

Titre professionnel

**Développeur en intelligence
artificielle**

RNCP 34757

Rapport E2 compétence C8 et C14

**Reconnaissance de chiffres
manuscrits**

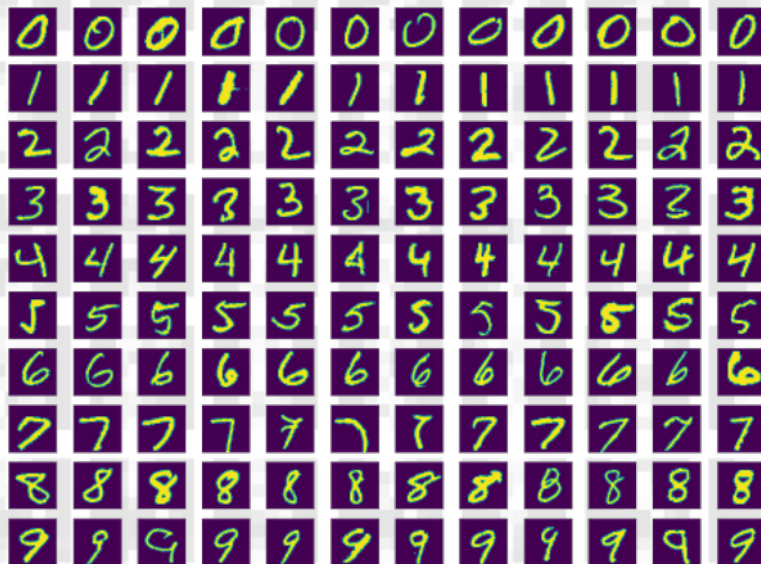


Table des matières

1	Présentation du projet existant.....	3
a	Préambule.....	3
b	Présentation du code.....	3
i	Présentation des données :.....	3
ii	Visualisation des images contenues dans la base de données.....	3
iii	Mise en œuvre de l'algorithme choisi.....	4
c	Conclusion.....	5
2	Amélioration du projet.....	6
a	Amélioration sur la qualité des images.....	6
i	Chargement de la base de données.....	6
ii	Visualisation d'un échantillon d'images.....	6
iii	Réduction du dataset.....	7
iv	Test de l'algorithme dans les mêmes conditions que sur le 1 ^{er} dataset.....	8
v	Amélioration de la régression linéaire.....	8
vi	GridSearchCV.....	10
vii	Learning curve, testons l'hypothèse 3.....	12
viii	Conclusion.....	13
b	Amélioration en testant de nouveaux algorithmes.....	13
i	Test du classifieur KNN.....	13
ii	Test avec un réseau de neurones.....	14
c	Conclusion de l'étude sur les 3 modèles proposés.....	15
3	Ajout de nouvelles fonctionnalités.....	15
a	Cahier des charges :.....	16
b	Test du programme final "lecture_cm.py" :.....	16
4	Test de non régression.....	17
5	Conclusion.....	17
6	Annexes	17
a	Jupyter notebook :.....	17
b	Programme python :.....	17

1 Présentation du projet existant.

a Préambule.

Il s'agit du projet d'étude numéro 5 de reconnaissance de chiffres manuscrits. La base de données utilisée est une base dérivée de la base MNIST (Modified National Institute of Standards and Technology) mais diffère par la dimension des images (8 x 8 pixels contre 28 x 28 pixels), par le codage des images (16 niveau de gris contre 256) ainsi que par la taille de la base de données (1797 échantillons contre 70 000).

La base de scikit-learn contient donc 1797 chiffres manuscrits de 0 à 9 avec environ 180 exemples par chiffre (représenté par une image de 8 x 8 pixels codée en 16 niveau de gris) et permet de tester facilement de nombreux algorithmes sans avoir à disposer d'une grande puissance de calcul.

La reconnaissance de l'écriture manuscrite est un problème difficile, ayant de multiples applications (on pense naturellement au déchiffrement des codes postaux à haute vitesse effectué dans les centres de tri).

La base MNIST est devenue une base de données de référence sur lequel bon nombre d'algorithmes ont été testés. Le site de Yann Lecun (<http://yann.lecun.com/exdb/mnist/>) donne un tableau récapitulatif par type d'algorithme et par algorithme des meilleurs résultats obtenus (le résultat le plus récent date de 2012 avec 0,23% d'erreur).

Selon le site Wikipédia (https://fr.wikipedia.org/wiki/Base_de_donn%C3%A9es_MNIST) un record a été établi en 2018 avec un taux d'erreur de 0,18 %.

Autant le dire tout de suite, s'il n'est pas si difficile que cela d'obtenir de bons résultats (au delà de 98%) le passage avec de "vraies" données peut réserver des surprises !

b Présentation du code.

Je ne donnerai pas ici tout le code du brief (l'intégralité du code se trouve en annexe) mais simplement les parties en rapport avec la classification multi-classe. L'exercice de classification mono-classe ainsi que l'exercice de codage d'une descente de gradient (sans utilisation de la librairie scikit-learn) ne feront donc pas partie de ce rapport.

i Présentation des données :

```
1 # Importation des données depuis scikit learn datasets
2 digits = load_digits()
3
4 # Les images sont organisées en matrice de 8x8
5 X = digits.images
6 # Labels des images (chiffre de 0 à 9)
7 Y = digits.target
8 # Les images sont "aplaties" et transformées en un tableau de 64 éléments
9 Xdata = digits.data
10
11 print(f"X Shape: {X.shape}, Xdata Shape: {Xdata.shape}, Y shape: {Y.shape}, "+
12       f"digits.target_name: {digits.target_names}\n")
```

X Shape: (1797, 8, 8), Xdata Shape: (1797, 64), Y shape: (1797,), digits.target_name: [0 1 2 3 4 5 6 7 8 9]

ii Visualisation des images contenues dans la base de données.

Ce bout de code permet d'afficher certaines images contenues dans la base de données choisie de façon aléatoire mais ordonnée par classe (de 0 à 9).

```

1 # Préparation de l'affichage
2 n_digits = np.unique(Y) # = 0,1,2,...9
3 M = 12
4
5 fig, axs = plt.subplots(len(n_digits), M, figsize=(20, 15))
6
7 # Afficher M exemples de tout les digits (de 0 à 9)
8 for i, d in enumerate(n_digits):
9     x = X[Y == d]
10    for j in range(M):
11        num = random.randint(0, x.shape[0]-1)
12        axs[i,j].imshow(X[Y == d][j], cmap="gray")
13        axs[i,j].axis('off')

```



Comme on peut le voir, la qualité des images est plutôt médiocre. Ce sera d'ailleurs un des axes d'amélioration du programme.

iii Mise en œuvre de l'algorithme choisi.

Nous utiliserons la régression logistique de scikit learn pour réaliser la tâche de classification des chiffres. On notera que nous n'avons pas cherché à optimiser le résultat obtenu (nous avons réalisé cet exercice au début de la formation, il s'agissait alors d'apprendre les différentes notions qui seront approfondies plus tard). Par ailleurs, bon nombre de tutos se contentent très souvent des paramètres par défaut des algorithmes qui sont, il faut le noter, plutôt pertinents. Mais il est clair que chercher à optimiser ces paramètres apporte souvent des gains appréciables tant au niveau de la précision, que de la qualité des prédictions (recall). Ce sera l'objet d'un autre axe d'amélioration.

Dans un premier temps, nous partageons les données en un jeu d'entraînement (80%) et un jeu de test (20%), puis nous construisons le modèle, enfin nous testons les prédictions réalisées. Un certain nombre d'erreurs apparaissent.

```

1 # On split le jeux de données
2 X_train, X_test, Y_train, Y_test = train_test_split(Xdata, Y, test_size=0.2, random_state= 511)
3
4 # On construit et on 'fit' le modèle
5 log_regr = LogisticRegression(solver='liblinear')
6 log_regr.fit(X_train, Y_train)
7
8 # on teste sur les données de.... test :)
9 prediction = log_regr.predict(X_test)
10
11 err, pre, d= erreur_prediction(valeurs_predites= prediction, valeurs_reelle= Y_test)
12 print(f"{err} erreurs sur {len(X_test)} images. soit une précision de {pre:.2f} %")
13 print(f"Dictionnaire des erreurs (clé valeur réelle: valeur prédite):\n{d}")
14

```

```

19 erreurs sur 360 images. soit une précision de 94.72 %
Dictionnaire des erreurs (clé valeur réelle: valeur prédite):
{8: [1, 1, 1, 3, 1], 3: [5, 8], 9: [8, 3, 4, 8], 5: [9, 9, 9, 9], 4: [7, 6], 1: [6, 8]}

```

Pour mieux quantifier ces erreurs, nous affichons la précision, la sensibilité (recall) ainsi que le f1 score (qui combine la précision et le recall en une sorte de moyenne pondérée) :

```

1 target_names= ["détection de 0", "détection de 1", "détection de 2", "détection de 3", "détection de 4",
2               "détection de 5", "détection de 6", "détection de 7", "détection de 8", "détection de 9"]
3 print("\n",classification_report(Y_test,prediction, target_names=target_names))
4

```

	precision	recall	f1-score	support
détection de 0	1.00	1.00	1.00	31
détection de 1	0.89	0.94	0.91	33
détection de 2	1.00	1.00	1.00	35
détection de 3	0.94	0.94	0.94	36
détection de 4	0.97	0.95	0.96	41
détection de 5	0.98	0.91	0.94	45
détection de 6	0.95	1.00	0.97	35
détection de 7	0.97	1.00	0.98	31
détection de 8	0.90	0.88	0.89	40
détection de 9	0.88	0.88	0.88	33
accuracy			0.95	360
macro avg	0.95	0.95	0.95	360
weighted avg	0.95	0.95	0.95	360

c Conclusion.

Le score obtenu est plutôt flatteur compte tenu du faible investissement dans la recherche d'optimisation. Mais quelques problèmes incitent à rester méfiant sur ce résultat.

D'abord la faible qualité des images n'augure pas de bons résultats sur des chiffres manuscrits qui ne proviendraient pas de cette base. Ensuite un simple changement de la graine initialisant une suite pseudo aléatoire dans la fonction `train_test_split` fait varier les résultats de façon importante (la précision augmente de 2 % passant à 96,5% avec `random_state= 42` ... le plus célèbre des nombres au lieu de 511). Enfin "lbfgs" qui est l'algorithme d'optimisation par défaut produit un avertissement indiquant qu'il n'a pas pu converger suffisamment (d'où son remplacement par "liblinear"). Pour que ce programme puisse fonctionner de façon correcte dans la « vraie vie » il faut donc l'améliorer quelque peu !

2 Amélioration du projet

Des pistes pour améliorer le projet sont l'amélioration de la qualité de l'image, l'amélioration de l'algorithme utilisé avec optimisation des hyper-paramètres et recherche d'autres algorithmes qui pourraient être utilisés

a Amélioration sur la qualité des images.

Dans un premier temps nous pourrions travailler sur le dataset original avec des images de meilleures qualités. Ce qui nous permettra d'avoir un algorithme utilisable dans des conditions réelles.

i Chargement de la base de données.

Cette base peut être chargée depuis la librairie h5py, mais j'aurais pu tout autant télécharger directement les données depuis un site de dataset.

```
1 # Importation du jeux d'entraînement
2 f = h5py.File("train.hdf5", 'r')
3 train_x, train_y = f['image'][...], f['label'][...]
4 f.close()
5
6 # Importation du jeux de test
7 f = h5py.File("test.hdf5", 'r')
8 test_x, test_y = f['image'][...], f['label'][...]
9 f.close()
10
11 etiquette= np.unique(test_y)
12 print(f"Entrainement: X Shape: {train_x.shape}, Y shape: {train_y.shape}\n"+\
13       f"Test: X Shape: {test_x.shape}, Y shape: {test_y.shape}\n"+\
14       f"Etiquette: {etiquette}")
```

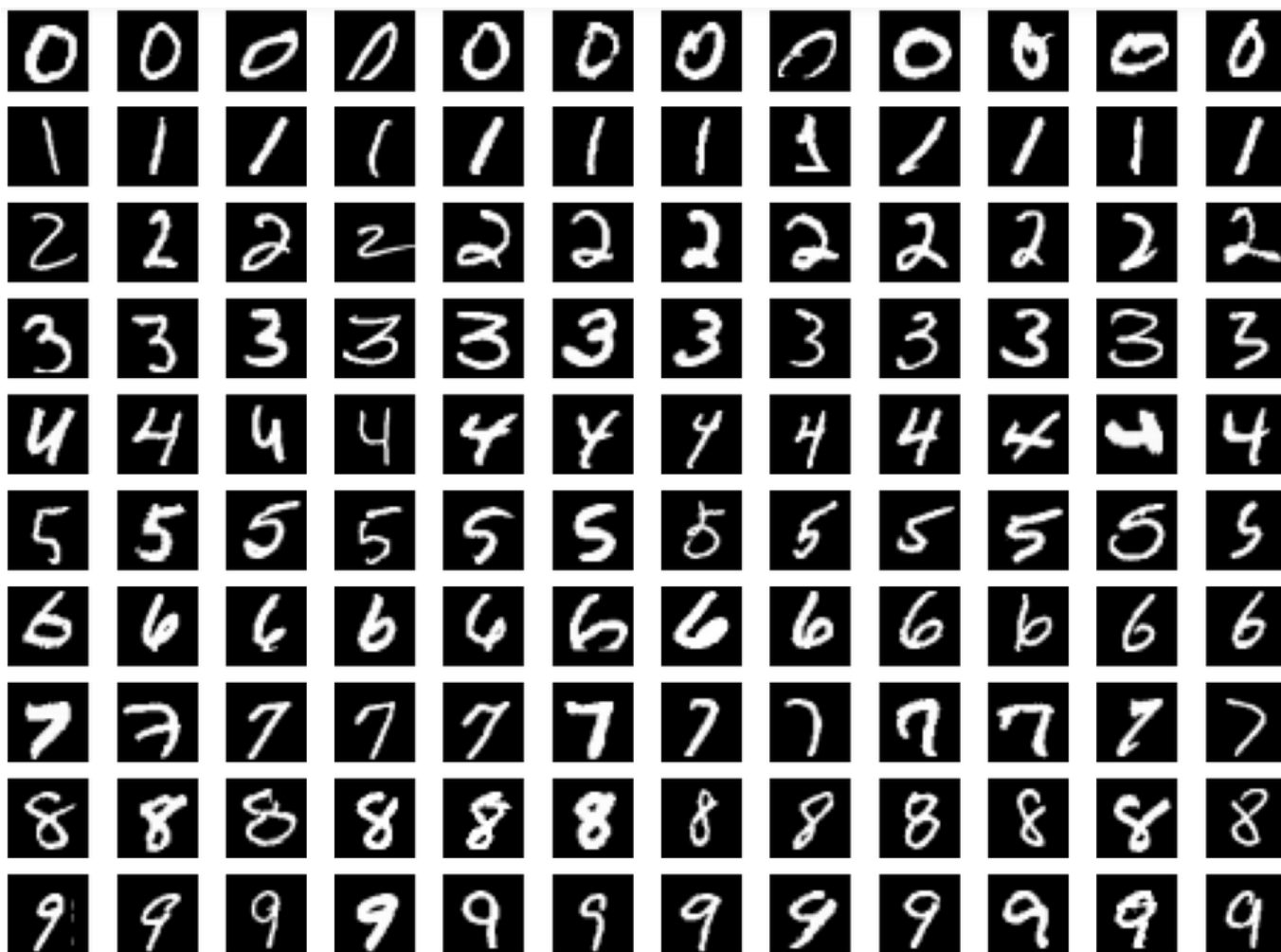
```
Entrainement: X Shape: (60000, 28, 28), Y shape: (60000,)
Test: X Shape: (10000, 28, 28), Y shape: (10000,)
Etiquette: [0 1 2 3 4 5 6 7 8 9]
```

Comme on peut le voir, cette base comprend 70000 données. Compte tenu des limitations de l'ordinateur d'une part, et le fait que nous n'aurons pas besoin de toutes ces données pour réaliser les tests sur les divers algorithmes d'autre part, je ne prendrais qu'une partie de ces données.

ii Visualisation d'un échantillon d'images.

```
1 # Préparation de l'affichage
2 M= 12
3 fig, axs = plt.subplots(len(etiquette), M, figsize=(20, 15))
4
5 # Afficher M exemples de tout les digits (de 0 à 9)
6 for i, d in enumerate(etiquette):
7     x= train_x[train_y == d]
8     for j in range(M):
9         num= random.randint(0, x.shape[0]-1)
10        axs[i,j].imshow(x[num], cmap= "gray")
11        axs[i,j].axis('off')
12
```

Comme on peut le constater sur la figure ci-dessous, les images sont de bien meilleure qualité. On peut aussi constater une grande variété dans l'écriture ce qui signifie que pour livrer un produit efficace, il faudra sans doute intégrer plus de données, peut-être même la totalité du dataset. L'étude de la courbe d'apprentissage (fonction learning curve de scikit learn) pourra être utile. Mais pour l'instant il s'agit de tester l'algorithme de régression linéaire utilisé précédemment.



iii Réduction du dataset.

Le précédent dataset sur lequel j'ai travaillé avait 1797 images. On va en construire un plus grand sans pour autant prendre la totalité du dataset MNIST. En effet, une optimisation des hyperparamètres par grid searchCV en prenant la totalité du dataset nécessiterait une puissance machine trop importante.

```

1 # On récupère les données. On ne prendra qu'une partie des données situé dans le
2 # jeux de train. On en profite pour "aplatir" les images
3 data= train_x.reshape(train_x.shape[0], train_x.shape[1]*train_x.shape[2])
4
5 # On réduit le jeux de données original
6 data_red, _, Y_red, _ = train_test_split(data, train_y, train_size=0.1, random_state= 65)
7
8 # On split le jeux de données
9 X_train_red, X_test_red, Y_train_red, Y_test_red = train_test_split(data_red, Y_red, train_size=0.8,
10                                                                    random_state= 65)
11
12 print(f"Entrainement: X Shape: {X_train_red.shape}, Y shape: {Y_train_red.shape}\n"+\
13       f"Test: X Shape: {X_test_red.shape}, Y shape: {Y_test_red.shape}\n"+\
14       f"Etiquette: {etiquette}")

```

Entrainement: X Shape: (4800, 784), Y shape: (4800,)

Test: X Shape: (1200, 784), Y shape: (1200,)

Etiquette: [0 1 2 3 4 5 6 7 8 9]

On regarde si ce dataset est correctement réparti.


```

1 # Le jeux de donnée est-il équilibré ?
2 d= {}
3 for i in etiquette:
4     l= len(X_train_red[Y_train_red== i])
5     d[i]= l
6     print(f"Le chiffre {i} est représenté {l} fois", end= " - ")
7 mini, maxi= min(d, key= d.get), max(d, key= d.get)
8 print(f"\nLe chiffre le moins représenté est {mini} représenté {d[mini]} fois.")
9 print(f"Le chiffre le plus représenté est {maxi} représenté {d[maxi]} fois.")

```

Le chiffre 0 est représenté 483 fois - Le chiffre 1 est représenté 546 fois - Le chiffre 2 est représenté 472 fois -
 Le chiffre 3 est représenté 483 fois - Le chiffre 4 est représenté 473 fois - Le chiffre 5 est représenté 431 fois -
 Le chiffre 6 est représenté 482 fois - Le chiffre 7 est représenté 480 fois - Le chiffre 8 est représenté 487 fois -
 Le chiffre 9 est représenté 463 fois -
 Le chiffre le moins représenté est 5 représenté 431 fois.
 Le chiffre le plus représenté est 1 représenté 546 fois.

On peut constater une différence significative entre le chiffre le plus représenté et celui étant le moins représenté. Un paramètre de la fonction de régression logistique permet de tenir compte de ce problème (class_weight) mais ne sera pas utilisé ici pour pouvoir comparer les résultats obtenus avec les deux différents datasets.

iv Test de l'algorithme dans les mêmes conditions que sur le 1^{er} dataset.

```

1 deb= time.time()
2 # On construit et on 'fit' le modèle
3 model_reglog = LogisticRegression(solver='liblinear')
4 model_reglog.fit(X_train_red, Y_train_red)
5 fin= time.time()
6 print(f"temps d'exécution: {fin-deb:.2f} s")

```

temps d'exécution: 144.15 s

/home/jpphi/anaconda3/lib/python3.8/site-packages/sklearn/svm/_base.py:1206: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
 warnings.warn(

Comme on peut le voir, l'algorithme a eu du mal à converger ! Les résultats ne sont d'ailleurs pas très bons.

```

# on teste sur les données de... test :)
prediction = model_reglog.predict(X_test_red)
erreur= 0
longueur= len(Y_test_red)
for i in range(longueur):
    if Y_test_red[i]!= prediction[i]: erreur+= 1
print(f"{erreur} erreurs sur {longueur} images soit une précision de {100*(1-erreur/longueur):.2f} %")

```

225 erreurs sur 1200 images soit une précision de 81.25 %

Pour améliorer les résultats, nous allons prendre un "solver" plus approprié, mais aussi normaliser les données. En effet, beaucoup d'algorithmes ont de bien meilleurs résultats avec des données qui ont été normalisées (même si celles-ci sont cohérentes entre elles puisqu'il s'agit de nombres entiers compris entre 0 et 255) avec l'une des méthodes proposées par scikit learn.

v Amélioration de la régression linéaire.

La première chose que l'on constate à la lecture de la documentation est que le solver "liblinear" n'est peut être pas le mieux adapté. On en testera donc un autre. D'autre part, la documentation de scikit learn donne une autre piste pour améliorer les résultats : La normalisation des données. Plusieurs méthodes existent (standardscaler, minmaxscaler, robustscaler pour ne citer que les plus connues).


```

1 # On récupère les données. On ne prendra qu'une partie des données situé dans le
2 # jeux de train. On en profite pour "aplatir" les images
3 data= train_x.reshape(train_x.shape[0], train_x.shape[1]*train_x.shape[2])
4
5 # On réduit le jeux de données original
6 data_red, _, Y_red, _ = train_test_split(data, train_y, train_size=0.1, random_state= 65)
7
8 # Standardisation
9 scaler=StandardScaler()
10 datastd= scaler.fit_transform(data_red)
11
12 # On split le jeux de données
13 X_train_red, X_test_red, Y_train_red, Y_test_red = train_test_split(datastd, Y_red, train_size=0.8,
14 random_state= 65)
15
16 print(f"Entrainement: X Shape: {X_train_red.shape}, Y shape: {Y_train_red.shape}\n"+\
17       f"Test: X Shape: {X_test_red.shape}, Y shape: {Y_test_red.shape}\n"+\
18       f"Etiquette: {etiquette}")

```

```

Entrainement: X Shape: (4800, 784), Y shape: (4800,)
Test: X Shape: (1200, 784), Y shape: (1200,)
Etiquette: [0 1 2 3 4 5 6 7 8 9]

```

Ensuite le choix d'un autre solver ; il faudra aussi tenir compte du fait que les classes ne contiennent pas le même nombre d'images.

```

1 deb= time.time()
2 # On construit et on 'fit' le modèle
3 model_reglog = LogisticRegression(solver='sag', class_weight= "balanced", max_iter= 2000, verbose= True)
4 model_reglog.fit(X_train_red, Y_train_red)
5 fin= time.time()
6 print(f"temps d'exécution: {fin-deb:.2f} s")

```

```

Epoch 1873, change: 0.00012927
Epoch 1874, change: 0.00012904
Epoch 1875, change: 0.00012909
Epoch 1876, change: 0.00012907
Epoch 1877, change: 0.00012884
Epoch 1878, change: 0.00012878
Epoch 1879, change: 0.00012875
Epoch 1880, change: 0.00012867
Epoch 1881, change: 0.00012849
Epoch 1882, change: 0.00012828
Epoch 1883, change: 0.00012828
Epoch 1884, change: 0.00012830
Epoch 1885, change: 0.00012804
Epoch 1886, chanmax_iter reached after 1366 seconds
temps d'exécution: 1366.36 s

```

```

/home/jpphi/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_sag.py:352: ConvergenceWarning: The max_iter
er was reached which means the coef_ did not converge
  warnings.warn(
[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed: 22.8min finished

```

La valeur par défaut du nombre en dessous duquel l'algorithme atteint son seuil de convergence est de $1e-4$. Comme on peut le voir ici, nous avons presque atteint ce chiffre.

La précision fait un bond de presque dix points ce qui est un bon résultat.

```

1 # on teste sur les données de... test :)
2 prediction = model_reglog.predict(X_test_red)
3 erreur= 0
4 longueur= len(Y_test_red)
5 for i in range(longueur):
6     if Y_test_red[i] != prediction[i]: erreur+= 1
7 print(f"{erreur} erreurs sur {longueur} images soit une précision de {100*(1-erreur/longueur):.2f} %")

```

```

125 erreurs sur 1200 images soit une précision de 89.58 %

```

```

1 target_names= ["détection de 0", "détection de 1", "détection de 2", "détection de 3", "détection de 4",
2               "détection de 5", "détection de 6", "détection de 7", "détection de 8", "détection de 9"]
3 print(f"Le score obtenu est de {model_reglog.score(X_test_red, Y_test_red)*100:.2f}")
4 print("\n",classification_report(Y_test_red,prediction, target_names= target_names))
5

```

Le score obtenu est de 89.58

	precision	recall	f1-score	support
détection de 0	0.91	0.94	0.93	126
détection de 1	0.90	0.99	0.94	117
détection de 2	0.92	0.80	0.85	137
détection de 3	0.84	0.88	0.86	112
détection de 4	0.90	0.91	0.90	123
détection de 5	0.88	0.77	0.82	103
détection de 6	0.94	0.97	0.96	135
détection de 7	0.92	0.91	0.92	119
détection de 8	0.84	0.85	0.85	110
détection de 9	0.89	0.92	0.90	118
accuracy			0.90	1200
macro avg	0.89	0.89	0.89	1200
weighted avg	0.90	0.90	0.89	1200

Pour espérer aller plus loin et améliorer le résultat un GridSearchCV sera effectué. On essaiera de trouver le meilleur algorithme de standardisation parmi les 3 principaux. On testera chacun des solvers composant l'algorithme de régression linéaire pour déterminer la meilleure des solutions.

vi GridSearchCV

On va construire un pipeline avec une première partie standardisation (c'est dans cette partie là que seront testés les 3 algorithmes de standardisation) et la seconde partie dans laquelle seul l'algorithme de régression logistique sera testé mais avec tous les solvers disponibles (à l'exception de liblinear qui a montré ses limites). Nous fixerons "class_weight" à "balanced", et laisserons les autres paramètres à leurs valeurs par défauts. C'est un compromis entre temps d'exécution et exhaustivité ! Et puis lorsque le meilleur "solver" sera trouvé on pourra toujours chercher à optimiser les autres paramètres même si cela ne garantit pas, qu'au final, la meilleure des solutions soit trouvée.

```

# On réalise un pipe avec 2 actions: un scaler et un algorithme
pipe = Pipeline(steps= [('scaler', StandardScaler()), ('algo', LogisticRegression())])

# On testera 3 standardisations et pour la régression logistique on testera tout les solvers
param_grid = {'pipeline__scaler': [StandardScaler(), MinMaxScaler(), RobustScaler()],
              'pipeline__algo__solver': ["sag", "newton-cg", "lbfgs", "saga"],
              'pipeline__algo__class_weight': ["balanced"],
              'pipeline__algo__max_iter': [2000]}
# instantiate and run as before:

model= make_pipeline(pipe)

grid = GridSearchCV(model, param_grid, cv=5, verbose= True, n_jobs= -1)

grid

GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('pipeline',
                                       Pipeline(steps=[('scaler',
                                                         StandardScaler()),
                                                         ('algo',
                                                         LogisticRegression()))])),
                                 n_jobs=-1,
                                 param_grid={'pipeline__algo__class_weight': ['balanced'],
                                             'pipeline__algo__max_iter': [2000],
                                             'pipeline__algo__solver': ['sag', 'newton-cg', 'lbfgs',
                                                                           'saga'],
                                             'pipeline__scaler': [StandardScaler(), MinMaxScaler(),
                                                                 RobustScaler()]},
                                 verbose=True)

```

Et, après 7heures 8 minutes et 37 secondes le meilleur choix standardisation/solver est :

```

deb= time.time()

grid.fit(X_train_red, Y_train_red)

fin= time.time()
print(f"temps d'exécution: {fin-deb:.2f} s")
/home/jpphi/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py:814: ConvergenceWarning: lbfgs
failed to converge (status=1):
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
Please also refer to the documentation for alternative solver options:
https://scikit-learn.org/stable/modules/linear\_model.html#logistic-regression
n_iter_i = _check_optimize_result(
/home/jpphi/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_sag.py:352: ConvergenceWarning: The max_it
er was reached which means the coef_ did not converge
warnings.warn(
/home/jpphi/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_sag.py:352: ConvergenceWarning: The max_it
er was reached which means the coef_ did not converge
warnings.warn(
/home/jpphi/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_sag.py:352: ConvergenceWarning: The max_it
er was reached which means the coef_ did not converge
warnings.warn(
temps d'exécution: 25717.32 s

```

```

grid.best_estimator_

Pipeline(steps=[('pipeline',
                 Pipeline(steps=[('scaler', MinMaxScaler()),
                                ('algo',
                                  LogisticRegression(class_weight='balanced',
                                                       max_iter=2000,
                                                       solver='newton-cg')))]))]

```

Le solver est "newton-cg" avec un MinMaxScaler pour la standardisation des données. Le gain de précision obtenu est décevant.

```

print(f"Précision obtenue: {100*grid.best_score_: .2f} %")

Précision obtenue: 88.40 %

```

Voyons ce que cela donne sur le jeu de test :

```

# on teste sur les données de... test :)
prediction = mod_prov.predict(X_test_red)

err, pre, d= erreur_prediction(valeurs_predites= prediction, valeurs_reelle= Y_test_red)
print(f"{err} erreurs sur {len(X_test_red)} images soit une précision de {pre:.2f} %")
print(f"Dictionnaire des erreurs (clé valeur réelle: valeur prédite):\n{d}")

111 erreurs sur 1200 images soit une précision de 90.75 %
Dictionnaire des erreurs (clé valeur réelle: valeur prédite):
{8: [3, 5, 5, 5, 3, 2, 5, 1, 5, 3, 1, 3, 1, 9], 2: [7, 8, 0, 7, 1, 3, 4, 7, 3, 1, 0, 1, 7, 6, 7, 6, 7, 3, 6, 8, 8,
8, 3], 5: [3, 3, 8, 3, 9, 4, 3, 0, 9, 3, 3, 9, 8, 0, 8, 8], 7: [5, 4, 3, 9, 9, 9, 9, 2, 5, 9, 9], 3: [8, 2, 5, 2, 0,
9, 0, 5, 5, 8, 7, 2, 8, 8], 6: [8, 5, 7, 5, 5], 4: [3, 2, 9, 8, 1, 9, 9, 8, 8, 8, 0], 0: [6, 5, 4], 9: [1, 4, 4, 8,
4, 7, 3, 3, 7, 8, 4, 7], 1: [3, 3]}

target_names= ["détection de 0", "détection de 1", "détection de 2", "détection de 3", "détection de 4",
               "détection de 5", "détection de 6", "détection de 7", "détection de 8", "détection de 9"]
print("\n",classification_report(Y_test_red,prediction, target_names= target_names))

```

	precision	recall	f1-score	support
détection de 0	0.95	0.98	0.96	126
détection de 1	0.93	0.98	0.96	117
détection de 2	0.95	0.83	0.89	137
détection de 3	0.83	0.88	0.85	112
détection de 4	0.93	0.91	0.92	123
détection de 5	0.86	0.84	0.85	103
détection de 6	0.97	0.96	0.97	135
détection de 7	0.91	0.91	0.91	119
détection de 8	0.83	0.87	0.85	110
détection de 9	0.88	0.90	0.89	118
accuracy			0.91	1200
macro avg	0.91	0.91	0.91	1200
weighted avg	0.91	0.91	0.91	1200

Comme attendu le gain est modeste 90,75 % contre 89,58 % sur le jeu de test !

Je vois plusieurs explications possibles à cela.

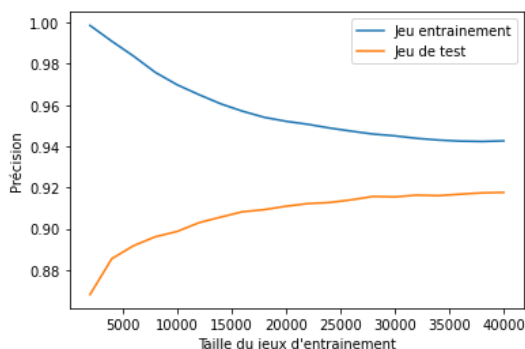
- La première c'est qu'avec GridSearchCV on ne se contente pas d'un seul calcul mais de 5 (paramètre "cv") et une moyenne des résultats est ensuite effectuée. Comme nous l'avons vu dans les conclusions de la première étude avec le dataset fourni par scikit learn, les résultats peuvent varier de façon significative lorsque l'on change d'échantillon (une différence de 2 % avait été constatée).
- La seconde est que l'algorithme atteint ses limites. Quelque soit les valeurs données nous n'obtiendrons jamais de meilleurs résultats. Il faudra alors se tourner vers d'autres algorithmes.
- La troisième hypothèse c'est que nous n'avons pas pris assez de données d'entraînement. La fonction learning curve de scikit learn peut permettre de vérifier (ou non) cette hypothèse.

vii Learning curve, testons l'hypothèse 3.

Pour tester cette hypothèse on prend le meilleur algorithme que l'on entraîne sur la totalité du jeu de train. De même la totalité du jeu de test sera utilisée.

```
train_size, train_score, val_score = learning_curve(pipe_minmax_logreg, data_train, train_y,
                                                    train_sizes = np.linspace(0.05, 1, 20), cv = 3)
```

```
plt.plot(train_size, train_score.mean(axis=1), label = "Jeu entraînement")
plt.plot(train_size, val_score.mean(axis=1), label = "Jeu de test")
plt.xlabel("Taille du jeux d'entraînement")
plt.ylabel("Précision")
_ = plt.legend()
```



Comme on le peut le voir, la précision sur le jeu de test à une allure de faux plat montant que détestent bon nombre de cyclistes ! La réduction du jeu de données a permis d'accélérer les calculs pour la recherche du meilleur algorithme sans que cela ne nuise au résultat.

viii Conclusion.

Il est probable que la régression logistique ait atteint un plafond que l'on peut estimer à 2 ou 3 points au dessus du niveau atteint. Il serait donc judicieux de se tourner vers d'autres algorithmes si l'on souhaite améliorer encore les résultats. En effet, il ne faut pas oublier que même les algorithmes avec un résultat flatteur de plus de 99 % peuvent décevoir sur des cas concrets. Améliorer la précision serait bienvenue ! Parmi tous les classificateurs, le KNN semble être un bon candidat. Mais on peut aussi décider de se tourner vers une autre technologie : Les réseaux de neurones. Un type particulièrement prisé dans la classification d'images est le réseau de neurone convolutif !

Dernier point, et pour pouvoir comparer les différentes méthodes, un modèle entraîné sur la totalité du jeu d'entraînement de MNIST et validé sur la totalité du jeu de test a été enregistré dans le répertoire "modeles". Les données présentées à cet algorithme doivent être des tableaux numpy à 1 dimension (utilisation de ravel) avec des nombres flottants compris entre 0 et 1.

b Amélioration en testant de nouveaux algorithmes.

Référentiel : C8. Modifier les paramètres et composants de l'intelligence artificielle afin d'ajuster aux objectifs du projet les capacités fonctionnelles de l'algorithme à l'aide de techniques d'optimisation.

i Test du classifieur KNN.

Après quelques tests d'optimisation avec la fonction GridSearchCV et en gardant la standardisation avec le MinMaxScaler le classifieur KNN montre une rapidité plus grande et une meilleure précision.

```
# On choisi les meilleurs paramètres en gardant la standardisation min/max
pipe_minmax_knn= make_pipeline(MinMaxScaler(), KNeighborsClassifier(n_neighbors= 6, weights= "distance",
                                                                    algorithm='ball_tree', p= 3))

pipe_minmax_knn.fit(data_train, train_y)
prediction = pipe_minmax_knn.predict(data_test)
```

Et le résultat est très encourageant !

```
err, pre, d= erreur_prediction(valeurs_predites= prediction, valeurs_reelle= test_y)
print(f"{err} erreurs sur {len(data_test)} images soit une précision de {pre:.2f} %")
print(f"Dictionnaire des erreurs (clé valeur réelle: valeur prédite):\n{d}")

263 erreurs sur 10000 images soit une précision de 97.37 %
Dictionnaire des erreurs (clé valeur réelle: valeur prédite):
{4: [0, 9, 6, 1, 9, 9, 9, 6, 9, 1, 9, 8, 9, 9, 9, 6, 6, 9, 9, 9, 1, 7, 1, 9, 7, 1, 1, 9, 9], 3: [5, 7, 7, 5, 5, 9,
7, 4, 5, 2, 5, 5, 5, 7, 7, 5, 7, 5, 5, 8, 2, 7, 6, 8, 9, 5, 9, 5, 9, 5, 5, 9, 8], 9: [8, 7, 8, 4, 0, 3, 5, 3, 4, 2,
6, 1, 4, 7, 0, 7, 1, 7, 4, 1, 1, 3, 3, 2, 7, 7, 2, 4, 1, 4, 5, 3, 5, 0, 0, 0, 7, 7, 4], 2: [7, 8, 7, 6, 7, 7, 0, 7,
0, 7, 0, 7, 1, 0, 7, 6, 1, 1, 3, 7, 7, 3, 7, 7, 7, 0, 7, 0, 1, 7, 0, 0, 0, 8, 7, 8, 0], 6: [4, 0, 5, 0, 1, 4, 5, 1,
0, 4, 0, 1, 0], 7: [4, 1, 4, 2, 1, 1, 1, 9, 9, 1, 1, 1, 1, 9, 1, 1, 1, 1, 9, 9, 9, 9, 1, 1, 1, 9, 9, 9, 9, 1, 1, 2,
2, 2], 8: [0, 1, 3, 3, 3, 9, 4, 3, 3, 7, 4, 5, 3, 2, 4, 2, 3, 3, 6, 6, 3, 5, 0, 0, 5, 2, 7, 9, 3, 7, 6, 0, 6, 2, 4,
9, 9, 7, 9, 4, 5, 5], 5: [9, 4, 3, 6, 9, 7, 3, 6, 8, 8, 3, 9, 9, 0, 8, 4, 6, 3, 6, 9, 6, 6, 0, 6, 0, 6], 1: [2, 2,
6], 0: [6, 2, 7, 5, 6, 1, 6]}
```

Avec la régression logistique optimisée le nombre d'erreurs était de 737 pour 10 000, avec le classifieur KNN optimisé le nombre d'erreurs passe à 263 ce qui est un excellent résultat. La précision grimpe à 97,37 %.

Il est clair que le classifieur est un excellent choix pour ce type de données et pour ce dataset en particulier. Cependant, dans la classification d'image les réseaux de neurones, et particulièrement les CNN semblent faits pour ce type de données.

ii Test avec un réseau de neurones.

La création d'un réseau de neurones de type CNN n'est pas, à proprement parler, difficile. Ce qui est plus compliqué c'est d'en faire un suffisamment "léger" pour qu'il puisse être entraîné facilement tout en étant efficace.

La phase d'apprentissage est une étape clef. Comme pour tous les algorithmes de machine learning, il faut s'assurer aussi qu'il ne soit pas sur-entraîné, perdant ainsi sa capacité de généralisation. On tracera les courbes d'apprentissage en regardant les courbes de perte et de précision sur le jeu d'entraînement, comme sur le jeu de test pour s'assurer que l'on ne soit pas en "overfitting". Le sur-apprentissage peut en effet se repérer quand l'évolution des courbes de perte et de précision sur les 2 jeux divergent.

Plusieurs réseaux ont été créés et leurs performances comparées sur des chiffres manuscrits qui ne sont pas issus de MNIST.

Sans présenter ici tout les réseaux entraînés (ils se trouvent dans le Jupyter notebook en annexe de ce document), voici les étapes suivies :

Conception du réseau :

```
batch_size = 128
epochs = 250

input_shape= (train_x.shape[1], train_x.shape[2], 1)
nb_classe= len(etiquette)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),activation='relu',input_shape= input_shape ))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classe, activation='softmax'))
model.compile(loss= keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.summary()
```

Entraînement et évaluation du réseau :

```
hist = model.fit(data_train_pour_cnn, train_y_conv, batch_size=batch_size, epochs=epochs,
                verbose=1, validation_data=(data_test_pour_cnn, test_y_conv), workers= -1)
```

```
score = model.evaluate(data_test_pour_cnn, test_y_conv, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

Test loss: 0.05704181641340256
Test accuracy: 0.9825000166893005

Courbes d'apprentissage :

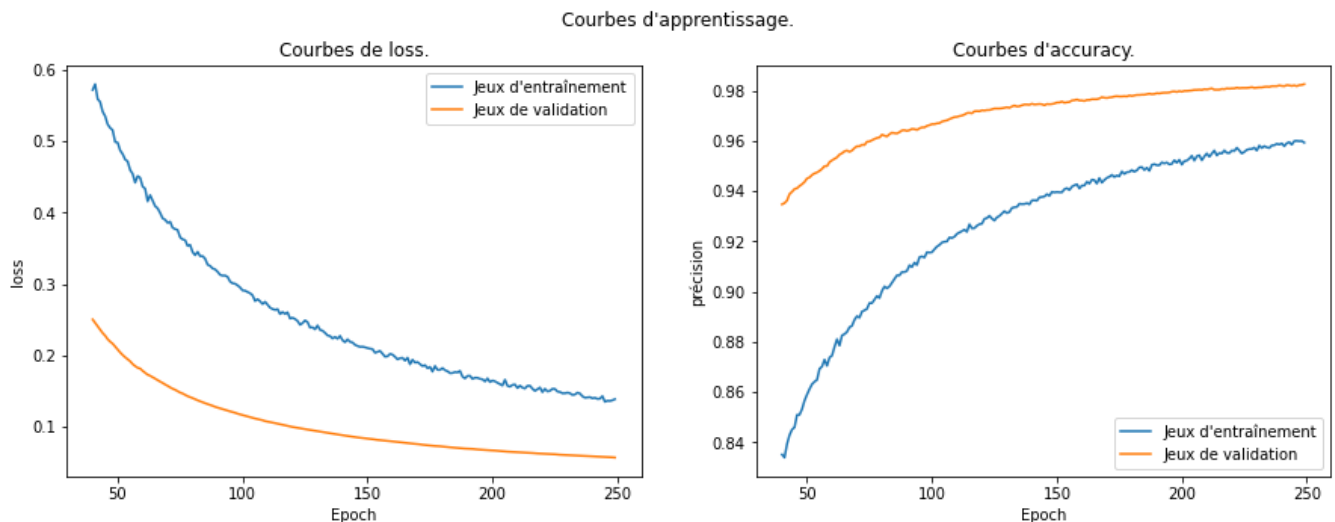
En "zoomant" sur la partie finale pour s'assurer qu'il n'y a pas de phénomène de "sur-apprentissage" qui se révèle en général quand les évolutions des courbes (du jeu d'entraînement et du jeu de validation) divergent.


```
x= range(len(hist.history["loss"]))
debut= 40

fig, axes= plt.subplots(1, 2, figsize= (15,5))
fig.suptitle("Courbes d'apprentissage.")

axes[0].plot(x[debut:], hist.history["loss"][debut:], label="Jeux d'entraînement")
axes[0].plot(x[debut:], hist.history["val_loss"][debut:], label="Jeux de validation")
axes[0].set_title("Courbes de loss.")
axes[0].set_xlabel= "Epoch", ylabel="loss")
axes[0].legend()

axes[1].plot(x[debut:], hist.history["accuracy"][debut:], label="Jeux d'entraînement")
axes[1].plot(x[debut:], hist.history["val_accuracy"][debut:], label="Jeux de validation")
axes[1].set_title("Courbes d'accuracy.")
axes[1].set_xlabel= "Epoch", ylabel="précision")
_ = axes[1].legend()
```



c Conclusion de l'étude sur les 3 modèles proposés

Le premier algorithme a les moins bonnes performances mais son modèle binaire est plutôt léger (environ 65 Ko).

Le second algorithme utilisant le KNN est beaucoup plus performant au prix d'un modèle particulièrement lourd (770 Mo). Nous gagnons en précision, mais au prix d'un besoin en ressource machine important.

Le dernier algorithme à base d'un réseau neuronal de type CNN est à la fois excellent en terme de résultat sans, pour autant, avoir un besoin en ressource machine comparable à celui du KNN. Le modèle entraîné ne pèse "que" 26 Mo (pour le premier réseau entraîné).

3 Ajout de nouvelles fonctionnalités

Référentiel : C14. Améliorer l'application d'intelligence artificielle en développant une évolution fonctionnelle pour répondre à un besoin exprimé par un client ou un utilisateur.

La précision obtenue est plutôt flatteuse (au delà des 98%) mais il s'agit de données issues de la même base, et l'expérience montre qu'il faut savoir être prudent tant que l'on ne valide pas ces résultats dans le monde réel.

On peut se donner comme objectif de réaliser un programme utilisant un navigateur internet pour saisir des chiffres dessinés par un vrai humain. On pourra alors tester plusieurs réseaux de neurones différents sur des cas concrets et se faire une idée plus précise de l'utilisation possible de ces algorithmes.

a Cahier des charges :

- Réaliser un programme utilisant un navigateur et qui pourrait être, dans un second temps, déployé.
- Définir une zone de dessin suffisamment grande pour que l'utilisateur puisse dessiner un chiffre.
- Prévoir une zone de sélection de divers modèles pouvant être testés
- Afficher la prédiction issue de l'algorithme.
- Écrire le programme en python.
- Livrable : Le ".py" ainsi que toute ses dépendances (y compris les modèles utilisés pour la prédiction).
- Prévoir un jeu de test pour réaliser des tests de non régression (même si cela est délicat sur des prévisions basées sur des probabilités).


b Test du programme final "lecture_cm.py" :

Les différents tests menés sur des chiffres dessinés sur le canva montre que nous sommes loin des 98 % affichés par les modèles à base de CNN et même des 92 % à 97 % des modèles de régression logistique et de KNN.

Voici un exemple de prédiction avec 2 modèles différents indiquant 4 pour la prédiction de l'un (correct) et 9 pour l'autre (Comment peut on voir un 9 ? Le charme des CNN et de la décomposition d'image).

Choix du modèle d'algorithme:

KNN



↓ ↶ ↷ 🗑️


Prédiction

4

Chiffre lu: 4

Choix du modèle d'algorithme:

CNN-32-64-F-256-Do



↓ ↶ ↷ 🗑️

Prédiction

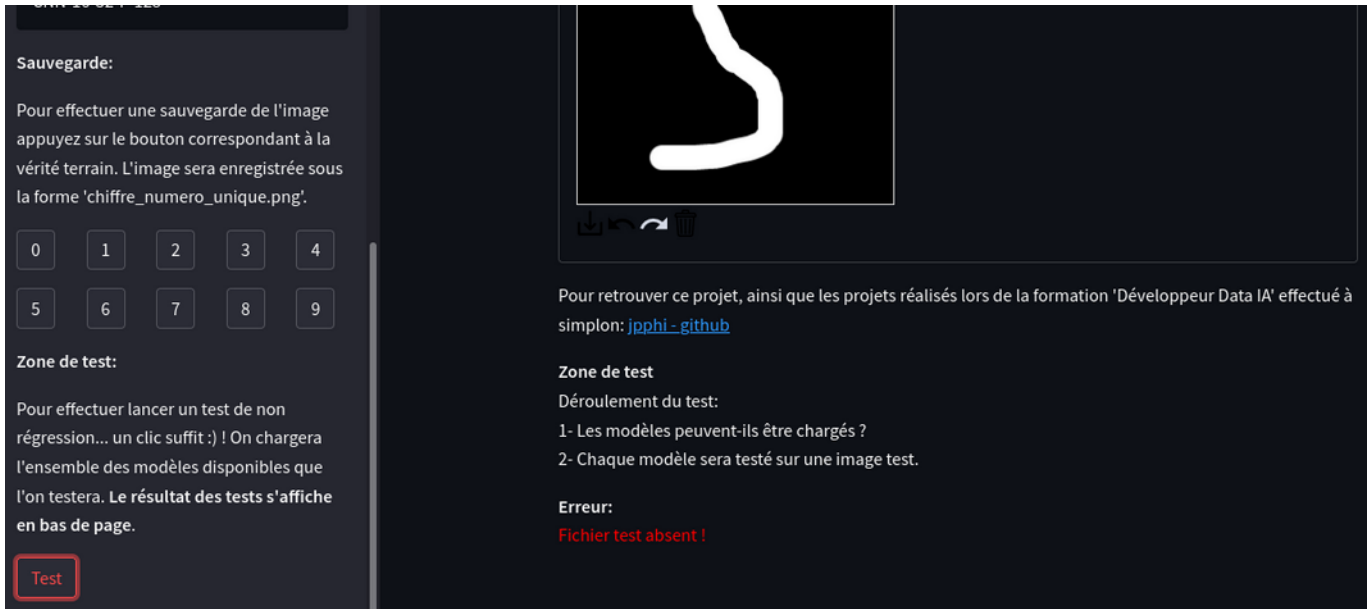
4

Chiffre lu: 9

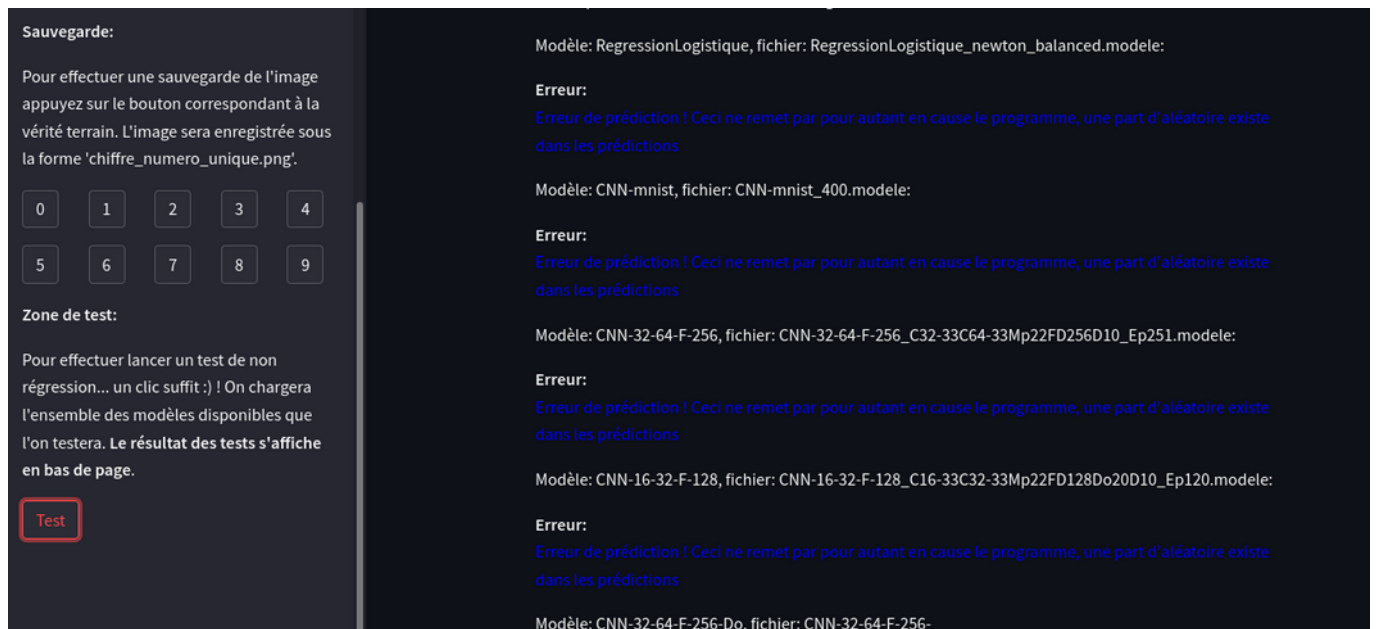
4 Test de fonctionnalité et de non régression.

Une série de tests a été réalisée. Le chargement des algorithmes est testé (test de fonctionnalités), ainsi que leurs réponses à une image test représentant un 5 (test de non régression) et qui est normalement reconnue par tous les modèles. Cependant si un modèle ne reconnaît pas ce chiffre un message s'affichera. Pour le code, voir le fichier "lecture_cm.py" en Annexe.

Voici un exemple d'erreur : Ici le fichier contenant l'image test est absent !



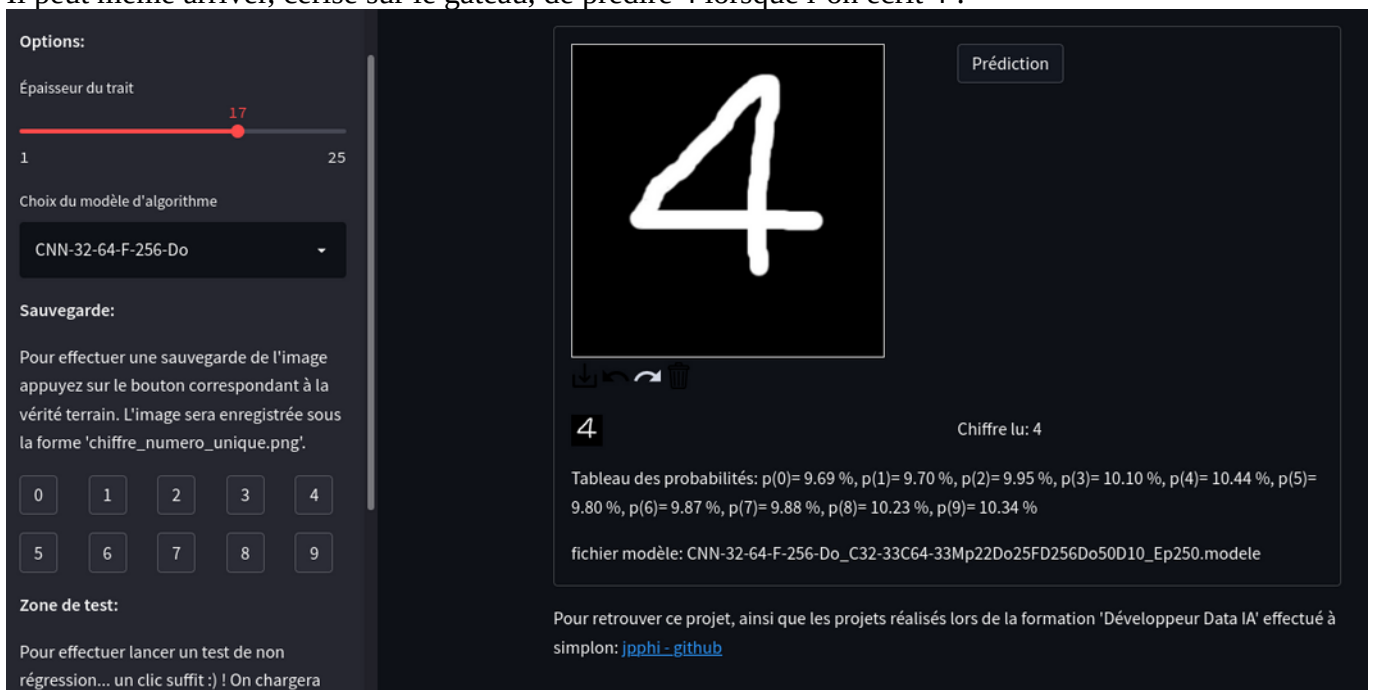
Lorsque l'image test n'est pas reconnue, il ne s'agit pas à proprement parlé d'une erreur ; mais si l'ensemble des algorithmes ne reconnaissent pas l'image, il faut s'en inquiéter !



Et il peut même arriver que les tests se passent bien et sans erreur !



Il peut même arriver, cerise sur le gâteau, de prédire 4 lorsque l'on écrit 4 !



5 Conclusion.

Comme on pouvait s'y attendre, il y a toujours une différence significative entre théorie et réalité. Pour autant, ce programme est fonctionnel et pourrait même évoluer en intégrant des fonctionnalités permettant d'utiliser les données issues des dessins pour compléter l'apprentissage réalisé sur la base MNIST. Pour plus de fiabilité, on pourrait faire la détection sur plusieurs modèles ce qui diminuerait le nombre de "faux positifs" mais augmenterait le nombre de "faux négatifs".

Une piste d'amélioration pourrait-être d'utiliser l'analyse en composante principale (PCA). En effet la majorité des pixels compose le fond de l'image et n'apporte pas d'information. En fait on peut même dire que la seule information intéressante est contenue dans les pixels formant le bord du chiffre. C'est d'ailleurs ce que j'espérais des CNN ; qu'ils arrivent à extraire les différentes petites zones du dessin représentant le chiffre.

6 Annexes .

a Jupyter notebook :

C'est le fichier avec lequel a été réalisé la mise au point, l'optimisation ainsi que l'ensemble des tests des différents modèles de machine learning. Si la totalité du code est fournie, la sortie des cellules de code est amputée car le document aurait comporté plus de 60 pages.

b Programme python :

Ce programme est une interface web réalisée grâce à la librairie streamlit. Il permet de dessiner un chiffre (l'épaisseur du trait est paramétrable) puis d'utiliser plusieurs modèles (à choisir dans un boîte de sélection) pour le "décoder". Il permet également d'enregistrer le dessin "avec la vérité terrain" pour constituer des données d'entraînement. Enfin un programme de test de non régression est intégré au programme.

Pour lancer le programme, s'assurer que les dépendances soient installées (sur ce pc, se placer dans **l'environnement prjfe**) puis lancer la commande **streamlit run lecture_cm.py** .

