2025

Projet Intellingence artificielle

JEAN-PIERRE Matthieu

01/01/2025

Phase 1 : Classification de fruits avec un arbre de décision.

- Ce script python est un script de classification des fruits avec un arbre de décision. Le début du code commence par l'importation des différents modules.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
```

- Deuxièmement on crée une liste attribut qui va contenir nos données de description des fruits. Une liste d'étiquette va servir à faire la correspondance entre la description des fruits et le nom du fruit. Les attributs des nouveaux fruits (non encore étiquetés) sont également encodés en utilisant les mêmes LabelEncoder que pour les fruits d'entraînement.

-

```
# Données (couleur, forme)
attributs = [
    ["Rouge", "Ronde"],
    ["Jaune", "Allongée"],
    ["Vert", "Ronde"],
    ["Jaune", "Ronde"],
    ["Rouge", "Allongée"],
    ["Marron","Allongée"],
    ["Marron", "Ronde"],
    ["Orange", "ronde"],
    ["Orange", "grosse"],
    ["Orange","petite"]
]
```

```
]
#ajout de différent fruits dans la liste etiquetteS
etiquettes = ["Pomme", "Banane", "Pomme", "Pomme", "Banane","Noix de coco",
              "Noix de coco", "clementine", "clementine","clementine","raisain","Pruneaux","pomme de terre"]

# Encodage des attributs en remplaçant chaque attribut par LabelEncoder()
encoders = [LabelEncoder() for _ in range(len(attributs[0]))]
```

- On va créer le modèle et entrainer le modèle avec DecisionTreeClassifier() et fit().Chaque fruit est encodé à l'aide de la méthode transform, qui applique le même encodage que celui utilisé pour l'entraînement.

```
nouveaux_fruits_encoded = []

for row in nouveaux_fruits: #on parcour chaque colonne de nouveaux_fruits
    ligne_encodee = []  # création d'un tableaue,
    for i in range(len(row)):  # boucle for permettant de chaque colonne
        valeur_encodee = encoders[i].transform([row[i]])[0]  #  création de la variable qui nous permets de encoder le text déjà connu en un nombre
        ligne_encodee.append(valeur_encodee)  # Ajoute la valeur encodée à ligne_encodee
    nouveaux_fruits_encoded.append(ligne_encodee)  # Ajoute la ligne encodée à la liste finale
```

- Etape suivante nous allons prédire les résultats par rapport à nos données encodées.

```python
# Prédiction
predictions = modele.predict(nouveaux_fruits_encoded) #predire les résustats des donnée provenant de nouveaux_fruits
```

- Pour conclure nous allons afficher les résultats des prédictions

```python
# Afficher les prédictions
print("Prédictions pour les nouveaux fruits :")
for i in range(len(nouveaux_fruits)):
    print(f"Un fruit {nouveaux_fruits[i][0]} et {nouveaux_fruits[i][1]} est prédit comme étant un(e) : {predictions[i]}")
```

Résumé ce script : Ce script python va prédire en fonction des caractéristiques physique présentes dans le tableau attributs et la tableau étiquette. La classification des fruit présentes dans notre liste « nouveaux_fruits ».

```
Prédictions pour les nouveaux fruits :
Un fruit Orange et petite est prédit comme étant un(e) : clementine
Un fruit Jaune et Allongée est prédit comme étant un(e) : Banane
Un fruit Vert et Ronde est prédit comme étant un(e) : Pomme
Un fruit Marron et Ronde est prédit comme étant un(e) : Noix de coco
Un fruit Violet et petit est prédit comme étant un(e) : raisain
Un fruit Orange et grosse est prédit comme étant un(e) : clementine
Un fruit Yellow et ronde est prédit comme étant un(e) : pomme de terre
Un fruit Marron et Allongée est prédit comme étant un(e) : Noix de coco
```

Remarque : j'ai eu des erreurs dans l'exécution de mon code. J'ai dû crée une 'compréhension list' pour résoudre le problème et une boucle for

```python
# Encodage des attributs
encoders = [LabelEncoder() for _ in range(len(attributs[0]))]
# on crée une liste encoders qui va contenir autant de fois que possible LabelEncoder() dans toute les colonnes du tableau attributs
attributs_encoded = [[encoders[i].fit_transform([row[i] for row in attributs])[j] for i in range(len(row))] for j, row in enumerate(attributs)]
# utilisation d'une compréhension liste
```

Phase 2 : Construction d'un modèle MLP sur MNIST – étapes préliminaires

La sortie affiche la variable x qui représente les 70000 images du datasets et 784 représentants les caractéristiques des images.

```
[4]
...  Forme des données d'images (X) : (70000, 784)
     Forme des étiquettes (y) : (70000,)
     Modèle MLP simple créé : MLPClassifier(hidden_layer_sizes=(50, 2), max_iter=10)
     Modèle MLP à deux couches créé : MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=10)
     MLP avec optimiseur Adam : MLPClassifier(hidden_layer_sizes=(50,), max_iter=10)
     MLP avec optimiseur SGD : MLPClassifier(hidden_layer_sizes=(50,), learning_rate_init=0.01, max_iter=10,
                   solver='sgd')
```

La variable y est une matrice de taille 70000 correspondant aux étiquettes de chaque image. Qui va permettre la classification des images.

Lorsque on créer différentes instances de MLPClassifier on doit indiquer le nombre de neurone pour chaque couche cachée, ainsi que le nombre d'itération qui correspond à la répétition de l'entrainement du modèle.

Le solver est un algorithme qui permet d'entrainer des modèles de réseaux de neurones. Dans notre cas 'sgd' fait référence à la descente stochastique utile dans les jeux aléatoires.

Hidden_layer_sizes sert à d'identifier le nombre de neurones à la N ème couche.

Dans notre cas du premier MLP il y a deux sous couches de 50 et 2 neurones.

```
# Exemple 1 : Un MLP très simple avec une seule couche cachée de 50 neurones
mlp_simple = MLPClassifier(hidden_layer_sizes=(50,2), max_iter=10)
print("Modèle MLP simple créé :", mlp_simple)

# Exemple 2 : Un MLP avec deux couches cachées
mlp_deux_couches = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=10)
print("Modèle MLP à deux couches créé :", mlp_deux_couches)

# Exemple 3 : Essayer différents algorithmes d'optimisation
mlp_adam = MLPClassifier(hidden_layer_sizes=(50,), solver='adam', max_iter=10)
print("MLP avec optimiseur Adam :", mlp_adam)

mlp_sgd = MLPClassifier(hidden_layer_sizes=(50,), solver='sgd', learning_rate_init=0.01, max_iter=10)
print("MLP avec optimiseur SGD :", mlp_sgd)

# Remarque : `max_iter` est limité ici pour éviter que l'entraînement ne prenne trop de temps si on l'exécute.
# L'objectif principal est l'instanciation du modèle.
```

PHASE 3 : Entraînement et évaluation d'un modèle MLP sur MNIST

Lors de l'exécution de l'entrainement la durée a été de 31.3s ce qui peut signifier qu'il y a beaucoup de donner dans le Dataset.

La précision en sortie est de 96.46% ce pourcentage montre la précision des chiffres. Plus le taux est haut et plus la précisions de classification est grande. Cela se traduit par fait que le modèle n'a pas réussi à classifier tous les chiffres.

Si on augmenter le nombre d'itération le pourcentage augmente.

```
# 3. Construire un modèle MLP (vous pouvez utiliser un des modèles de l'étape 2 ou en
#    Commençons par un modèle simple pour l'entraînement initial.
# Le nombre d'ter max va jouer sur le temps d'execution du script.
mlp = MLPClassifier(hidden_layer_sizes=(100,50), max_iter=10, random_state=42)
print("Modèle MLP créé :", mlp)
```

Résultat liée a la modification :

```
Modèle MLP créé : MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=10, random_state=42)

Début de l'entraînement du modèle...
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local
  warnings.warn(
Entraînement terminé.

Précision du modèle sur l'ensemble de test : 95.95%

Quelques prédictions et étiquettes réelles :
Image 1: Prédiction = 8, Réel = 8
Image 2: Prédiction = 4, Réel = 4
Image 3: Prédiction = 8, Réel = 8
Image 4: Prédiction = 7, Réel = 7
Image 5: Prédiction = 7, Réel = 7
Image 6: Prédiction = 0, Réel = 0
Image 7: Prédiction = 6, Réel = 6
```

Nous allons modifier le nombre de couche ainsi que le nombre de neurones par couche. À la suite de cela on peut observer qu'il a une augmentation du temps de traitement on passe de 31.3 à 53.s.

Plus le nombre de neurones est important et plus il capturer des de motifs complexe dans les données. Le nombre d'itération max va jouer sur le temps d'entrainement du modèle, une itération allant jusqu'à 40 va prendre environ 1m42.5s

Un nombre important de couche pourra capturer des caractéristiques plus abstraites.

```
# Commençons par un modèle simple pour l'entraînement initial.
# Le nombre d'
mlp = MLPClassifier(hidden_layer_sizes=(100,50), max_iter=40, random_state=42)
print("Modèle MLP créé :", mlp)
```

Ces modifications ont un impact sur le pourcentage de réussite qui augmente lui aussi atteignant 97.79%. Iter_max va permettre de diriger l'algorithme vers un analyse optimale. Mais si on itère trop de fois, l'algorithme risque d'être trop spécialisé et d'être moins généraliste sur ses classifications.

Notre script python contient la fonction train_test_split. Cette fonction va permettre de séparer les données en deux, une partie pour les tests, et un autre pour l'entrainement. Dans notre cas j'ai modifié pour qu'il ait 50/50.

Remarque : On observe une diminution de la précision du modèle. Cela s'explique car 50% de test pourrai donner une estimation plus fiable du modèle. Mais 50% de donnée entrainée est trop faible ce qui pourra causer de problème car le modèle n'aura pas été entrainé avec peu de données disponibles.

Plus la proportion de donnée entrainé diminue, plus le temps le temps d'entrainement sera court.

PHASE 4 : Amélioration de la précision du modèle MLP

Modifcation du hidden_layer_sizes a 50/50

Résultat de hidden_layer_sizes 150, dans cette capture d'écran on observe une couche cachée avec 150 neurones avec un taux de classification de 97.09 ce qui est un assez bon taux

A l'inverse lorsque l'on modifie pour avoir 2 couches cachés avec 50 neurones chacun il y a moins de neurones au total cela peut expliquer le moins bon taux de classification des chiffres.

```
# Modèle 4 : Utilisation de l'optimiseur 'adam' (qui est l'optimiseur par défaut)
mlp_adam = MLPClassifier(hidden_layer_sizes=(50, 50), max_iter=20, solver='adam', random_state=42)
mlp_adam.fit(X_train, y_train)
y_pred_adam = mlp_adam.predict(X_test)
accuracy_adam = accuracy_score(y_test, y_pred_adam)
print(f"Précision avec l'optimiseur Adam : {accuracy_adam * 100:.2f}%")

# Modèle 5 : Utilisation de l'optimiseur 'sgd' (Stochastic Gradient Descent) lbfgsvec un taux d'apprentissage
mlp_sgd = MLPClassifier(hidden_layer_sizes=(50, 50 ), max_iter=20, solver='lbfgs', learning_rate_init=0.01, random_state=42)
mlp_sgd.fit(X_train, y_train)
y_pred_sgd = mlp_sgd.predict(X_test)
accuracy_sgd = accuracy_score(y_test, y_pred_sgd)
print(f"Précision avec l'optimiseur lbfsg (taux d'apprentissage=0.01) : {accuracy_sgd * 100:.2f}%")
```

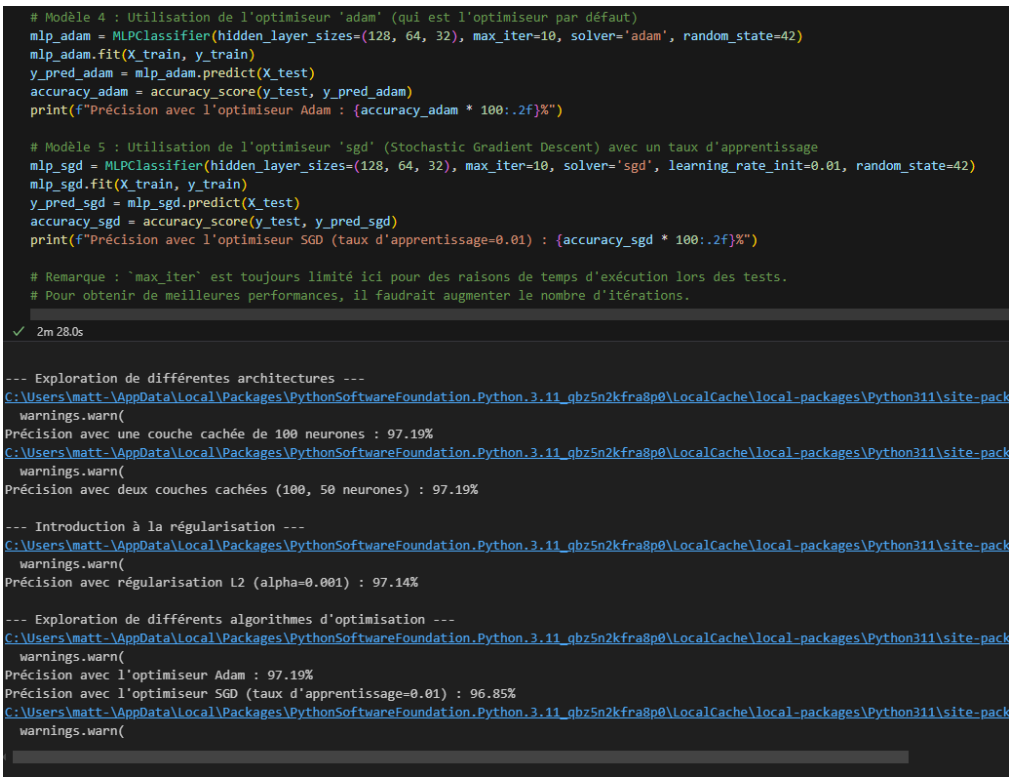
```
mlp_regularized = MLPClassifier(hidden_layer_sizes=(128,64, 32 ), max_iter=10, alpha=0.011, random_state=42)
mlp_regularized.fit(X_train, y_train)
y_pred_regularized = mlp_regularized.predict(X_test)
accuracy_regularized = accuracy_score(y_test, y_pred_regularized)
print(f"Précision avec régularisation L2 (alpha=0.001) : {accuracy_regularized * 100:.2f}%")
```

```
Précision avec régularisation L2 (alpha=0.001) : 97.09%
```

Ps : ne fait pas attention au print je ne les modifie pas dans les étapes

Cette configuration (128,64,32) est plus performante que les anciennes car hidden_layer_sizes contient 3 couches cachés la première avec 128 neurones ce qui va permettre les calculs les plus complexes, ensuite 64 le pour raffiner les données de la première, et la dernière de 32 pour aider à synthétiser les caractéristiques les plus pertinentes pour la classification.

Dans l’ensemble il va y avoir un filtrage progressif avec moins de neurones au fur et à mesure. Ce qui pourra rendre le modèle généraliste et donc un meilleur pourcentage de classification.



```python
# Modèle 4 : Utilisation de l'optimiseur 'adam' (qui est l'optimiseur par défaut)
mlp_adam = MLPClassifier(hidden_layer_sizes=(128, 64, 32), max_iter=10, solver='adam', random_state=42)
mlp_adam.fit(X_train, y_train)
y_pred_adam = mlp_adam.predict(X_test)
accuracy_adam = accuracy_score(y_test, y_pred_adam)
print(f"Précision avec l'optimiseur Adam : {accuracy_adam * 100:.2f}%")

# Modèle 5 : Utilisation de l'optimiseur 'sgd' (Stochastic Gradient Descent) avec un taux d'apprentissage
mlp_sgd = MLPClassifier(hidden_layer_sizes=(128, 64, 32), max_iter=10, solver='sgd', learning_rate_init=0.01, random_state=42)
mlp_sgd.fit(X_train, y_train)
y_pred_sgd = mlp_sgd.predict(X_test)
accuracy_sgd = accuracy_score(y_test, y_pred_sgd)
print(f"Précision avec l'optimiseur SGD (taux d'apprentissage=0.01) : {accuracy_sgd * 100:.2f}%")

# Remarque : `max_iter` est toujours limité ici pour des raisons de temps d'exécution lors des tests.
# Pour obtenir de meilleures performances, il faudrait augmenter le nombre d'itérations.
```

✓ 2m 28.0s

```
--- Exploration de différentes architectures ---
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-pack
  warnings.warn(
Précision avec une couche cachée de 100 neurones : 97.19%
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-pack
  warnings.warn(
Précision avec deux couches cachées (100, 50 neurones) : 97.19%

--- Introduction à la régularisation ---
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-pack
  warnings.warn(
Précision avec régularisation L2 (alpha=0.001) : 97.14%

--- Exploration de différents algorithmes d'optimisation ---
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-pack
  warnings.warn(
Précision avec l'optimiseur Adam : 97.19%
Précision avec l'optimiseur SGD (taux d'apprentissage=0.01) : 96.85%
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-pack
  warnings.warn(
```

Ne nombre de couches et neurones varie en fonction du volumes de nos données et de la complexité de leurs caractéristiques.

Un nombre important de couches ou de neurones pourront avoir aussi un risque de surapprentissage car modèle sera trop spécialisé et donc non généraliste.

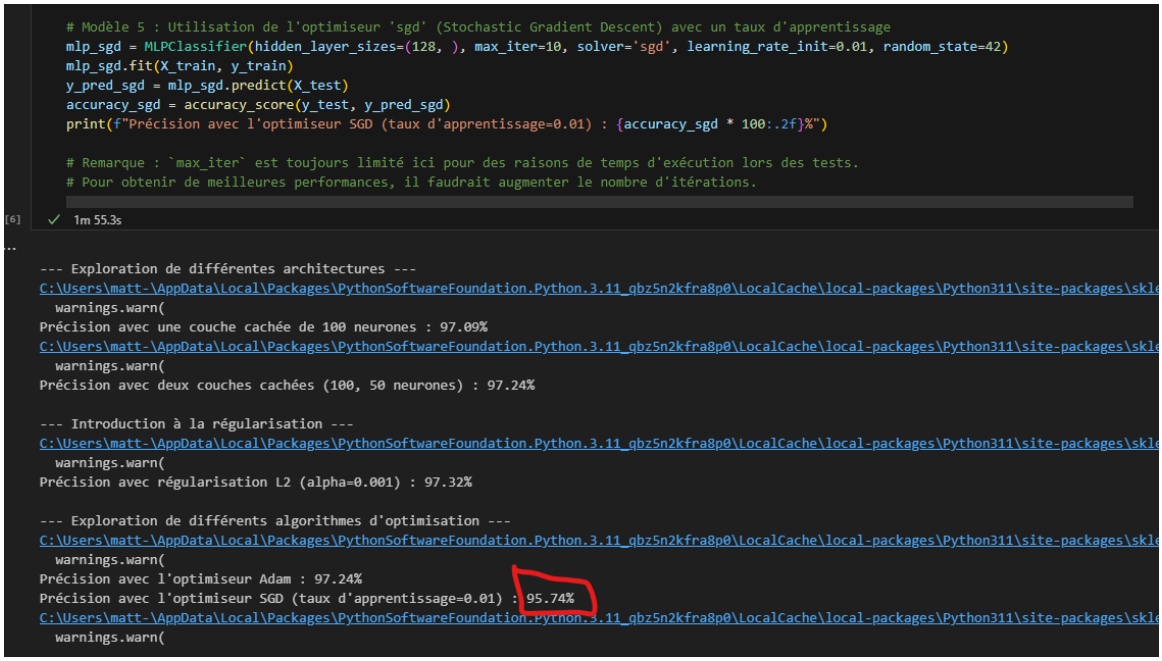
Alpha dans ce script est utilisé contre le surajustement plus alpha est élevé et plus le modèle sera incapable de prédire avec précision. A l’inverse plus alpha est petit et plus le modèle pourrait surajuster.

Optimisation : Un taux élevé au niveau de learning_rate_init va avoir conséquences de mises à jour importante ce accélère la convergence.

Un taux faible a l’inverse entraine de mise a jours moindres qui entraine une faible convergence.

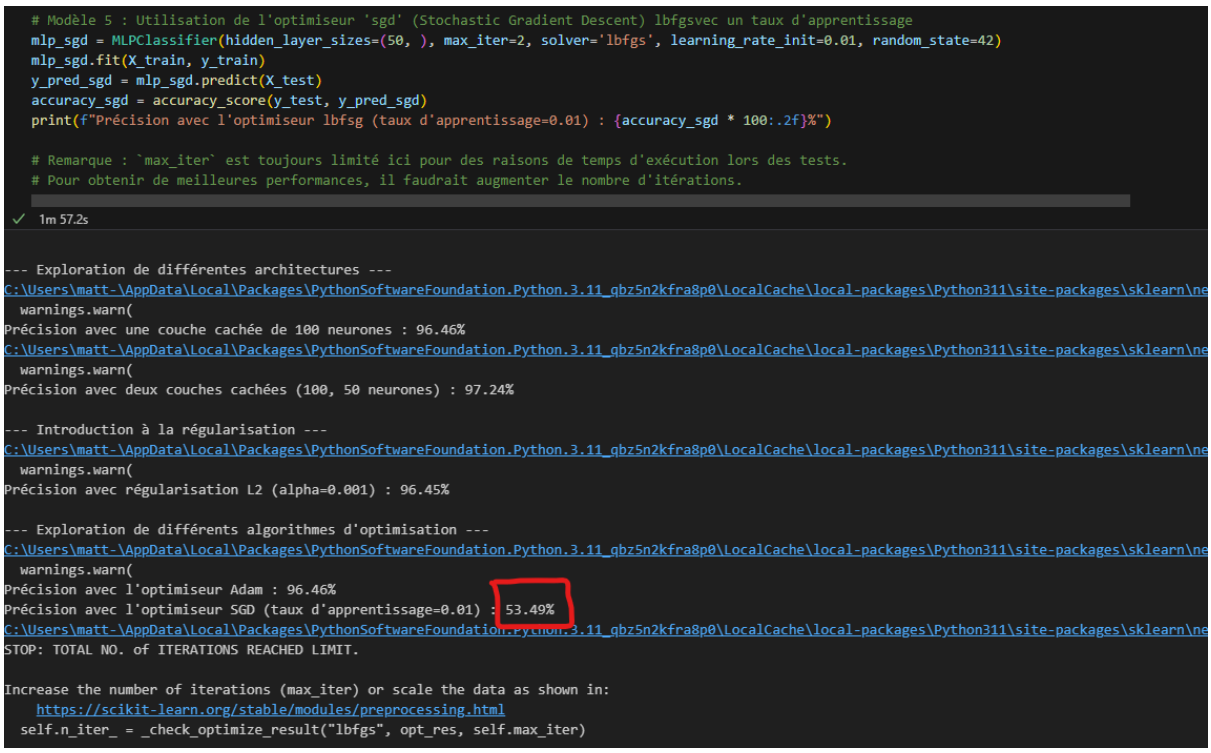
Cet exemple j’ai modifier pour avoir 0.0001 il y a eu une convergence qui a diminué le taux de classification

Cette capture d’écran nous montre qu’avec une config à 0.01 on obtient un meilleur taux de classification.



```
    # Modèle 5 : Utilisation de l'optimiseur 'sgd' (Stochastic Gradient Descent) avec un taux d'apprentissage
    mlp_sgd = MLPClassifier(hidden_layer_sizes=(128, ), max_iter=10, solver='sgd', learning_rate_init=0.01, random_state=42)
    mlp_sgd.fit(X_train, y_train)
    y_pred_sgd = mlp_sgd.predict(X_test)
    accuracy_sgd = accuracy_score(y_test, y_pred_sgd)
    print(f"Précision avec l'optimiseur SGD (taux d'apprentissage=0.01) : {accuracy_sgd * 100:.2f}%")

    # Remarque : `max_iter` est toujours limité ici pour des raisons de temps d'exécution lors des tests.
    # Pour obtenir de meilleures performances, il faudrait augmenter le nombre d'itérations.
[6] ✓ 1m 55.3s

...
    --- Exploration de différentes architectures ---
    C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\skle
      warnings.warn(
    Précision avec une couche cachée de 100 neurones : 97.09%
    C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\skle
      warnings.warn(
    Précision avec deux couches cachées (100, 50 neurones) : 97.24%

    --- Introduction à la régularisation ---
    C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\skle
      warnings.warn(
    Précision avec régularisation L2 (alpha=0.001) : 97.32%

    --- Exploration de différents algorithmes d'optimisation ---
    C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\skle
      warnings.warn(
    Précision avec l'optimiseur Adam : 97.24%
    Précision avec l'optimiseur SGD (taux d'apprentissage=0.01) : 95.74%
    C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\skle
      warnings.warn(
```

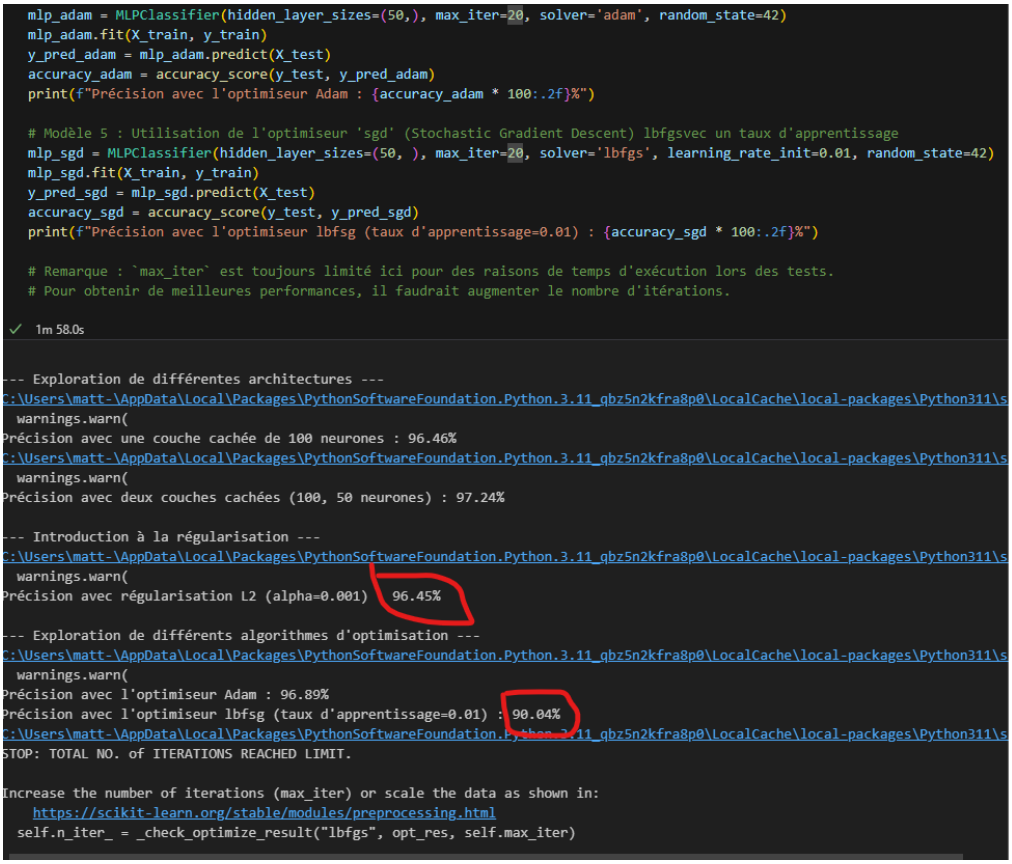
On peut conclure que le choix du taux d'apprentissage initial peut influencer l'entraînement et la performance d'un modèle de réseau de neurones dans un contexte de régression.



```
  # Modèle 5 : Utilisation de l'optimiseur 'sgd' (Stochastic Gradient Descent) lbfgsvec un taux d'apprentissage
  mlp_sgd = MLPClassifier(hidden_layer_sizes=(50, ), max_iter=2, solver='lbfgs', learning_rate_init=0.01, random_state=42)
  mlp_sgd.fit(X_train, y_train)
  y_pred_sgd = mlp_sgd.predict(X_test)
  accuracy_sgd = accuracy_score(y_test, y_pred_sgd)
  print(f"Précision avec l'optimiseur lbfsg (taux d'apprentissage=0.01) : {accuracy_sgd * 100:.2f}%")

  # Remarque : `max_iter` est toujours limité ici pour des raisons de temps d'exécution lors des tests.
  # Pour obtenir de meilleures performances, il faudrait augmenter le nombre d'itérations.
✓ 1m 57.2s

--- Exploration de différentes architectures ---
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\ne
  warnings.warn(
Précision avec une couche cachée de 100 neurones : 96.46%
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\ne
  warnings.warn(
Précision avec deux couches cachées (100, 50 neurones) : 97.24%

--- Introduction à la régularisation ---
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\ne
  warnings.warn(
Précision avec régularisation L2 (alpha=0.001) : 96.45%

--- Exploration de différents algorithmes d'optimisation ---
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\ne
  warnings.warn(
Précision avec l'optimiseur Adam : 96.46%
Précision avec l'optimiseur SGD (taux d'apprentissage=0.01) : 53.49%
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\ne
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)
```

Lorsque on test le ‘lbfsg’ on remarque que c’est pas du tout adapté car il ya seulement 53.49 % de classification.

Nous pouvons voir que lorsque nous augmentons le nombre d'itération il y a une augmentation du taux de classification. On peut conclure que plus le nombre d'itération sera important et plus le modèle sera efficace.



```
mlp_adam = MLPClassifier(hidden_layer_sizes=(50,), max_iter=20, solver='adam', random_state=42)
mlp_adam.fit(X_train, y_train)
y_pred_adam = mlp_adam.predict(X_test)
accuracy_adam = accuracy_score(y_test, y_pred_adam)
print(f"Précision avec l'optimiseur Adam : {accuracy_adam * 100:.2f}%")

# Modèle 5 : Utilisation de l'optimiseur 'sgd' (Stochastic Gradient Descent) lbfgsvec un taux d'apprentissage
mlp_sgd = MLPClassifier(hidden_layer_sizes=(50, ), max_iter=20, solver='lbfgs', learning_rate_init=0.01, random_state=42)
mlp_sgd.fit(X_train, y_train)
y_pred_sgd = mlp_sgd.predict(X_test)
accuracy_sgd = accuracy_score(y_test, y_pred_sgd)
print(f"Précision avec l'optimiseur lbfsg (taux d'apprentissage=0.01) : {accuracy_sgd * 100:.2f}%")

# Remarque : `max_iter` est toujours limité ici pour des raisons de temps d'exécution lors des tests.
# Pour obtenir de meilleures performances, il faudrait augmenter le nombre d'itérations.
```

✓ 1m 58.0s

```
--- Exploration de différentes architectures ---
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\s
  warnings.warn(
Précision avec une couche cachée de 100 neurones : 96.46%
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\s
  warnings.warn(
Précision avec deux couches cachées (100, 50 neurones) : 97.24%

--- Introduction à la régularisation ---
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\s
  warnings.warn(
Précision avec régularisation L2 (alpha=0.001)  96.45%

--- Exploration de différents algorithmes d'optimisation ---
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\s
  warnings.warn(
Précision avec l'optimiseur Adam : 96.89%
Précision avec l'optimiseur lbfsg (taux d'apprentissage=0.01) : 90.04%
C:\Users\matt-\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packages\Python311\s
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
  self.n_iter_ = _check_optimize_result("lbfgs", opt_res, self.max_iter)
```

PHASE 5 : VISUALISATION

Observation :

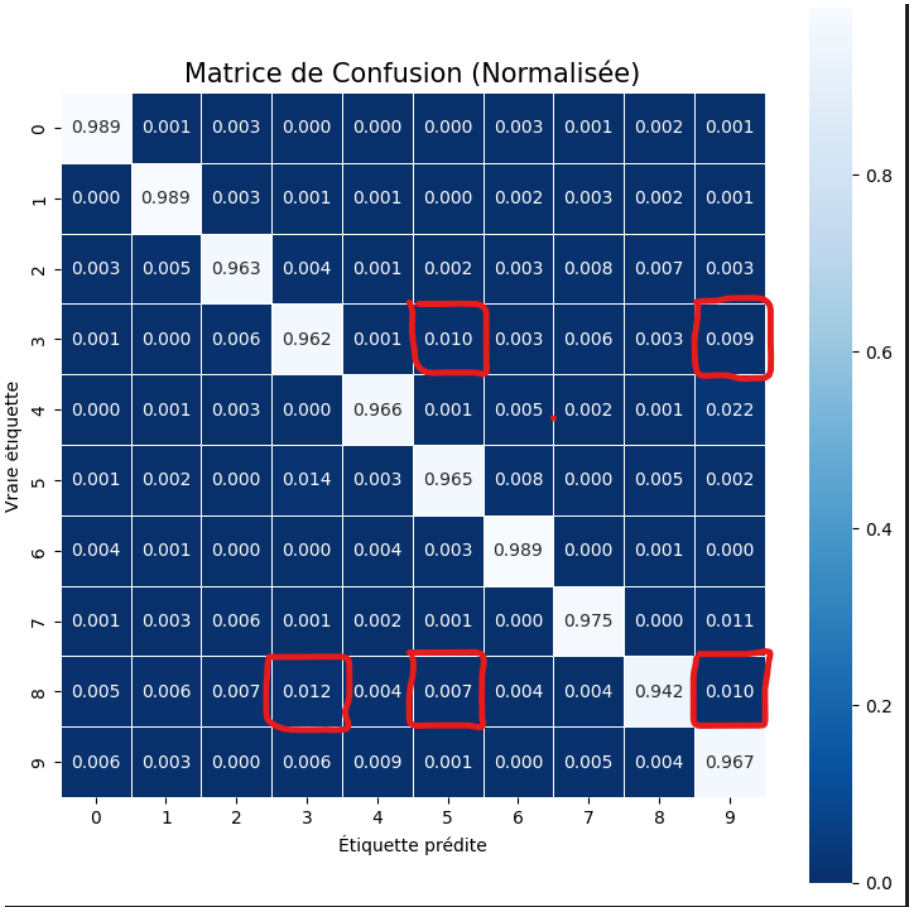
- On observe que le modelé ne s'est pas trompé dans la prédiction des chiffres
- Au niveau de la matrice de confusion on observe qu'il y a une bonne performance du modèle. Mais il a des difficultés sur le 8 et le 3. On peut par exemple voir pour le chiffre 8 qu'il du mal à faire la différence avec le chiffre 3, 5 et 9
- Les lignes représentent les vraies étiquettes et les colonnes la prédiction
- Les valeurs situées sur la diagonal représentent les cas où les étiquettes prédites sont correctes.
- Les valeurs les plus élevés se trouvent principalement dans les diagonales
- En dehors de la zone diagonale il y a un pourcentage faible. Ces chiffres nous montrent le pourcentage des prédictions par rapport aux vrais chiffres.

Expérimentation :

En modifiant l'architecture comment par exemple rajouter la régulation L2 On observe qu'il y a une diminution du pourcentage. Cela s'explique car le seuil que j'ai mis est assez haut. Ce qui aura pour conséquence de limiter le surapprentissage.

L'affiche de plus de 10 images nécessite de modifier la boucle for et aussi modifier afin d'avoir par exemple 3 lignes et colonnes.

```
print("Classes réelles     (les 20 premières) :", y_test[:20])

# 5. Visualiser les prédictions sur les 15 premières images de test
plt.figure(figsize=(20, 6))
for index in range(15):
    plt.subplot(3, 5, index + 1)

    plt.imshow(X_test.iloc[index].values.reshape(28, 28), cmap=plt.cm.gray)
    plt.title(f"Prédit : {y_test_pred[index]}")
    plt.axis("off")
plt.tight_layout()
plt.show()
```



Matrice de confusion après modification : j'ai un taux de 97.09

Apres augmentation du nombre d’image mon pourcentage reste le même.