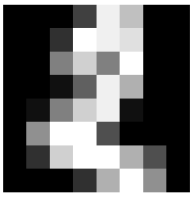


Figure 3: Visualisation de la 3ème image de notre data base : 2

Figure 4: Visualisation de la 4ème image de notre data base 3

Nous avons ensuite affiché les 16 premiers digits afin d'avoir une idée visuelle de leur représentation à l'aide du code suivant :

Code Python

```
def plot_multi(data, y):
    nplots = 16
    nb_classes = len(np.unique(y))
    cur_class = 0
    fig = plt.figure(figsize=(15,15))
    for j in range(nplots):
        plt.subplot(4, 4, j + 1)
        to_display_idx = np.random.choice(np.where(y == cur_class)[0])
        plt.imshow(data[to_display_idx].reshape((8, 8)), cmap='binary')
        plt.title(cur_class)
        plt.axis('off')
        cur_class = (cur_class + 1) % nb_classes
    plt.show()

plot_multi(digits.data, digits.target)
```

On obtient l'affichage :

Figure 5: Visualisation des 15 premiers nombres

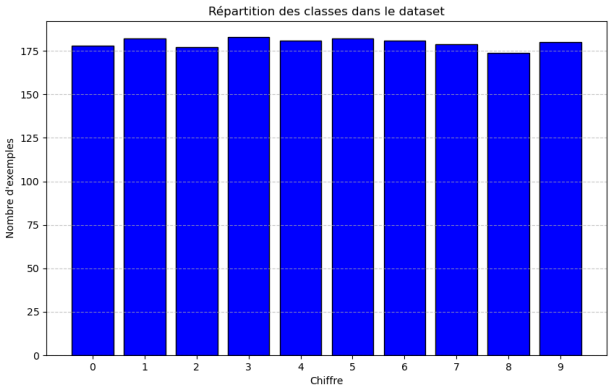
Par la suite, nous avons décidé d'explorer plus en détail notre base de données en :

- **Ajoutant des informations sur la dimensionnalité des données :**
  Chaque donnée est représentée par un vecteur de 300 valeurs, obtenu grâce à un modèle d'embedding préentraîné. Ainsi, l'ensemble de la base peut être vu comme une matrice de taille $(n \times 300)$, où n est le nombre total d'exemples.
- **Étudiant la distribution des classes via la fonction `get_statistics_text` :**

Code Python

```
def get_statistics_text(targets):
    labels, counts = np.unique(targets, return_counts=True)
    return labels, counts

labels, counts = get_statistics_text(digits.target)
plt.figure(figsize=(10, 6))
plt.bar(labels, counts, color='blue', edgecolor='black')
plt.xlabel("Chiffre")
plt.ylabel("Nombre d'exemples")
plt.title("Repartition des classes dans le dataset")
plt.xticks(labels)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

Ce code nous renvoie le graphe suivant :



Ce diagramme en bâtons montre que toutes les classes sont plus ou moins également réparties. On observe que chacune d'entre elles apparaît entre 174 et 187 fois dans notre base de données.

Figure 6: Distribution des différentes classes de notre base de données

2.2 Visualisation et préparation des données pour le modèle

A présent, nous allons chercher à visualiser non plus des données uniques (comme l'image d'un chiffre), mais plutôt la base de données dans son ensemble.

Code Python

```
X = digits.data
y = digits.target

print(f"Feature matrix shape: {X.shape}. Max value = {np.max(X)}, Min value
    = {np.min(X)}, Mean value = {np.mean(X)}")
print(f"Labels shape: {y.shape}")

# On pose F notre matrice X normalisee sur l'intervalle [0,1]
F = X / 16.0
print(f"Feature matrix F shape: {F.shape}. Max value = {np.max(F)}, Min
    value = {np.min(F)}, Mean value = {np.mean(F)}")
```

Nous allons donc poser :

- X : une "feature matrix" telle que X=digits.data

- y : un vecteur d'étiquettes tel que y=digits.target

On peut afficher plus en détails des informations sur notre matrice X, telles que :

- X est une matrice 1797x64
- La valeur maximale est : 16.0
- La valeur minimale est : 0.0
- La valeur moyenne est : 4.885

On peut ensuite normaliser nos données en divisant notre matrice X par 16.0 (valeur maximale). On pose donc F notre matrice après normalisation telle que :

- F est une matrice 1797x64
- La valeur maximale est : 1.0
- La valeur minimale est : 0.0
- La valeur moyenne est : 0.305

2.3 Réduction de dimension avec une ACP (PCA) pour la visualisation

2.3.1 Principe

L'analyse en Composantes Principales est une méthode de réduction de la dimensionnalité qui améliore la performance des algorithmes de machine learning en éliminant les variables corrélées qui ne contribuent à aucune prise de décision.
Notre objectif sera donc de prendre les données de notre espace de grande dimension (8x8 = 64 dimensions), et de les remplacer par d'autres dans un espace de dimension inférieure (2 ou 3 dimensions), mais qui contiennent encore la plupart des renseignements présents dans l'ensemble initial.
Autrement dit, on cherche à construire le moins de variables possible, tout en conservant le maximum d'informations.

2.3.2 Génération du modèle ACP

Nous avons donc généré un modèle ACP avec différentes valeurs de composantes principales. Le but est alors de reconstruire un vecteur de dimension 8x8=64 à partir de l'espace ACP de dimension réduite, puis de transformer l'approximation obtenue en une matrice 8x8. Nous avons également jugé pertinent de noter, pour chaque nouveau tracé, l'erreur d'approximation associée.
le code que nous avons rédigé est le suivant :

Code Python

```
sample_index = 9
original_vector = F[sample_index]  # notre vecteur original en 64 dimensions
original_image = F[sample_index].reshape(8, 8)  # matrice 8 8  pour la
    visualisation

fig, axes = plt.subplots(4, 4, figsize=(10, 10))
fig.suptitle('Image originale et approximation par ACP', fontsize=16)

for i in range(16):
    ax = axes[i // 4, i % 4]

    if i == 0:  # on affiche l'image originale
        ax.imshow(original_image, cmap='binary')
        ax.set_title("Original")
        ax.axis('off')
    else:
        # On applique PCA avec i composantes
        pca = PCA(n_components=i)
        F_reduced = pca.fit_transform(F)
        F_approx = pca.inverse_transform(F_reduced)

        # On r cup re la reconstruction du vecteur original
        reconstructed_vector = F_approx[sample_index]
        reconstructed_image = reconstructed_vector.reshape(8, 8)

        # On calcule l'erreur (MSE)
        error = np.mean((original_vector - reconstructed_vector) ** 2)

        # On affiche l'image reconstruite
        ax.imshow(reconstructed_image, cmap='binary')
        ax.set_title(f"{i} comps\nErr={error:.4f}")
        ax.axis('off')

plt.tight_layout()
plt.subplots_adjust(top=0.90)
plt.show()
```

En essayant de représenter les nombres 3; 5 et 9, nous obtenons les résultats suivants :

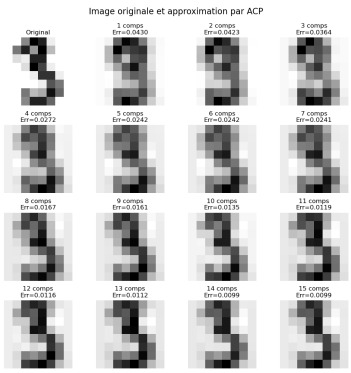


Figure 7: Visualisation du nombre 3

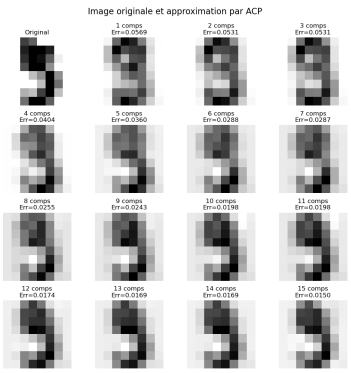


Figure 8: Visualisation du nombre 5

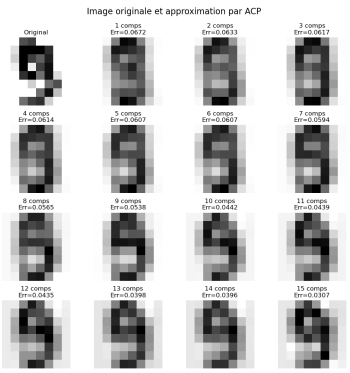


Figure 9: Visualisation du nombre 9

On remarque qu'à partir d'environ 10 composantes, la représentation semble correcte (du moins, elle ne

va pas nettement s'améliorer pour un nombre supérieur de composantes).
Cependant, on observe aussi que notre représentation graphique contient tout de même des limites puisque la représentation du chiffre 5 par exemple n'est pas tout à fait optimale. Il est en effet difficile, même pour un nombre de composantes maximal, de deviner que c'est bien le chiffre 5 qui est représenté.

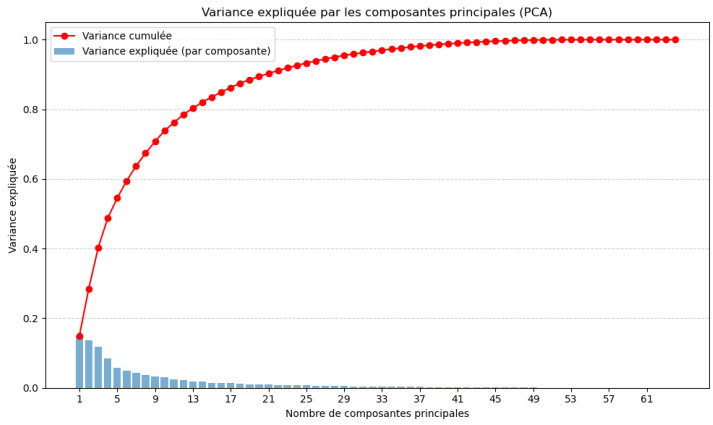
2.3.3 Analyse de la variance

Afin d'appliquer l'ACP, nous utilisons la fonction `PCA` de scikit-learn.
Dans un premier temps, on récupère la variance expliquée ainsi que la variance cumulée pour chaque composante. Puis on les trace via le code suivant :

Code Python

```
pca = PCA()
pca.fit(F)

# On recupere la variance expliquee pour chaque composante
explained_variance = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance)

# On trace la variance expliquee et la variance cumulee
plt.figure(figsize=(10, 6))
plt.bar(range(1, len(explained_variance) + 1), explained_variance,
        alpha=0.6, label="Variance expliquee (par composante)")
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance,
         color='red', marker='o', label="Variance cumulee")
plt.xlabel("Nombre de composantes principales")
plt.ylabel("Variance expliquee")
plt.title("Variance expliquee par les composantes principales (PCA)")
plt.xticks(np.arange(1, 65, step=4))
plt.grid(axis='y', linestyle='--', alpha=0.6)
plt.legend()
plt.tight_layout()
plt.show()
```

Nous obtenons le graphe suivant :



On observe ici que les premières barres de la variance expliquée sont plus hautes que les suivantes. Cela signifie que les premières composantes expliquent une grande partie de la variance.
D'autre part, la courbe rouge (variance cumulée) nous montre à partir de combien de composantes est ce que l'on peut capturer 95% de la variance (par exemple). Ici en l'occurrence, on peut suggérer que ce sera à partir de 30 composantes que l'on aura capturé environ 95% de la variance.

Figure 10: représentation graphique de la variance expliquée et de la variance cumulée

Afin de vérifier ces résultats graphiques, on peut écrire la fonction suivante qui nous permet de connaitre précisément le nombre de composantes, qui, une fois dépassé, nous indique que l'on a atteint plus de

95% de capture de la variance :

Code Python

```
n_components_95 = np.argmax(cumulative_variance >= 0.95) + 1
print(f"Nombre de composantes pour expliquer au moins 95\% de la variance :
    {n_components_95}")
```

Ce script renvoie : "Nombre de composantes pour expliquer au moins 95% de la variance : 29", ce qui est conforme aux prédictions faites via la visualisation du graphique.
Dans notre code "optimal", nous choisirons donc d'utiliser 29 composantes.

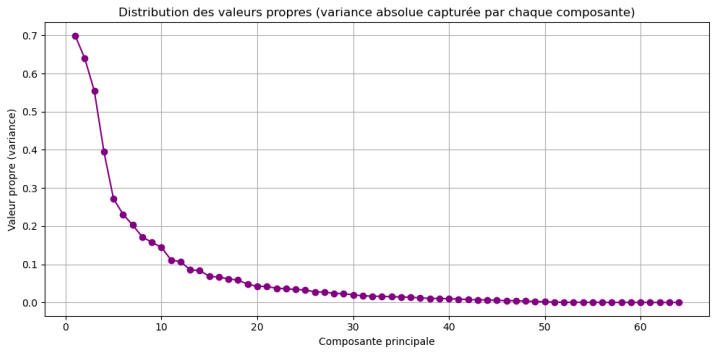
2.3.4 Analyse de la distribution des valeurs propres

De manière à aller plus loin dans notre analyse, nous pouvons chercher à analyser la distribution des valeurs propres. Cela revient à étudier la variance expliquée brute. Nous utilisons le code suivant :

Code Python

```
eigenvalues = pca.explained_variance_  # on recupere les valeurs propres (la
    variance expliquee brute)

plt.figure(figsize=(10, 5))
plt.plot(range(1, len(eigenvalues) + 1), eigenvalues, marker='o', linestyle=
    '-', color='purple')
plt.title("Distribution des valeurs propres (variance absolue capturee par
    chaque composante)")
plt.xlabel("Composante principale")
plt.ylabel("Valeur propre (variance)")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Nous obtenons le graphe :



On observe ici une forte décroissance au début, ce qui indique que les premières composantes capturent la majorité de la variance. Cela est cohérent avec les résultats énoncés précédemment.
Par ailleurs, on remarque que les dernières valeurs propres tendent vers 0, ce qui signifie que certaines composantes n'apportent aucune information, et qu'elles peuvent donc être ignorées sans perte significative.
Enfin, les valeurs propres sont assez faibles ($< 0{,}7$), ce qui confirme que les images des chiffres manuscrits sont fortement redondantes

Figure 11: représentation graphique de la variance expliquée et de la variance cumulée

2.3.5 Visualisation en 2D

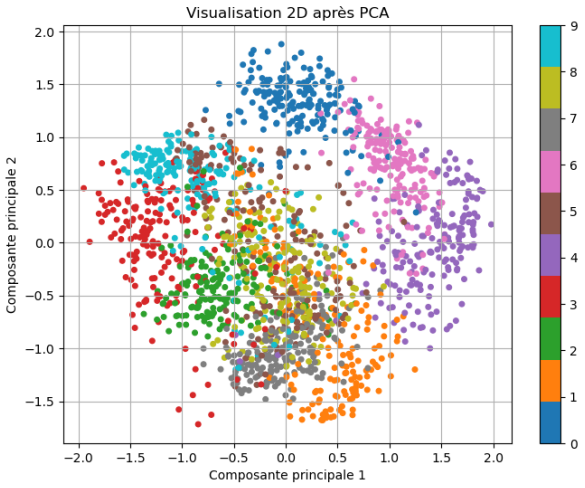
On peut compléter nos visualisations en ajoutant une ACP à 2 dimensions et en affichant l'ensemble des données avec une couleur par label. Cela peut nous permettre de visualiser les clusters/classes dans

l'espace des 2 premières composantes principales.
Le code utilisé est :

Code Python

```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(F)  # F est ta matrice normalis e (entre 0 et 1)
plt.figure(figsize=(8,6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='tab10', s=15)
plt.colorbar()
plt.title("Visualisation 2D apres PCA")
plt.xlabel("Composante principale 1")
plt.ylabel("Composante principale 2")
plt.grid(True)
plt.show()
```

Nous obtenons la représentation 2D suivante :



On observe ici plusieurs regroupements de couleur. Cela indique bien que dans le cas du chiffre 0 par exemple que toutes ses représentations possèdent globalement les mêmes caractéristiques, et qu'elles sont donc similaires entres elles.
En revanche, dans le cas de la couleur marron qui représente le chiffre 5, on remarque que les points sont assez dispersés les uns des autres, ce qui conforte l'idée que la représentation du chiffre 5 est assez vaste : d'un point de vue "dispersion de points", et d'un point de vue "affichage graphique du chiffre lui-même". Cela rejoint l'observation dans la partie "Génération du modèle ACP" dans laquelle nous avions observé que la représentation graphique du chiffre 5 pouvait aussi s'apparenter à celle d'un 9 par exemple.

Figure 12: Visualisation en 2D de nos données

2.3.6 Visualisation en 3D

Nous pouvons aussi représenter nos données en 3D afin de visualiser davantage d'éventuelles proximités entre nos points de couleur.
Pour ce faire, nous avons rédigé le code suivant :

Code Python

```
pca = PCA(n_components=3)
X_pca_3d = pca.fit_transform(F)
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(X_pca_3d[:, 0], X_pca_3d[:, 1], X_pca_3d[:, 2], c=y,
    cmap='tab10', s=20, alpha=0.8)

ax.set_title("Visualisation 3D apres PCA")
ax.set_xlabel("Composante 1")
ax.set_ylabel("Composante 2")
ax.set_zlabel("Composante 3")

cbar = fig.colorbar(scatter, ax=ax, shrink=0.6)
cbar.set_label('Classe')

plt.tight_layout()
plt.show()
```

2.3.7 Création d'une matrice caractéristique à 20 dimensions

On peut aussi créer une matrice de caractéristiques réduite à 20 dimensions en réutilisant l'analyse en composantes principales. Cela nous permet de réduire la dimension des données d'origine en extrayant les 20 composantes principales. On compresse donc l'information tout en conservant l'essentiel de la variance.

Remarque :
On choisit 20 dimensions car c'est un nombre à la fois suffisant pour conserver la majorité de la variance, et assez petit pour réduire fortement le nombre de dimensions.

Le code utilisé est donc le suivant :

Code Python

```
pca_20 = PCA(n_components=20)
F_pca = pca_20.fit_transform(F)  # F est la matrice normalisee (entre 0 et
    1)
print(f"Feature matrix F_pca shape: {F_pca.shape}")
```

Notre programme renvoie bien : `Feature matrix F_pca shape:(1797, 20)`.

Pour la matrice caractéristique initiale à 64 dimensions, nous avions : `Feature matrix F_pca shape: (1797, 64)`.
Le nombre de colonnes a donc bien été réduit, ce qui confirme le bon fonctionnement de la réduction de dimension.

2.4 Extraction des caractéristiques

Afin d'améliorer la représentation des images, et donc la performance de nos modèles d'apprentissage, nous avons décidé de procéder à une extraction multicritères des caractéristiques.

Dans un premier temps, nous appliquons une extraction zonale. Ce procédé consiste à diviser chaque image en trois zones horizontales (haut, milieu et bas), puis de calculer la moyenne des intensités dans chaque zone. Cette étape nous permet de capturer des informations sur chaque zone de l'image. Nous pourrons ainsi mieux reconnaitre les chiffres en regardant les différentes zones de notre image :

Code Python

```
def extract_zone_features(images):
    n = images.shape[0]
    mean = []
    for i in range(n):
        img = images[i].reshape(8, 8)
        zone1 = img[0:3, :]   # zone sup rieure
        zone2 = img[3:5, :]   # zone centrale
        zone3 = img[5:8, :]   # zone inferieure
        # Calcul de la moyenne des intensit s de chaque zone
        mean_zone1 = np.mean(zone1)
        mean_zone2 = np.mean(zone2)
        mean_zone3 = np.mean(zone3)
        mean.append([mean_zone1, mean_zone2, mean_zone3])
    return np.array(mean)

F_zones = extract_zone_features(F)
print(f"Feature matrix F_zones shape: {F_zones.shape}")
```

Une fois la division de l'image en trois zones, on trouve que la forme de la matrice créée `F_zones` (1797, 3). On a donc bien 3 dimensions.

Ensuite, nous effectuons une extraction basée sur la détection de contours à l'aide du filtre de Sobel. Ce filtre permet de repérer les changements brusques d'intensité qui correspondent aux bords. Pour chaque image, on calcule la moyenne de ces contours, ce qui nous donne une nouvelle information liée à la forme et à la texture du chiffre.
Remarque : Cette étape est très importante car les contours sont essentiels pour reconnaitre la forme et la structure des chiffres.
Nous avons donc rédigé le code suivant :

Code Python

```
sobel_image = sobel(original_image)

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(original_image, cmap='RdPu')
plt.colorbar()
plt.title("Image originale")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(sobel_image, cmap='RdPu')
plt.title("Avec le filtre de Sobel")
plt.colorbar()
plt.axis('off')

plt.tight_layout()
plt.show()

n = F.shape[0]
F_edges = np.zeros((n, 1))
for i in range(n):
    F_edges[i] = np.mean(sobel(F[i].reshape(8, 8)))

print(f"Feature matrix F_edges shape: {F_edges.shape}")
```

Ce script nous affiche 2 images : l'une correspondant à l'image originale et l'autre à cette même image une fois que l'on applique le filtre de Sobel. D'un point de vue visuel, le filtre met en valeur les contours du chiffre, c'est-à-dire là où l'intensité change le plus.

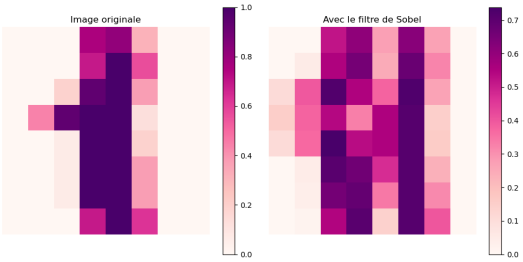


Figure 13: Image originale du chiffre 1 VS image du chiffre 1 avec le filtre de Sobel

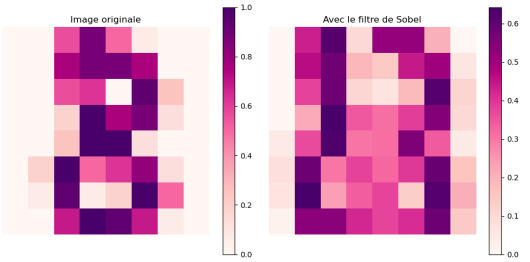


Figure 14: Image originale du chiffre 8 VS image du chiffre 8 avec le filtre de Sobel

Lorsque l'on choisit le chiffre 1 (figure 13), l'image de Sobel affiche nettement un trait vertical au centre de la figure, ce qui est cohérent avec la forme que l'on connait du chiffre 1.
Si l'on choisit le chiffre 8 (figure 14), l'image de Sobel met en évidence deux boucles, ce qui est bien représentatif du chiffre 8.
Une fois l'extraction basée sur la détection de contours effectuée, on trouve que la forme de notre matrice caractéristique est (1797, 1). On a donc bien 1 dimension.

Par ailleurs, pour chaque application du filtre de Sobel sur une image, nous avons calculé la moyenne de l'intensité des contours et l'avons stockée dans une nouvelle matrice `F_edges`, de taille (1797, 1).

A ce stade là, nous avons donc les 3 matrices suivantes :

- `F_pca` de taille (1797,20)
- `F_zones` de taille (1797,3)
- `F_edges` de taille (1797,1)

Nous combinons donc nos 3 caractéristiques : PCA, zonales, et de contours en une seule matrice. Cela va nous permettre d'enrichir la représentation des données et d'améliorer la capacité de notre modèle à distinguer les classes.
Pour ce faire, nous avons écrit le script suivant :

Code Python

```
F_final = np.hstack([F_pca, F_zones, F_edges])
```

Enfin, nous avons appliqué une normalisation (entre 0 et 1) de nos caractéristiques à l'aide du scaler "MinMax". Cette normalisation est essentielle car elle garantit que toutes les caractéristiques ont une échelle comparable :

Code Python

```
scaler = MinMaxScaler()
F_final = scaler.fit_transform(F_final)
```

Finalement, la matrice obtenue a une taille de (1797,24). Chaque ligne représente une image et chaque colonne correspond à l'intensité d'un pixel dans une grille 8x8. La matrice finale contient donc toutes les informations visuelles des chiffres manuscrits sous forme vectorielle, une forme pratique pour appliquer des analyses statistiques, ou encore pour entrainer des modèles de machine learning.

3 Classification des données

L'objectif de cette nouvelle étape est de prédire la catégorie à laquelle appartient une nouvelle observation, et ce, en étudiant au préalable des données étiquetées. Nous allons donc entrainer notre modèle, puis étudier ensuite comment ce dernier se comporte-t-il en phase de "test" face à de nouvelles données qui n'ont pas encore été étudiées.

3.1 Division des données pour l'apprentissage

Afin d'améliorer la généralisation de notre modèle, nous procédons à une division "split" de nos données. Cela nous permettra de tester notre modèle sur des données inédites, et d'éviter le sur-apprentissage (overfitting).

Nous avons séparé nos données de la manière suivante : 80% pour la phase d'entrainement, et 20% pour la phase de test.
Le code utilisé est le suivant :

Code Python

```
X = digits.data
y = digits.target

k = 80 / 100  # proportion de donnees pour l'entrainement

# On calcule l'indice de decoupage
split_X = int(k * X.shape[0])  # 80% des lignes de X
split_Y = int(k * y.shape[0])  # 80% des lignes de y

# On separe les donnees
X_train, X_test = X[0:split_X, :], X[split_X:, :]
y_train, y_test = y[0:split_Y], y[split_Y:]
```

Ce code nous permet d'obtenir les classes de tailles suivantes :

- Total samples: 1797
- Training set: 1437 samples
- Testing set : 360 samples

Par la suite, nous avons décidé de tracer un diagramme en bâtons qui nous permet de visualiser la répartition des classes dans 3 ensembles de données :

- Les données originales (`y_original`)
- Les données d'entraînement (`y_train`)
- Les données de test (`y_test`)

Nous avons tracé 2 graphes, l'un avec les données brutes, et l'autre en normalisant nos données (entre 0 et 1) :

Code Python

```
def plot_normalized_distribution(y_original, y_train, y_test):
    labels, counts_orig = get_statistics_text(y_original)
    _, counts_train = get_statistics_text(y_train)
    _, counts_test = get_statistics_text(y_test)

    # Normalisation par le total
    total_orig = len(y_original)
    total_train = len(y_train)
    total_test = len(y_test)

    counts_orig_norm = counts_orig / total_orig
    counts_train_norm = counts_train / total_train
    counts_test_norm = counts_test / total_test

    x = np.arange(len(labels))
    width = 0.25

    plt.figure(figsize=(10, 6))
    plt.bar(x - width, counts_orig_norm, width, label='Original', color='
        pink')
    plt.bar(x, counts_train_norm, width, label='Train', color='skyblue')
    plt.bar(x + width, counts_test_norm, width, label='Test', color='
        lightgreen')

    plt.xlabel("Chiffre")
    plt.ylabel("Proportion (entre 0 et 1)")
    plt.title("Distribution des classes normalisee (par rapport au total)")
    plt.xticks(x, labels)
    plt.legend()
    plt.grid(axis='y', linestyle='--', alpha=0.7)
    plt.tight_layout()
    plt.show()

plot_normalized_distribution(y, y_train, y_test)
```

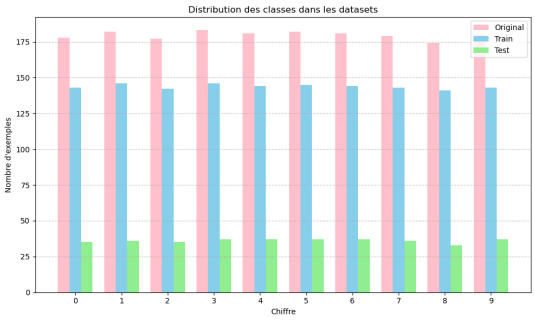


Figure 15: Répartition des classes de notre dataset

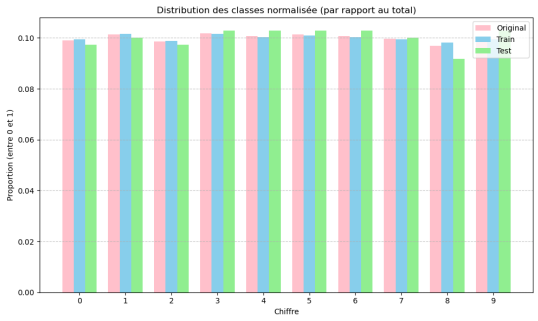


Figure 16: Répartition normalisée des classes de notre dataset

Sur la figure normalisée, on observe que toutes les barres ont à peu près la même hauteur pour une classe donnée (environ 0.1). Cela signifie que la distribution des classes est équilibrée dans les trois ensembles (original, train, test). Notre représentation est donc cohérente.

Remarque :
Pour ces représentations, nous avons décidé de poser k=80%, avec k la proportion des données utilisée

pour l'entrainement. Mais comment évoluent les distributions lorsque l'on fait varier k ? Et quel serait le k "optimal à choisir ?
Concernant la phase d'apprentissage, on peut dire que plus notre k est grand, et plus l'on donne de données au modèle pour apprendre. Cela améliore donc la qualité de l'apprentissage. A contrario, une valeur plus petite de k laisse moins de données pour l'entrainement, ce qui peut réduire la capacité du modèle à "bien apprendre", mais qui offre une évaluation plus robuste.
Concernant la phase d'évaluation, on peut dire que plus notre k est grand et plus notre testset sera petit, et pourra donc donner une évaluation peu fiable. Au contraire, plus notre k est petit et plus notre testset sera grand, ce qui pourra pénaliser inutilement la performance du modèle (car pas assez de données pour s'entrainer correctement).

On en déduit donc que choisir un k=80% est un bon compromis entre entrainement et évaluation. C'est donc la proportion que nous allons garder pour la suite.

3.2 Méthodes de classification avec SVM

A présent, nous allons chercher à classifier les images de chiffres manuscrits. Nous allons utiliser un pipeline qui nous permettra d'améliorer la précision en exploitant à la fois des descripteurs comme le gradient de Sobel ou encore l'intensité des zones, et des méthodes statistiques (PCA).

Dans un premier temps, on extrait le gradient moyen d'une image à l'aide du filtre de Sobel. Ce dernier nous permet de mesurer les bords et les contours.
On transforme donc chaque image 8x8 en image de gradients, puis on retourne le gradient moyen. Cette étape nous permet de capturer les zones de changement rapide :

Code Python

```
class EdgeInfoPreprocessing(BaseEstimator, TransformerMixin):

    def __init__(self):
        pass

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        features = []
        for img in X:
            img_2d = img.reshape((8, 8))
            sobel_img = sobel(img_2d)

            mean_gradient = np.mean(sobel_img)
            features.append([mean_gradient])

        return np.array(features)
```

Ensuite, on extrait l'intensité moyenne des 3 zones que nous avions décrites dans la partie "Transformation des données", à savoir la partie haute, le milieu, et la partie basse. Cette partie-là nous permet d'obtenir la moyenne d'intensité propre à chaque zone. Cela revient à détecter si une partie de l'image est plus sombre ou claire :

Code Python

```python
class ZonalInfoPreprocessing(BaseEstimator, TransformerMixin):

    def __init__(self):
        pass

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        features = []
        for img in X:
            img_2d = img.reshape((8, 8))
            top = img_2d[0:3, :]
            middle = img_2d[3:5, :]
            bottom = img_2d[5:8, :]
            means = [np.mean(top), np.mean(middle), np.mean(bottom)]
            features.append(means)
        return np.array(features)
```

Nous appliquons en suivant une analyse en composantes principales (ACP) dans le but de réduire la dimension à 20 composantes principales. L'objectif est alors de compresser au maximum l'information, tout en conservant la variance la plus significative possible :

Code Python

```python
class PCAFeatureExtractor(BaseEstimator, TransformerMixin):

    def __init__(self, n_components=20):
        self.n_components = n_components
        self.pca = PCA(n_components=self.n_components)

    def fit(self, X, y=None):
        self.pca.fit(X)
        return self

    def transform(self, X):
        return self.pca.transform(X)
```

On combine nos 3 transformateurs (présentés précédemment) afin de générer une matrice finale à 24 dimensions :

- 20 dimensions pour la PCA.
- 3 dimensions pour la zone d'intensité moyenne.
- et 1 dimension pour le gradient moyen obtenu avec Sobel.

Code Python

```python
all_features = FeatureUnion([
    ('pca', PCAFeatureExtractor(n_components=20)),
    ('zonal', ZonalInfoPreprocessing()),
    ('edge', EdgeInfoPreprocessing())
])
```

On crée ensuite notre propre pipeline "clf", avec un classifieur SVC linéaire comme prédiction.

Code Python

```
clf = Pipeline([
    ('features', all_features),
    ('scaler', StandardScaler()),
    ('classifier', SVC(kernel='linear'))
])
```

Enfin, on lance une phase d'apprentissage et de test en suivant les étapes suivantes :

1. On entraîne le pipeline avec `clf.fit(X_train, y_train)`.
2. On évalue la précision avec `clf.score(X_test, y_test)`.
3. Puis on affiche le nombre total de caractéristiques extraites.

Nous avons donc écrit le code suivant :

Code Python

```
clf.fit(X_train, y_train)
print("Accuracy:", clf.score(X_test, y_test))

F = all_features.fit(X_train, y_train).transform(X_train)
print("Nb features computed: ", F.shape[1])
```

"Accuracy" correspond à la précision globale de notre pipeline "clf". Ici, le code nous indique que notre modèle prédit correctement 93.3% des chiffres sur le jeu de test. Ce résultat nous indique que notre pipeline (avec PCA + features manuelles + scaling + linearSCv) est efficace.

3.3 Premier entrainement d'un SVC

Dans cette partie nous allons entrainer, puis évaluer et analyser les performances d'un modèle de classification SVM (via le pipeline clf) appliqué à notre jeu de données.

On commence par entrainer notre modèle, puis effectuer des prédictions et les évaluer via le code suivant :

Code Python

```
clf.fit(X_train, y_train)

predict_test = clf.predict(X_test)
predict_train = clf.predict(X_train)

print("Accuracy of the SVC on the test set: ", clf.score(X_test, y_test))
print("Accuracy of the SVC on the train set: ", clf.score(X_train, y_train))
```

On effectue ainsi une analyse graphique en dessinant les matrices de confusion pour le modèle d'entrainement et le modèle test. Ces matrices nous permettent de visualiser combien d'images de chaque chiffre ont été

bien ou mal classées, ainsi que les erreurs spécifiques du modèle.

Code Python

```
# 1. Print of matrix
conf_mat_test = confusion_matrix(y_test, predict_test)
conf_mat_train = confusion_matrix(y_train, predict_train)

print("\nConfusion matrix (test):\n", conf_mat_test)
print("\nConfusion matrix (train):\n", conf_mat_train)

# 2. Figure of matrix
fig1, ax1 = plt.subplots()
disp_test = ConfusionMatrixDisplay(confusion_matrix=conf_mat_train)
disp_test.plot(ax=ax1, cmap='PuRd')
ax1.set_title("Matrice de confusion - Train set")

fig2, ax2 = plt.subplots()
disp_train = ConfusionMatrixDisplay(confusion_matrix=conf_mat_test)
disp_train.plot(ax=ax2, cmap='PuRd')
ax2.set_title("Matrice de confusion - Test set")

plt.show()
```

Less matrices obtenues sont les suivantes :

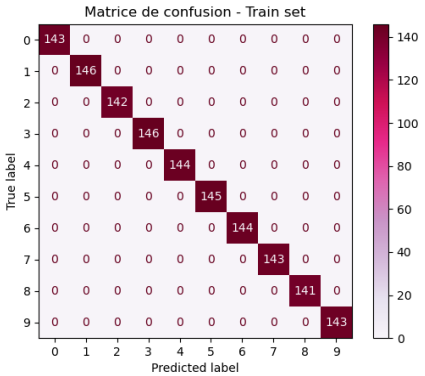


Figure 17: Matrice de confusion / Train set

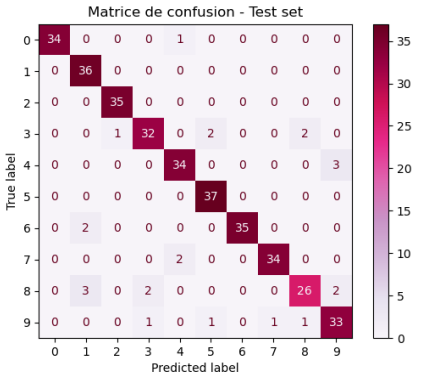


Figure 18: Matrice de confusion / Test set

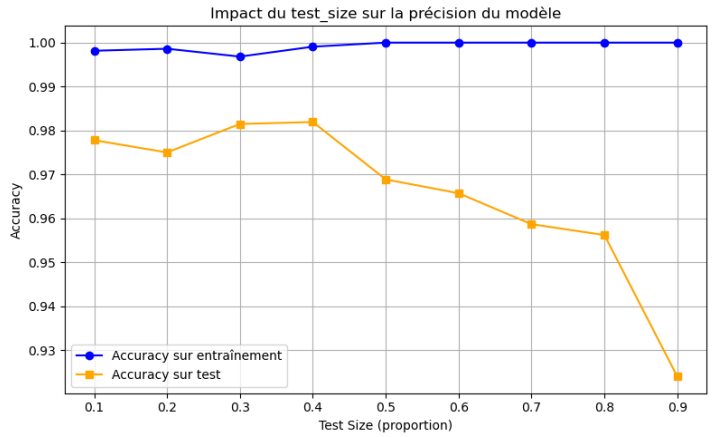
On observe que les valeurs sur la diagonale de la matrice de confusion du set d'entrainement sont très élevées. Cela signifie que notre modèle d'entrainement fonctionne correctement, comme espéré.
Quant à la matrice de confusion du set de test, on observe que certaines classes sont confondues. Par exemple, le 4 peut être confondu avec un 9, le 8 avec un 1 ou un 3 etc...

Enfin, nous avons décidé de nous pencher sur l'influence de la taille vis à vis de la précision de notre modèle. Nous avons donc rédigé le code suivant dans lequel nous avons tester différentes proportions de tests (de 10% à 90%) pour observer à la fois comment la taille de notre jeu de test influence-t-elle la performance, et s'il y a ou non un overfitting/underfitting :

Code Python

```
test_sizes = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
train_scores = []
test_scores = []

for test_size in test_sizes:
    # Split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=
        test_size, random_state=42)

    # Fit
    clf.fit(X_train, y_train)

    # Accuracy
    acc_train = clf.score(X_train, y_train)
    acc_test = clf.score(X_test, y_test)

    train_scores.append(acc_train)
    test_scores.append(acc_test)

    print(f"Test size: {test_size:.1f} | Accuracy train: {acc_train:.3f} |
        Accuracy test: {acc_test:.3f}")

# Plot
plt.figure(figsize=(8, 5))
plt.plot(test_sizes, train_scores, label='Accuracy sur entrainement', marker
    ='o', color='blue')
plt.plot(test_sizes, test_scores, label='Accuracy sur test', marker='s',
    color='orange')
plt.xlabel('Test Size (proportion)')
plt.ylabel('Accuracy')
plt.title('Impact du test_size sur la precision du mod le')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Nous obtenons ainsi le graphe :



On observe ici une forte diminution de la précision globale (courbe orange) sur le jeu de test lorsque la taille de ce dernier dépasse 40 %. En revanche, la précision du modèle sur le jeu d'entraînement reste relativement stable, ce qui peut indiquer un début de surapprentissage.

Figure 19: Évolution de la précision globale sur le jeu de test et sur le jeu d'entraînement en fonction de la proportion `test_size`.

L'analyse de ce graphique nous confirme que l'utilisation de 20 % des données pour les tests est un

choix pertinent. Ce résultat est cohérent avec l'affectation complémentaire de 80 % des données à l'entraînement, ce qui nous assure ainsi un bon équilibre entre apprentissage et évaluation du modèle.

Cependant, on observe aussi que pour une valeur de `test_size` égale à 0,4, la précision globale de notre modèle est maximale. En effet, elle atteint 98,2 % pour la phase de test et 100 % pour la phase d'entraînement. C'est pourquoi, dans notre modèle optimal, nous utiliserons les proportions suivantes :

- 60 % des données dédiées à l'entraînement
- 40 % des données réservées aux tests

Remarque : L'affectation de 80 % des données à l'entraînement et 20 % aux tests reste néanmoins valable, car elle assure également un bon équilibre entre apprentissage et évaluation du modèle.

3.4 Hyperparamètres et validation croisée (CV)

Afin d'améliorer au maximum les performances de notre modèle sur le jeu de données, nous allons optimiser notre pipeline de classification SVM (avec noyau RBF), et ce, en ajustant les hyperparamètres via validation croisée et GridSearchCV.

Pour commencer, on crée un nouveau pipeline en changeant le classifieur linéaire par un SVM à noyau RBF. Nous définissons ensuite des hyperparamètres à optimiser. En l'occurrence, pour ce pipeline, nous choisissons d'optimiser les paramètres :

- c : qui correspond au coefficient de régularisation du SVM
- gamma : qui correspond au paramètre du noyau RBF. Il peut être "scale", "auto", ou numérique (compris entre 0.001 et 1).

Notre code est donc le suivant :

Code Python

```
clf = Pipeline([
    ('features', all_features),
    ('scaler', StandardScaler()), # normalisation des features
    ('classifier', SVC(kernel='rbf'))
])

param_grid = {
    'classifier__C': [0.01, 0.1, 1, 10, 100, 1000],
    'classifier__gamma': ['scale', 'auto', 0.001, 0.01, 0.1, 1, 10]
}
```

Ensuite, nous recherchons les paramètres optimaux avec GridSearchCV, afin par la suite, d'évaluer le "meilleur" modèle.

Code Python

```
grid_search = GridSearchCV(estimator=clf,param_grid=param_grid,cv=5,n_jobs
    =1,verbose=10,scoring='accuracy',refit=True )
grid_search.fit(X_train, y_train)

# 1. Check the results
print("Meilleurs param tres trouv s: ")
print(grid_search.best_params_)
best_params = grid_search.best_params_

C_opt = best_params.get('classifier__C', 1)
gamma_opt = best_params.get('classifier__gamma', 'scale')

# 2. Update the original pipeline (or create a new one) with all the
    optimized hyper parameters
optimized_clf = Pipeline([('features',
    FeatureUnion([('pca', PCAFeatureExtractor(n_components=29)),
    ('sobel', EdgeInfoPreprocessing()),
    ('zones', ZonalInfoPreprocessing())])),
    ('scaler', StandardScaler()),
    ('classifier', SVC(kernel='rbf', C=C_opt, gamma=gamma_opt))])

# 3. Retrain on the whole train set, and evaluate on the test set
optimized_clf.fit(X_train, y_train)
test_accuracy = optimized_clf.score(X_test, y_test)
print(f"Pr cision sur le test set avec les meilleurs param tres: {
    test_accuracy:.4f}")


# 4. Answer the questions below and report on your findings
print(" K-Fold Cross-Validation Results:")
print(f"- Best Cross-validation score: {grid_search.best_score_}")
print(f"- Best parameters found: {grid_search.best_estimator_}")
```

Ce code prend en compte le paramètre "cv" qui contrôle le nombre de plis (K) dans notre validation croisée. Quel est donc l'impact de K sur nos données ?
En passant K de 5 à 10, on modifie la façon dont on découpe les données d'entraînement. Ainsi, augmenter notre nombre de plis rend l'estimation de la performance plus stable, puisque l'on a plus d'évaluations intermédiaires. Comme on entraine plus de fois notre modèle (10 fois au lieu de 5 par exemple), on augmente aussi le temps de calcul.
Sur un petit dataset, on privilégiera le choix K=10, tandis que pour un dataset plus grand, on prendra K=5.

On va donc lancer notre code pour deux valeurs différentes de K : 5 et 10 afin d'observer les différences qui en découlent :

1. Dans le cas ou $K = 10$:
   On remarque que pour une validation croisée qui découpe les données en 10 sous-groupes (plis), il y a 24 combinaisons différentes d'hyperparamètres testées.Comme on a $24 \times 10 = 240$ entrainements réalisés (d'où l'affichage "totalling 240 fits").
   Le (CV_score) correspond au score moyen obtenu lors de la validation croisée. Ici, avec 10 plis, la meilleure moyenne de précision est de **94,44%**.
   Les hyperparamètres qui ont donné la meilleure performance dans la validation croisée à 10 plis sont : `C = 100` et `gamma = 0.001`.

2. Dans le cas ou $K = 5$:
   On remarque que pour une validation croisée qui découpe les données en 5 sous-groupes (plis), il y a 24 combinaisons différentes d'hyperparamètres testées.Comme on a $24 \times 5 = 120$ entraînements

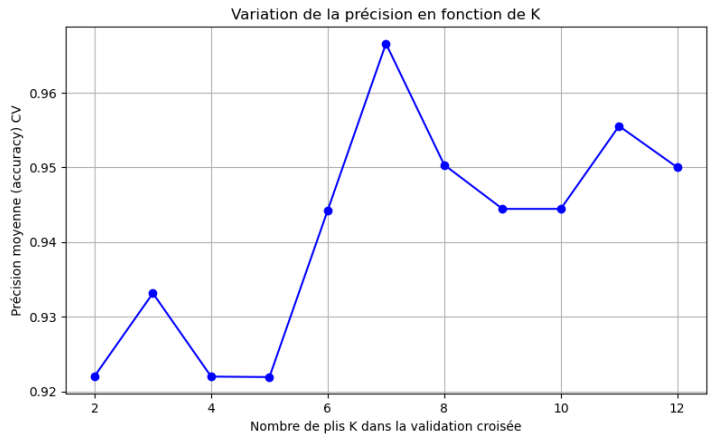
réalisés (d'où l'affichage "totalling 120 fits").
Le (CV_score) correspond au score moyen obtenu lors de la validation croisée. Ici, avec 5 plis, la meilleure moyenne de précision est de **92,19%**.
Les hyperparamètres qui ont donné la meilleure performance dans la validation croisée à 5 plis sont : `C = 100` et `gamma = 0.001`.

De manière à observer visuellement l'effet du paramètre K sur la précision CV, nous avons rédigé le code suivant :

Code Python

```
k_values = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
cv_scores = []

for k in k_values:
    grid_search_k = GridSearchCV(clf, param_grid, cv=k, n_jobs=-1, verbose
        =0)
    grid_search_k.fit(X_train, y_train)
    cv_scores.append(grid_search_k.best_score_)
    print(f"K={k} | Best CV Score: {grid_search_k.best_score_:.4f}")

# Tracer le graphique
plt.figure(figsize=(8,5))
plt.plot(k_values, cv_scores, marker='o', linestyle='-', color='blue')
plt.xlabel("Nombre de plis K dans la validation croisee")
plt.ylabel("Precision moyenne (accuracy) CV")
plt.title("Variation de la precision en fonction de K")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Le graphe obtenu est le suivant :



On observe sur ce graphique que l'on a la meilleure précision pour K=7.

Figure 20: Évolution de la précision globale sur le jeu de test et sur le jeu d'entraînement en fonction de la proportion `test_size`.

Une fois que l'on connait le paramètre K optimal (K=7), nous allons relancer notre code de manière à obtenir les valeurs des hyperparamètres (C et gamma) qui donnent la meilleure performance dans la validation croisée à K=7 plis. Nous trouvons les résultats suivants :

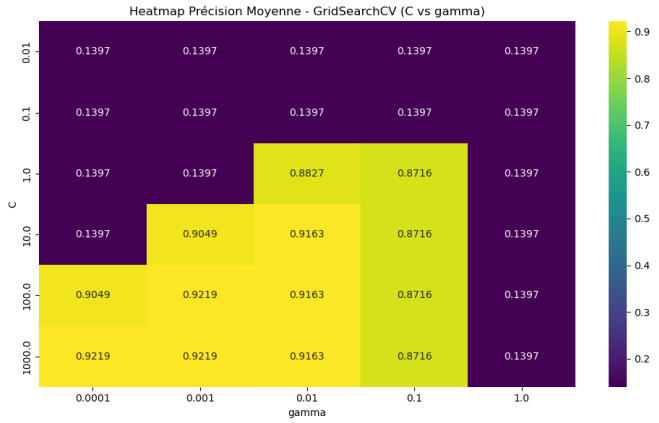
- Il y a 24 combinaisons différentes d'hyperparamètres testées, et $24 \times 7 = 168$ entraînements réalisés (d'où l'affichage "totalling 168 fits").
- Le (CV_score) correspond au score moyen obtenu lors de la validation croisée. Ici, avec 7 plis, la meilleure moyenne de précision est de **96,66%**.
  Les hyperparamètres qui ont donné la meilleure performance dans la validation croisée à 7 plis sont : `C = 100` et `gamma = 0.001`.

De manière à compléter nos observations et d'étudier plus en détails l'influence des paramètres C et gamma sur la précision de notre modèle, nous avons décidé de dessiner une heatmap. En effet, ce graphique nous a aidé à visualiser pour quelle paire (C, gamma) est ce que nous avions les meilleures performances.
Nous avons donc établit le code suivant :

Code Python

```
gamma_vals = [0.0001, 0.001, 0.01, 0.1, 1]
c_vals = [0.01,0.1, 1, 10, 100, 1000]

param_grid = {
    'classifier__C': c_vals,
    'classifier__gamma': gamma_vals
}

pipeline = Pipeline([
    ('features', all_features),
    ('scaler', StandardScaler()),
    ('classifier', SVC(kernel='rbf'))
])

grid_search = GridSearchCV(pipeline, param_grid, cv=5, n_jobs=-1)
grid_search.fit(X_train, y_train)

results_df = pd.DataFrame(grid_search.cv_results_)

# Tableau pour la heatmap
heatmap_data = results_df.pivot_table(
    index='param_classifier__C',
    columns='param_classifier__gamma',
    values='mean_test_score'
)

#Affichage de la heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(heatmap_data, annot=True, fmt=".4f", cmap="viridis")
plt.title("Heatmap Pr cision Moyenne - GridSearchCV (C vs gamma)")
plt.xlabel("gamma")
plt.ylabel("C")
plt.tight_layout()
plt.show()
```

La représentation graphique qui en découle est :



On observe que la performance de notre modèle est d'autant plus haute que gamma n'est petit et C grand ! En effet, ce sont les paires (1000, 0.0001) ; (1000, 0.001) et (100, 0.001) qui ont la précision la plus haute, à savoir 0.9219.
Dans notre cas, nous avons opté pour le couple (100, 0.001) qui est conforme aux résultats trouvés précédemment via l'étude du paramètre K.

Figure 21: Heatmap indiquant la précision de notre modèle en fonction de la paire (C, gamma)

Afin d'obtenir le modèle le plus optimal possible (partie 5), nous choisirons donc les hyperparamètres : `C = 100` et `gamma = 0.001`.

3.5 One-vs-One vs One-vs-Rest

Dans le cadre d'un problème de classification multi-classe, nous avons testé deux approches courantes : **One-vs-One (OvO)** et **One-vs-Rest (OvR)**. L'objectif est d'évaluer leur performance en termes de précision et de temps d'exécution en utilisant le meilleur classifieur trouvé via une recherche par validation croisée.

Prétraitement:
Nous avons extrait les étapes du pipeline d'apprentissage optimal :

Code Python

```
best_pipeline = grid_search.best_estimator_
features = best_pipeline.named_steps["features"]
classifier = best_pipeline.named_steps["classifier"]

X_train_transformed = features.fit_transform(X_train)
X_test_transformed = features.transform(X_test)
```

Méthode One-vs-One (OvO):

La stratégie OvO consiste à entraîner un classifieur pour chaque paire de classes. Si on a K classes, alors on entraîne $\binom{K}{2} = \frac{K(K-1)}{2}$ classifieurs.

Code Python

```
start_ovo = time.time()
ovo = OneVsOneClassifier(classifier)
ovo.fit(X_train_transformed, y_train)
ovo_pred = ovo.predict(X_test_transformed)
ovo_time = time.time() - start_ovo
ovo_acc = accuracy_score(y_test, ovo_pred)
```

Résultats OvO :

- Précision : `0.943`
- Temps d'exécution : `0.65 secondes`

Méthode One-vs-One (OvO):
La stratégie OvR consiste à entraîner un classifieur par classe, en distinguant chaque classe contre toutes les autres. Cette méthode nécessite K modèles pour K classes.

Code Python

```
start_ovr = time.time()
ovr = OneVsRestClassifier(classifier)
ovr.fit(X_train_transformed, y_train)
ovr_pred = ovr.predict(X_test_transformed)
ovr_time = time.time() - start_ovr
ovr_acc = accuracy_score(y_test, ovr_pred)
```

Résultats OvR :

- Précision : `0.947`
- Temps d'exécution : `0.14 secondes`

Comparaison de OvO et OvR:
Pour OvO on a un temps de 0.65 secondes alors que pour OvR on a 0.14 secondes. OvO est plus lent que OvR car il entraîne un classifieur pour chaque paire de classes, ainsi le nombre de modèles à entrainer est plus élevé ce qui augmente le temps. OvR est donc plus efficace en termes de temps, surtout lorsque le nombre de classes est élevé. De plus, pour notre dataset, OvR a une meilleure précision que OvO.

Quelle méthode est la plus adapté suivant le dataset utilisé?
OvR est préférable lorsque le dataset contient un grand nombre de classes. Il est mieux adapté aux grands volumes de données, car il est moins coûteux en termes de calcul. OvO est plus adapté pour les petits datasets.

4 Réseau de neurones

Dans cette partie, nous construisons un réseau de neurones à l'aide de `TensorFlow` et `Keras` pour résoudre un problème de classification multi-classes sur la base de données `digits`. On utilise un réseau de neurones car cela nous permet de décider quel chiffre correspond à F, qui contient les caractéristiques des chiffres. Le réseau de neurones apprend à faire cette association en ajustant ses paramètres avec des données d'entraînement.

Code Python

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
digits = load_digits()
X = digits.data
y = digits.target
X = X / 16.0
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.2,
    random_state=42, stratify=y)
```

Nous divisons le jeu de données en un ensemble d'entraînement qui correspond à 80% et un ensemble de test qui correspond à 20%.

Modèle utilisé

- une couche d'entrée correspondant aux 64 pixels de l'image,
- une couche cachée avec 32 neurones et fonction d'activation relu,

- une couche de sortie avec 10 neurones (un par chiffre) et fonction d'activation `softmax`.

Code Python

```
model = keras.Sequential([keras.layers.Input(shape=(64,)), keras.layers.
    Dense(32, activation='relu'),
     keras.layers.Dense(10, activation='softmax')])
```

Compilation et entraînement Nous compilons le modèle avec :

- l'optimiseur `Adam`,
- la fonction de perte `sparse_categorical_crossentropy`,
- la métrique `accuracy`.

Code Python

```
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])

history = model.fit(X_train, y_train, epochs=45,
     batch_size=32, validation_data=(X_test, y_test), verbose=1)
```

Visualisation des pertes
Nous traçons l'évolution de la perte (loss) au cours de l'entraînement.

Code Python

```
# Extract loss values
train_loss = history.history['loss']
val_loss = history.history['val_loss']
# Plot loss vs epochs
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(train_loss) + 1), train_loss, label='Training Loss',
    marker='o')
plt.plot(range(1, len(val_loss) + 1), val_loss, label='Validation Loss',
    marker='s')
plt.xlabel("Epochs")
plt.yscale("log")  # Apply log scale to the y-axis
plt.ylabel("Loss")
plt.title("Loss vs Epochs")
plt.legend()
plt.grid()
plt.show()
```

On obtient donc le graphe suivant :

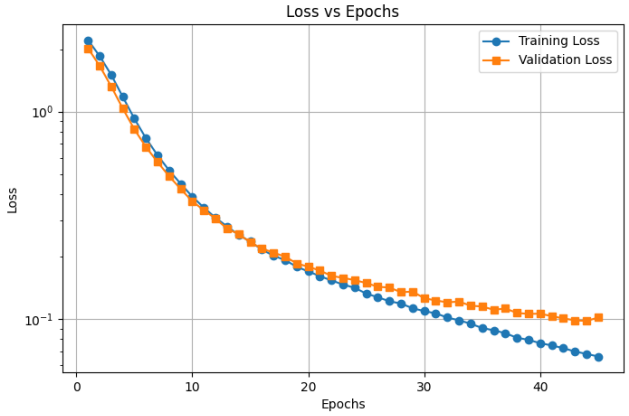


Figure 22: Loss vs Epochs (avec epochs=45)

Le graphe montre une bonne convergence du modèle, avec une diminution rapide puis progressive de la perte (loss) sur les 45 époques. Les courbes d'entraînement et de validation restent proches. Le passage en échelle logarithmique permet de mieux observer la convergence du modèle vers une erreur minimale. La stabilisation vers la fin pour "validation loss" suggère que 30 époques pourraient suffire. On a donc voulu visualiser le phénomène avec un nombre d'époques bien plus grand. On a alors fixé epochs=300, et on a obtenu le graphe suivant :

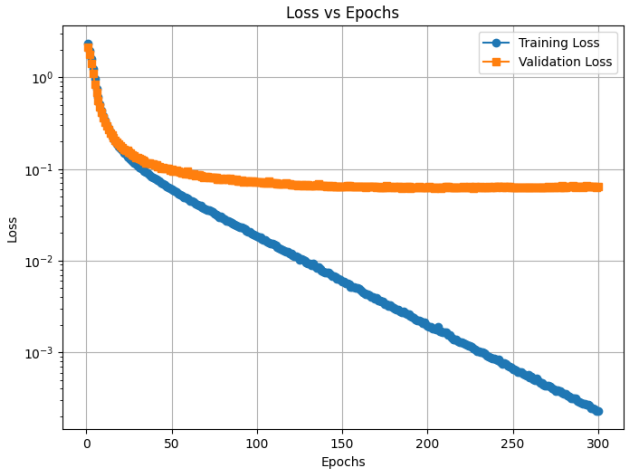


Figure 23: Loss vs Epochs (avec epochs=300)

On peut alors observer que "validation loss" se stabilise à mesure que les époques avancent tandis que "training loss" continue de décroître.

5 Modèle optimal

Dans cette partie finale, notre objectif sera de déterminer le "meilleur" code, avec les paramètres les plus optimaux possibles.
Pour cela, on a besoin d'importer ces différentes bibliothèques:

Code Python

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import FeatureUnion
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from skimage.filters import sobel
```

On a besoin des classes EdgeInfoPreprocessing, ZonalInfoPreprocessing et PCAFeatureExtractor. Enfin, on crée une variable `all_features` qui utilise la classe FeatureUnion qui permet d'unir nos 3 features: PCA, les contours et les zones. On utilise ensuite la classe Pipeline qui applique une transformation de caractéristiques personnalisée, standardise les données, puis entraîne un modèle SVM avec un noyau non linéaire (rbf). Le choix des valeurs des paramètres **C** et **gamma** a été fait à partir des tests d'optimisation fait précédemment. De plus, on a fixé le nombre de composantes du PCA à **29** et le `test_size` à **0.4**, car c'est avec ces valeurs que l'on a les meilleurs résultats.

Code Python

```
all_features = FeatureUnion([('pca', PCAFeatureExtractor(n_components=29)),(
    'sobel', EdgeInfoPreprocessing()), ('zones', ZonalInfoPreprocessing())])
clf = Pipeline([('features', all_features),('scaler', StandardScaler()), ('
    classifier', SVC(kernel='rbf', C=100.0, gamma=0.001))])
```

6 Conclusion

Ce projet a permis d'explorer différentes méthodes de machine learning pour la classification des chiffres manuscrits. Nous avons utilisé des techniques de prétraitement comme la PCA pour réduire la dimensionnalité, extrait des caractéristiques pertinentes (zones, contours), et testé des modèles tels que SVM et un réseau de neurones. Les résultats montrent que le SVM avec noyau RBF, optimisé via validation croisée (C=100, gamma=0.001), offre une précision élevée. Le réseau de neurones a également démontré une bonne performance. Ce travail souligne l'importance du choix des hyperparamètres et des méthodes de prétraitement pour optimiser les modèles.