

# Rapport de TPs - Apprentissage Supervisé (KNN-ANN-SVM)

Auteur : Anaïs RABARY (5SDBD - Gr A)

Date : 7 Décembre 2018

NOTES : Le beau rapport avec un plan et des images dimensionnées ainsi que les codes des TP 1, 2 et 3 sont disponibles sur ce git : [https://github.com/anaisrabary/Tps\\_ApprentissageSupervise](https://github.com/anaisrabary/Tps_ApprentissageSupervise) ([https://github.com/anaisrabary/Tps\\_ApprentissageSupervise](https://github.com/anaisrabary/Tps_ApprentissageSupervise)) .

## Introduction

Dans le cadre des Travaux Pratiques sur l'Apprentissage Supervisé réalisés en 5ème Année SDBD (Système Distribués Big Data), sous la supervision de M-V. Le Lann et M. Siala, nous avons étudiés différentes méthodes d'apprentissage supervisé : KNN, K-Nearest Neighbors ou les algo des K plus proches voisins, ANN, Artificial Neural Network ou Réseau Artificiel de Neurones et SVM, Support Vector Machine ou machine à support de vecteur. L'étude est réalisée à partir de l'outil Scikit-Learn sur python (<https://scikit-learn.org/>).

L'objectif de ce rapport est de présenter ces 3 méthodes puis de les comparer. Ce rapport se veut pédagogique pour former un support personnel qui pourra être réutilisé pour des études futures. On présentera notamment les paramètres entrant dans l'étude de comparaison. Les codes exécutés pour réaliser l'étude seront rendus dans leur totalité dans les autres notebook. Je n'ai pas jugé nécessaire de surcharger ce rapport et d'entraver sa bonne lisibilité.

Ces 3 méthodes ont pour but de classifier les données. Afin de pouvoir les évaluer, nous travaillerons sur un même jeu de données, [MNIST](http://yann.lecun.com/exdb/mnist/) (<http://yann.lecun.com/exdb/mnist/>), base de données ouvertes de chiffres écrits à la main. Dans ce cas, le but sera d'arriver à déterminer les 10 classes de chiffres. Les éléments de comparaison dépendront des méthodes, mais nous analyserons à chaque fois leurs performances et leur temps d'exécution. On s'attardera aussi sur les choix des paramètres permettant d'obtenir les meilleurs résultats pour la classification sur le jeu de données mnist.

## PLAN

### Introduction

#### 1. Présentations générales

##### 1.1 Le Dataset

##### 1.2 KNN

##### 1.3 ANN

##### 1.4 SVM

## 2. Etudes séparées

### 2.1 Méthodes de vérification

### 2.2 Indicateurs de performance

### 2.3 Choix des paramètres de l'algorithme

#### 2.3.1 Mnist et KNN

#### 2.3.2 Mnist et les neurones

#### 2.3.3 Mnist et SVM

## 3. Analyse comparative

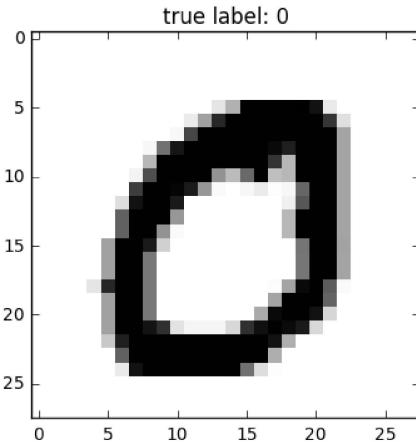
### Conclusion

# 1. Présentations générales

## 1.1 Le Dataset

Le dataset utilisé pour la comparaison des algorithmes, les chiffres de 0 à 9 écrits à la main, contient 70 000 images labélisées. Ces images font 28x28 pixels, et donc comportent 784 pixels. L'objectif des algorithmes développés ici est de pouvoir classifier ces images et ainsi pouvoir prédire le chiffre d'une image non labélisée. Ci-après voici un exemple d'affichage des données mnist.

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2  
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3  
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4  
5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5  
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6  
7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7  
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8  
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9



Ce jeu de données a un but pédagogique. Les 70000 images sont donc toutes labélisées. On veillera donc à séparer notre jeu en 2 parties : une partie pour l'apprentissage et une partie pour le test. Grâce aux labels des données de tests, on pourra mesurer le score de notre modèle. Le jeu comprends plusieurs attributs dont notamment data et target qui vont nous être utiles pour apprendre, tester et vérifier nos models:

- data, un tableau de n instances x m attributs

- target, le label associé à chaque instance.

## 1.2 KNN

On rappelle ici les principes de KNN : L'algorithme des k-plus proches voisins, ou K Neighbors, est un algorithme simple et intuitif. Il est utilisé pour des problèmes de classification. Pour chaque objet  $o$  que l'on souhaite classifier, on cherche ses  $K$  plus proches voisins connus (labellisés). Puis on associe à  $o$  le label qui est majoritaire parmi ses voisins.

On effectue les tests en utilisant le modèle de scikit-learn, `sklearn.neighbors`, qui est paramétrable.

Pour trouver les  $K$  plus proches voisins, l'algorithme KNN repose sur une mesure de distance.

Cependant, il existe plusieurs types de distances différentes :

- Euclidienne :  $\sqrt{\sum(x - y)^2}$
- Manhattan :  $\sum |x - y|$
- Chebyshev :  $\max(|x - y|)$
- Minkowski :  $(\sum |x - y|^p)^{1/p}$
- Hamming :  $\sum |x - y|$  avec  $x = y \implies D = 0$  et  $x \neq y \implies D = 1$

Par défaut, c'est la distance euclidienne qui est utilisée. Une étude comparative de ces distances est réalisée dans la partie 2.3.1.

Voici ci-après le squelette de code qui servira à l'étude des performances de KNN sur Mnist. On rappelle que la totalité du code ayant servi à l'étude de KNN est donnée dans le notebook dédié.

```

1 #Imports Liés à KNN et Mnist
2 from sklearn import datasets
3 import numpy as np
4 from sklearn.model_selection import train_test_split
5 from sklearn import neighbors
6 #Imports pour la mesure de performance
7 import matplotlib.pyplot as plt
8 from sklearn import metrics
9 import time
10
11 # Charger les données
12 mnist=datasets.fetch_mldata('MNIST original')
13
14 #ECHANTILLONS DE 10000
15 indices = np.random.randint(70000, size=10000)
16 data = mnist.data[indices]
17 target = mnist.target[indices]
18 #pour séparer le dataset en 2 échantillons de training et de test
19 #Data fait donc 8000 et test 2000
20 xtrain, xtest, ytrain, ytest =train_test_split(data, target, train_size=0.8)
21
22 # Classifier KNN, apprentisage, prédiction et score
23 clf=neighbors.KNeighborsClassifier(10)
24 clf.fit(xtrain,ytrain)
25 predict = clf.predict(xtest)
26 score = clf.score(xtest,ytest)

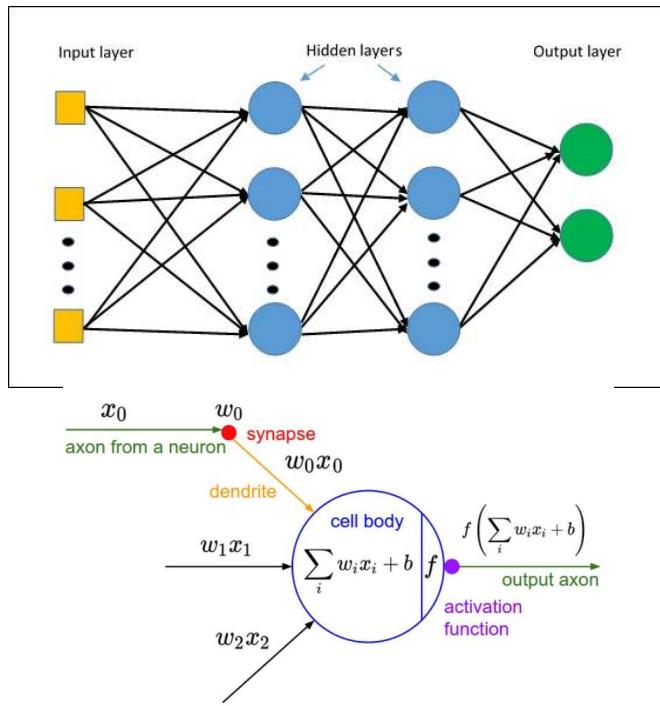
```

## 1.3 ANN

On présente ici un réseau de neurones multicouches, perceptron multi-couche (MLP, Multi-layered perceptron ou Artificial Neural Network). On peut distinguer 3 parties :

- les entrées : permettent de prendre un vecteur à plusieurs dimensions (une dimension par neurone),
- les sorties : présentent le résultat,
- les couches cachées (hidden layers) : permettent de faire une séparation entre les données, par combinaison linéaire.

Sur les liens entre chaque neurones se trouvent des **poids**  $w_0, w_1, w_2$  (weight). C'est ces poids qui sont adaptés pendant l'apprentissage. Dans un neurone, on calcule la somme pondérée de tous les signaux entrants,  $\sum(w_i \dot{x}_i)$ . Puis on y ajoute un **biais**  $b$ , qui permet de décaler la fonction d'activation. En sortant du neurone, le signal calculé passe par une **fonction d'activation** qui laisse passer un signal suivant un seuil. On a enfin le signal de sortie du neuronne.



Lors de la création d'un tel réseau, les poids et biais sont initialisés aléatoirement. Puis c'est par une méthode de backpropagation qu'ils sont recalculés jusqu'à la fin de l'apprentissage.

Le modèle proposé par scikit learn est paramétrable.

Pour les neurones dans les couches cachées, il y a plusieurs fonction d'activation possibles :

- identity,  $f(x) = x$  donc pas de fonction d'activation,
- logistic,  $f(x) = 1/(1 + \exp(-x))$ , la fonction d'activation du sigmoid,
- tanh,  $f(x) = \tanh(x)$ , la fonction hyperboliques de tan,
- relu,  $f(x) = \max(0, x)$ , la fonction de rectification linéaire unitaire. C'est relu qui est utilisé par défaut.

On peut aussi modifier le solver utilisé pour calculer les poids :

- lbfgs, un optimiseur de la famille des méthodes quasi Newtonienne,
- sgd, la méthode de descente de gradient stochastic,
- adam, une autre méthode de descente de gradient, optimisée par Kingma, Diederik et Jimmmy Ba. Par défaut, c'est adam qui est utilisé.

Il est aussi possible de paramétriser alpha  $\alpha$ , qui est le paramètre de régularisation dans la pénalité L2. Il permet de prévenir l'overfitting, en contrignant la taille des poids.

Choisir une architecture de réseau de neurone demande une certaine expertise. Il faut essayer différentes architectures et choisir le réseau donnant le meilleur résultat de généralisation sur les tests.

Voici ci-après le squelette de code qui servira à l'étude des performances de ANN sur Mnist. On rappelle que la totalité du code ayant servi à l'étude de ANN est donnée dans le notebook dédié.

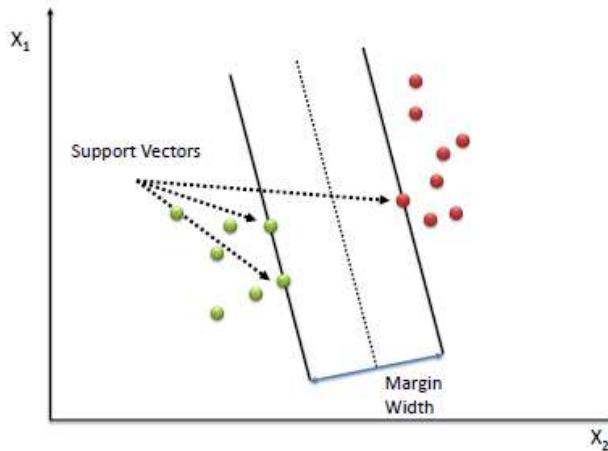
```

1 #import nécessaires
2 import numpy as np
3 from sklearn import datasets
4 from sklearn.model_selection import train_test_split
5 from sklearn.neural_network import MLPClassifier
6 from sklearn import metrics
7 import time
8
9 #charger le jeu de données MNIST
10 mnist=datasets.fetch_mldata('MNIST original')
11
12 #randomise Data et target
13 indices = np.random.randint(70000, size=70000)
14 data = mnist.data[indices]
15 target = mnist.target[indices]
16 # on veut un training set de 49000
17 xtrain, xtest, ytrain, ytest =train_test_split(data, target, train_size=49000)
18
19 clf = MLPClassifier(hidden_layer_sizes=(50))
20 clf.fit(xtrain, ytrain)
21 predict = clf.predict(xtest)
22 score = clf.score(xtest,ytest)
23
24 recall = metrics.recall_score(ytest, predict, average ='macro')
25 precision = metrics.precision_score(ytest, predict, average='macro')
26 loss01 = metrics.zero_one_loss(ytest, predict)
27
28
29 print("Ce modèle de MLP, d'1 couche de 50, à un score de ", score*100, "%.")
30 print("4eme image : prédiction ",predict[3], "reel : ", ytest[3])
31 print ("ce modèle de MLP à une précision de", precision*100, "%.")
32 print ("ce modèle de MLP à un recall de",recall*100, "%.")
33 print ("ce modèle de MLP à un zero-one_loss de",recall*100, "%.")
34 print("    temps apprentissage : ", timetrain, "sec , temps prediction = ", ti
35
36
37 #Ce modèle de MLP, d'1 couche de 50, à un score de  98.5047619047619 %.
38 #4eme image : prédiction 5.0 reel : 5.0
39 #ce modèle de MLP à une précision de 98.50503688809177 %.
40 #ce modèle de MLP à un recall de 98.49104431028593 %.
41 #ce modèle de MLP à un zero-one_loss de 98.49104431028593 %.
42 #    temps apprentissage :  322.9891335964203 sec , temps prediction =  0.6702

```

## 1.4 SVM

La méthode d'apprentissage par Machine à Vecteurs de Support ou SVM est la dernière méthode que nous allons voir dans ce rapport. Le but de SVM est de maximiser la marge de séparation entre les classes. Cela permet en fait de séparer linéairement nos données en augmentant la dimension. La méthode permet de trouver un hyperplan qui sépare nos données, tout en maximisant les marges.



Nous allons utiliser SVC de Scikit-learn.

Il est possible de faire varier plusieurs paramètres. **Le kernel :**

- linéaire
- poly
- rbf
- sigmoid Par défaut c'es RBF qui est utilisé

Il est aussi possible de faire varier **le coût et gamma**.

Voici ci-après le squelette de code qui servira à l'étude des performances de SVM sur Mnist. On rappelle que la totalité du code ayant servi à l'étude de SVM est donnée dans le notebook dédié.

Entrée [ ]:

```
1 #import nécessaires
2 import numpy as np
3 from sklearn import datasets
4 from sklearn.model_selection import train_test_split
5 from sklearn.model_selection import GridSearchCV
6 from sklearn.svm import SVC
7 from sklearn import metrics
8 import matplotlib.pyplot as plt
9 import numpy as np
10 import time
11
12 #charger le jeu de données MNIST
13 mnist=datasets.fetch_mldata('MNIST original')
14
15 #randomise Data et target
16 indices = np.random.randint(10000, size=10000)
17 data = mnist.data[indices]
18 target = mnist.target[indices]
19 #pour séparer le dataset en 2 échantillons de trainint et de test
20 # on veut un training set de 49000 (70% training set - 30 % test set)
21 xtrain, xtest, ytrain, ytest =train_test_split(data, target, train_size=int(le
22
23 clf = SVC(kernel='rbf')
24 startTrain =time.time()
25 clf.fit(xtrain, ytrain)
26 endTrain = time.time()
27 # PREDIC
28 startpred= time.time()
29 predict = clf.predict(xtest)
30 endpred = time.time()
31 # METRICS
32 score = clf.score(xtest,ytest)
33 recall = metrics.recall_score(ytest, predict, average ='macro')
34 precision = metrics.precision_score(ytest, predict, average='macro')
35 loss01 = metrics.zero_one_loss(ytest, predict)
36 timetrain = endTrain - startTrain
37 timePred = endpred - startpred
38
39 print("Ce modèle SVC avec un kernel", ker, "a un score de ", score*100, "%.")
40 print("4eme image : prédition ",predict[3], "reel : ", ytest[3])
41 print ("précision :", precision*100)
42 print ("recall :",recall*100)
43 print ("zero-one_loss :",recall*100)
44 print( "training time :", timetrain)
45 print( "prediction time :", timePred)
46
```

## 2. Etudes séparées

### 2.1 Méthode de vérification

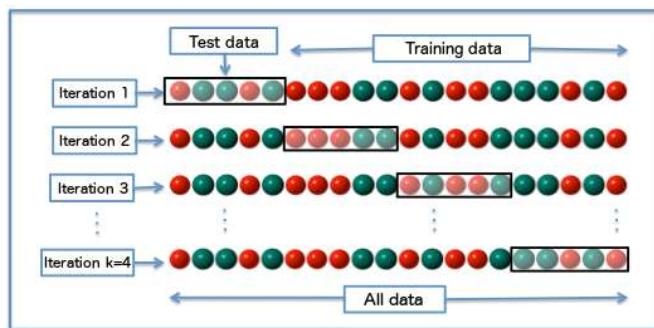
Il existe 3 grandes méthodes de vérifications :

- Test set validation (holdout method)
- K-fold cross-validation
- Leave-one-out cross-validation

La 1ère, Test Set validation est la plus simple. Elle consiste à séparer le jeu de données en 2. Une partie est réservée pour l'apprentissage, l'autre pour le test et donc la validation du modèle. Cela peut être par exemple, réserver 70% des données pour l'apprentissage et les 30% restants pour les tests.

Cependant, si l'on changeait nos de tests, mais qu'on gardait le même modèle, le score serait sûrement différent. De même, en choisissant un modèle dont l'apprentissage repose sur les mêmes données, sommes nous certains que l'apprentissage aurait été le même pour d'autres données ? Il est vrai que toutes les données servant à tester ne font pas parties des données d'apprentissage.

Voilà tout l'intérêt des 2 prochaines méthodes qui reposent sur la même idée. Le but est d'apprendre et tester notre modèle sur toutes les données. Dans le cas du K-fold cross validation, on divise le jeu de données en  $K$  parties égales. Puis on définit notre test set comme étant l'une des  $K$  parties. Les autres  $K - 1$  parties sont regroupées pour former le training set. On fait apprendre notre classifieur. Puis on change de test set en prenant la partie suivante et on réitère  $K$  fois jusqu'à ce que toutes les parties aient été un test set.



Il est recommandé de faire un 10-fold cross validation. La méthode du leave-one out cross validation correspond à la méthode du K-fold cross validation, poussé à l'extrême, pour  $k = n$ . En effet, à chaque itération, le test set est composé d'une seule data et le training set est composé de  $n - 1$  data.

## 2.2 les indicateurs de performance

On peut utiliser différents indicateurs pour mesurer la performance des algorithmes.

**Le score** nous donne tout simplement le pourcentage de réussite de classification des données de test. Il représente en fait la moyenne exacte des data prédites  $x_{test}$  en fonction de leur label  $y_{test}$ .

**Le recal ou sensibilité** renvoie le ratio  $Tp/(Tp + Fn)$  où  $Tp$ , true positive et  $Fn$ , false negative. Il représente l'abilité du classifieur de trouver les exemples positifs. C'est la probabilité de détection.

**La précision** retourne le ratio  $Tp/(Tp + Fp)$  où  $Fp$  false positive. Cet indicateur présente la capacité d'un classifieur à ne pas labéliser positivement un label qui est négatif.

**L'erreur zero-to-one loss**, retourne le ratio d'exemples mal classifiés. La meilleure performance. Il me semble que le zero-to-one loss est en fait  $1 - score$ .

**La mean squared error**,  $MSE(y_{pred}, y_{label}) = \frac{1}{n_{samples}} \sum (y_{labeli} - y_{predi})^2$ .

### Les temps de training et de prediction.

Note : pour le recall et la précision, il faudra setter le paramètre average à macro et non micro.

Voici ci-après un exemple de code avec les métriques utilisées, sur un exemple de KNN :

Entrée [ ]:

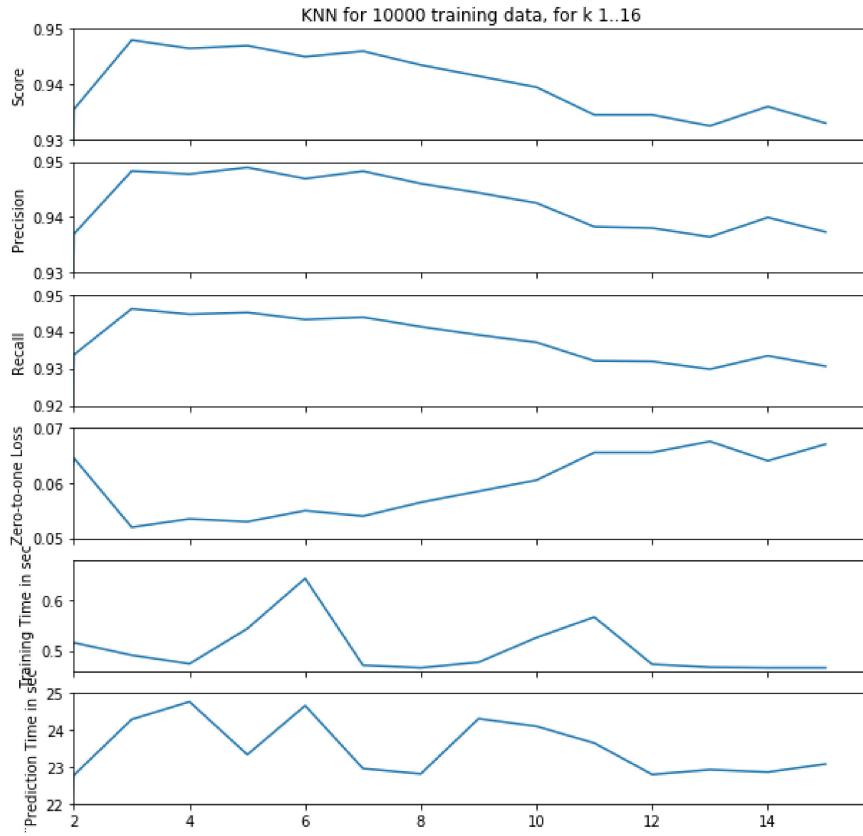
```
1 #Classifier
2 clf=neighbors.KNeighborsClassifier(k)
3 # temps de training et prédiction mesuré
4 startTrain = time.time()
5 clf.fit(xtrain,ytrain)
6 endTrain = time.time()
7 startpred = time.time()
8 clf.predict(xtest)
9 endpred = time.time()
10 # Metrics (score, precision, recall, zero-to-one loss, temps de training et d
11 predict = clf.predict(xtest)
12 score = clf.score(xtest,ytest)
13 precision = metrics.precision_score(ytest, predict, average='macro')
14 recall = metrics.recall_score(ytest, predict, average ='macro')
15 loss01 = metrics.zero_one_loss(ytest, predict)
16 timetrain = endTrain - startTrain
17 timePred = endpred - startpred
```

## 2.2 Choix des paramètres de l'algorithme

### 2.2.1 Mnist et KNN

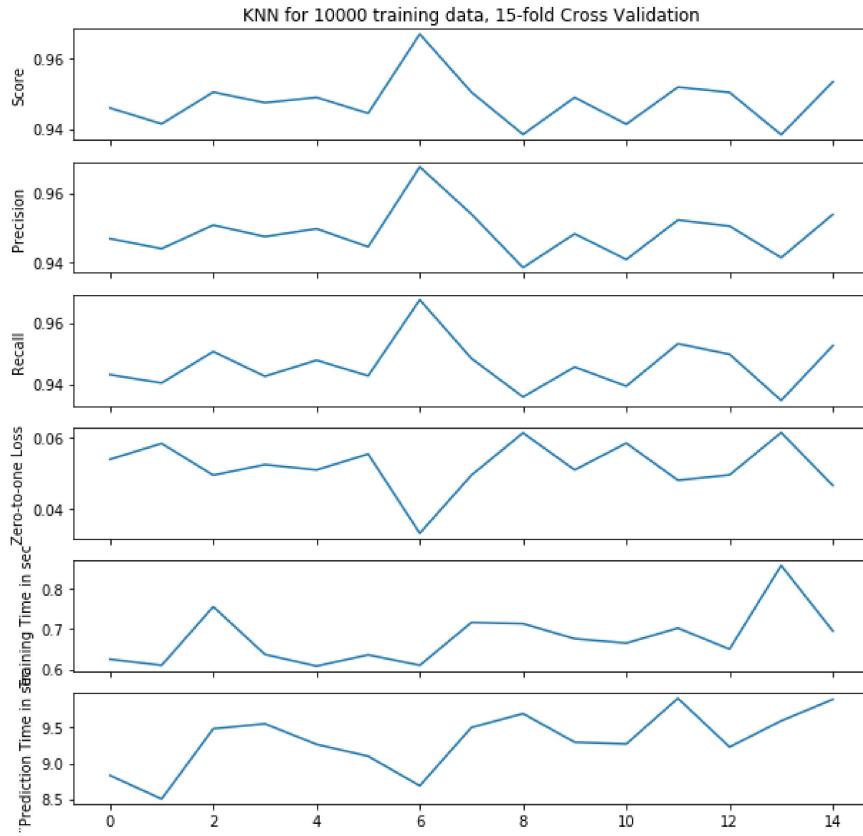
Nous avons réalisé plusieurs études sur KNN et les données de Mnist. Le training set était composé de 8000 données et le test set de 2000 données

Tout d'abord, on a cherché à **déterminer le meilleur K** pour nos données MNIST. On a donc entraîné et testé notre KNN pour des K différents (de 2 à 15).



On peut constater sur le graphe ci-dessus, que le meilleur score est atteint pour K=3 (94,8%) avec une bonne précision (94,83%) ainsi que le meilleur recall (94,62%). On notera que la précision pour K=5 est légèrement meilleur(94,90%).

**Les temps de training sont faibles**(entre 0,4 et 0,7sec). Ceux de **prédition sont plus élevés** mais globallement compris entre 22,5 et 25 sec. Les variations observées ici ne sont pas conséquentes et dépendent des autres processus s'exécutant sur l'ordinateur. Dans l'étude ci-dessous, le temps de prédition est plus faible. Je pense que c'est dû à mon ordinateur car j'ai mené cette étude un jour différent.



En faisant un **15-fold validation pour K=3**, on se rend compte que le modèle est plus sensible suivant les données d'entraînement. Mais cela reste correct car le score est compris entre 94% et 96%. En aucun cas le K-fold révèle un score en dessous de 94%.

Si l'on regarde l'effet de la taille du training et du test set sur le modèle pour k=3 on remarque que **plus le training set est grand, plus le score est bon**. Pour training set 10% le score est de 91%. Puis il augmente régulièrement jusqu'à 96,5% pour un training set de 90%. De même la précision et le recall augmentent.

D'autre part, si l'on fait varier le nombre de données utilisées pour entraîner et tester le modèle, on constate que plus le training set est grand, plus le score est meilleur. Pour un training set de 4000 données, le score est de 92,8%. Pour un training set de 20000 données, le score est de 96,4%.

Cependant, il faut aussi penser au temps de prédiction qui est plus important avec l'augmentation du nombre de données. C'est pour cela que l'algorithme KNN n'est pas adapté à de trop grosses quantités de données. En effet, KNN compare toutes les données de tests avec leur voisins (dans le training set), puis finalise le label. **Le temps de prédiction peut donc être très important si le training set est grand.**

KNN pour K=3 semble être un bon choix de paramètre. C'est un juste milieu entre le **underfitting et le overfitting**. Dans le premier cas, si K est trop petit, on risque de mal déterminer les classes. Dans le 2eme cas, on rique de "sur déterminer" les classes et de trop resserrer les frontières autour d'elles. Cela peut causer des mauvaises prédictions. Il faut donc trouver le juste milieu pour que le modèle puisse bien généraliser.

Mais qu'en est-il des distances présentées dans la partie 1 ?

Voici les scores obtenus pour les différentes distances :

- distance Euclidienne, le score = 96.45 %
- distance de Manhattan, le score = 93.65 %
- distance de Chebyshev, le score = 68.45 %
- distance de Hamming, le score = 74.0 %
- distance de Minkowski p=3 , le score = 95.3 %
- distance de Minkowski p=4, le score = 95.5 %
- distance de Minkowski p=5, le score = 95.85 %

On constate que dans le cas de mnist, la **distance euclidienne**, celle par défaut, nous donne le meilleur score. Il faut toutefois rester attentif à ce paramètre dans un autre domaine d'application.

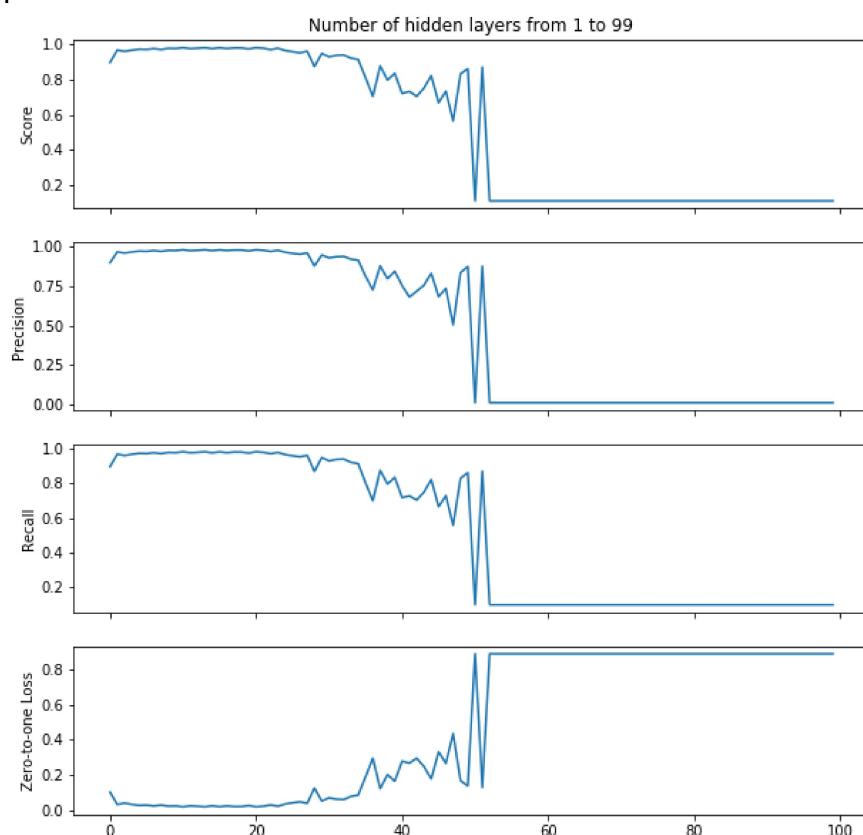
Le dernier paramètre étudié est celui des jobs. Il correspond au nombre de jobs exécutés en parallèle pour faire tourner l'algorithme KNN. -1 signifie que l'algo utilise tous les processeurs disponibles. L'utilisation de **toutes les ressources** disponible réduit le temps de prédiction :

- temps d'entraînement 1 job : 0.49367713928222656 sec.
- temps de prediction 1 job : 25.477837085723877 sec.
- temps d'entraînement -1 job : 0.5066468715667725 sec.
- temps de prediction -1 job : 7.67464017868042 sec.

## 2.2.2 Mnist et les neurones

Pour pouvoir évaluer le meilleur réseau de neurones, il faut tester différentes architectures.

Premièrement, on étudie **l'effet du nombre de couches cachées**. Pour cela, on fixe le nombre de neurones sur chaque couche à 50. Et on fait varier notre réseau de 1 à 100 couches cachées.

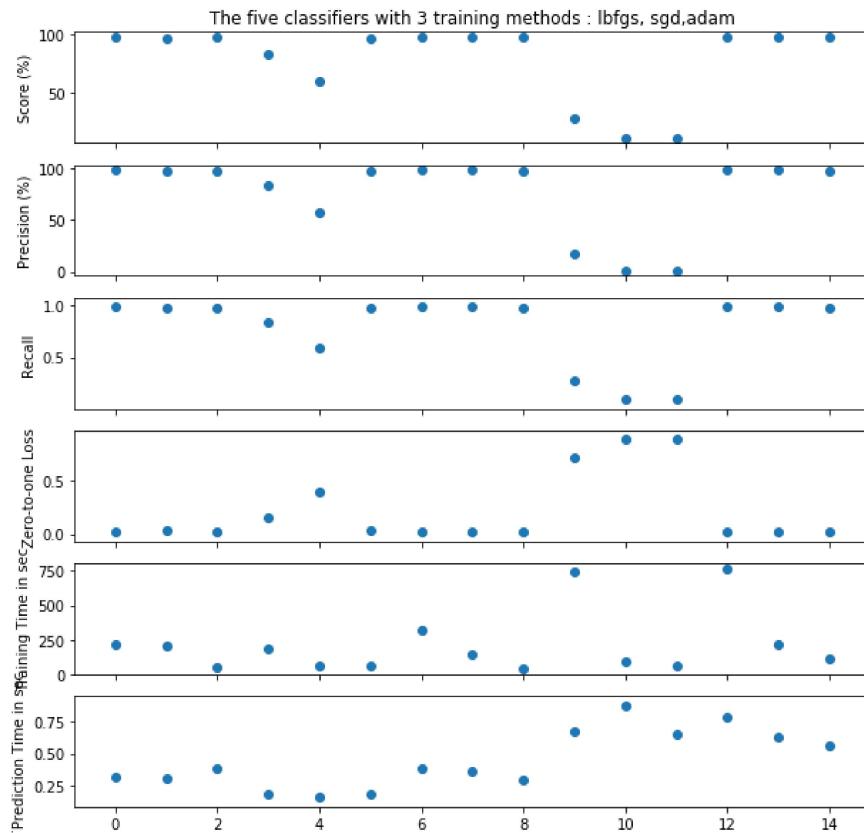


On remarque que passer 50 hidden layers, le réseau n'arrive même plus à classifier. IL a un score stagnant autour de 11%. Mais pour un réseau ayant entre 2 et 35 couches cachées, le score varie entre 96 et 98%

Ensuite on a essayé d'observer **l'effet du nombre de neurones sur chaque couche**. pour cela, on a créé 5 classifier comprenant entre 1 et 10 couches cachées et avec entre 10 et 300 neurones sur chaque couches :

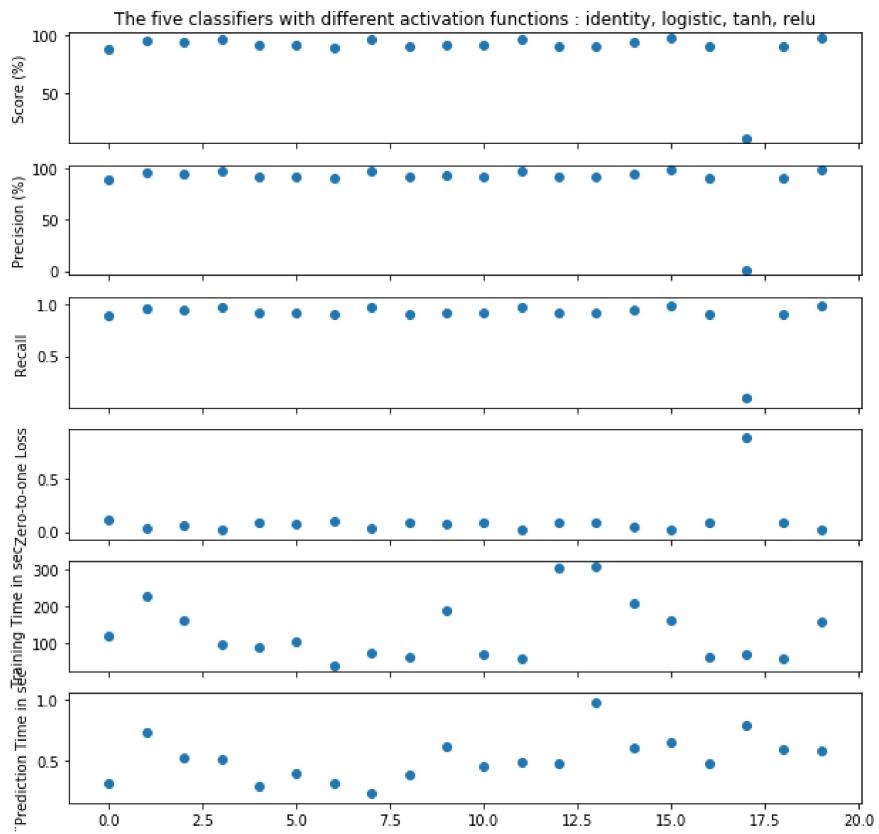
- Réseau 1 : 1 couche avec 300 neurones
- Réseau 2 : 3 couches avec respectivement 20, 200 et 50 neurones
- Réseau 3 : 5 couches avec 50, 100, 200, 100, 50 neurones (allure gaussienne)
- Réseau 4 : 7 couches avec 300, 250, 200, 150, 100, 50, 10 neurones (nombres décroissants)
- Réseau 5 : 9 couches avec 30, 60, 90, 120, 150, 180, 210, 240, 270 neurones (nombres croissants)

Ensuite, avec ces caractéristiques architecturales, on essaie différentes **méthodes de calcul des poids** (LBFGS, SGD et ADAM, comme vue dans la partie 1). Dans le graphe ci-après, les points de gauche à droites peuvent être regrouper par 3. Chaque groupe de 3 représente un réseau avec dans l'ordre des solvers LBFGS puis SGD puis Adam.



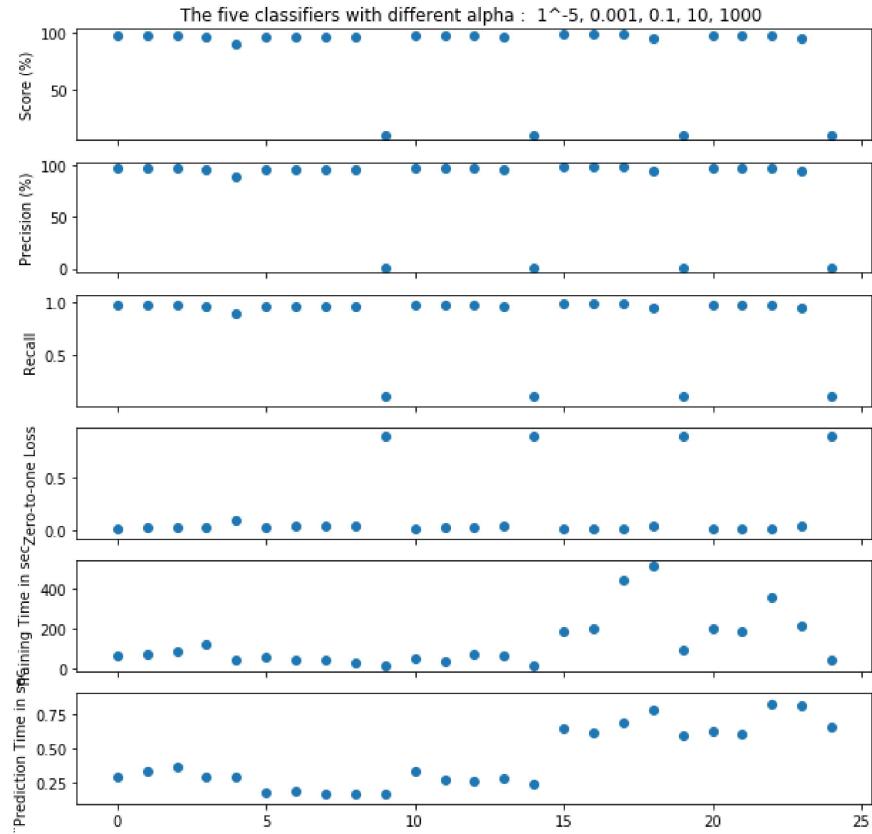
Pour les réseaux 1, 3 et 5, les 3 méthodes de calcul des poids donnent un résultats plus que correct (entre 96 et 98%). Le réseau 4 nous révèle ses faiblesses. On peut l'éliminer de notre étude. Le réseau 2 est un modèle acceptable uniquement si ADAM est employé. De façon générale, on remarque que la méthode LBFGS est plus lente à apprendre, avec un temps d'entraînement pouvant dépasser les 10 min pour 49000 données. C'est la méthode ADAM qui a le temps d'apprentissage le plus faible, pour les 5 réseaux testés.

On peut aussi comparer ces 5 réseaux en changeant leur **fonction d'activation**. On étudie ici les fonctions d'activation suivantes : identity, logistic, tanh et relu (cf partie 1 pour une présentation théorique). De gauche à droite, chaque groupe de 4 points représente un réseau.



Ici c'est le réseau 5 qui n'a pas réussi à classifier, avec la fonction d'activation logistique. On remarque globalement que c'est la fonction d'activation Relu, celle par défaut qui donne le meilleur score, la meilleure précision et le meilleur recall. Pour les temps d'apprentissage, c'est plus mitigé et cela dépend des réseaux. Avec la fonction d'activation logistique, le temps d'apprentissage est souvent plus élevé. Et avec Relu il est souvent plus faible.

On peut aussi faire varier le alpha qui sert à éviter l'overfitting. Un alpha faible encouragera des poids et un biais élevés, résultant en des frontières de classification plus lisses. Tandis qu'un alpha élevé encouragera des poids plus petits, résultat en des frontières potentiellement plus compliquées.

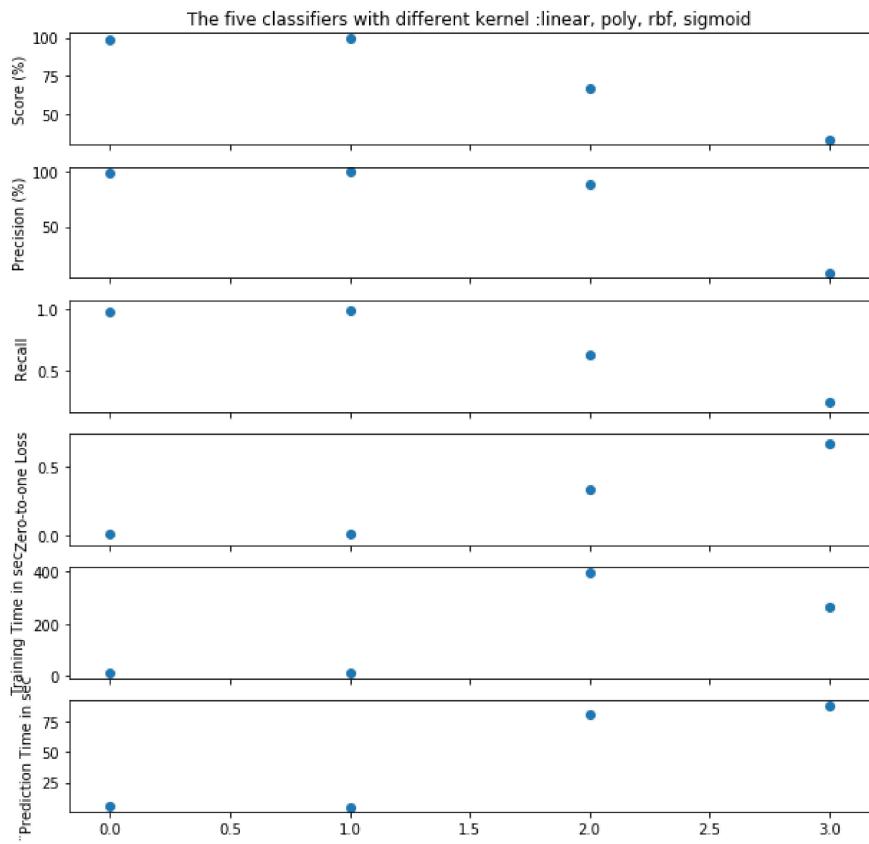


Sur nos 5 réseaux, on constate que un alpha trop élevé (1000) ne permet plus de classifier nos données. Le score chute à 11%. En regardant les temps d'apprentissage, un faible alpha (0,001 ou 0,1) est préférable.

Avec l'étude sur ces 5 réseaux, on remarque que le réseau 4 qui a un nombre croissant de neurones sur chaque couche (forme de pyramide) n'est pas idéal. Au contraire avoir un pyramide inversée, comme avec le réseau 5, ou une forme losange comme le **réseau 3**, offre un bon score de classification. Pour les méthodes de poids, il faut privilégier **ADAM** qui offre le meilleur score pour le temps d'apprentissage le plus faible. De même, la fonction d'activation par défaut, **relu**, semble être la fonction la plus adéquate. Enfin, il faut privilégier un **alpha entre 0,001 et 0,1** pour ne pas avoir de hunderfitting et ne plus pouvoir classifier nos données.

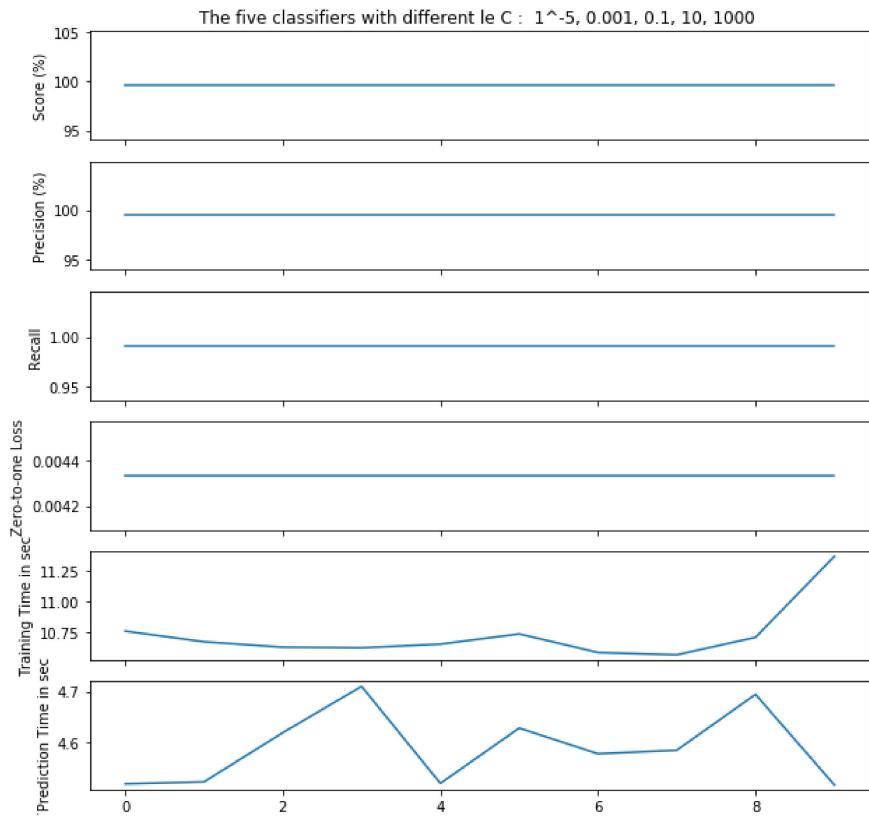
### 2.3.3 Mnist et SVM

Pour étudier le modèle SVC de scikitlearn, sur mnist, on commence par étudier les différents **kernels**.



On constate ici que les kernel linéaire et poly ont un score satisfaisant (98,6% et 99,5%), comparé aux kernel rbf et sigmoid. De même, leur temps d'apprentisage et de prédiction sont très faibles (environ 11 et 5 sec).

Maintenant, on fait varier le paramètre **C**, avec un kernel poly :



Pour C, rien ne change, le score reste à 99,56%, seul le temps d'entraînement et de prédiction varient. C'est à cause des autres processus s'exécutant sur mon ordinateur.

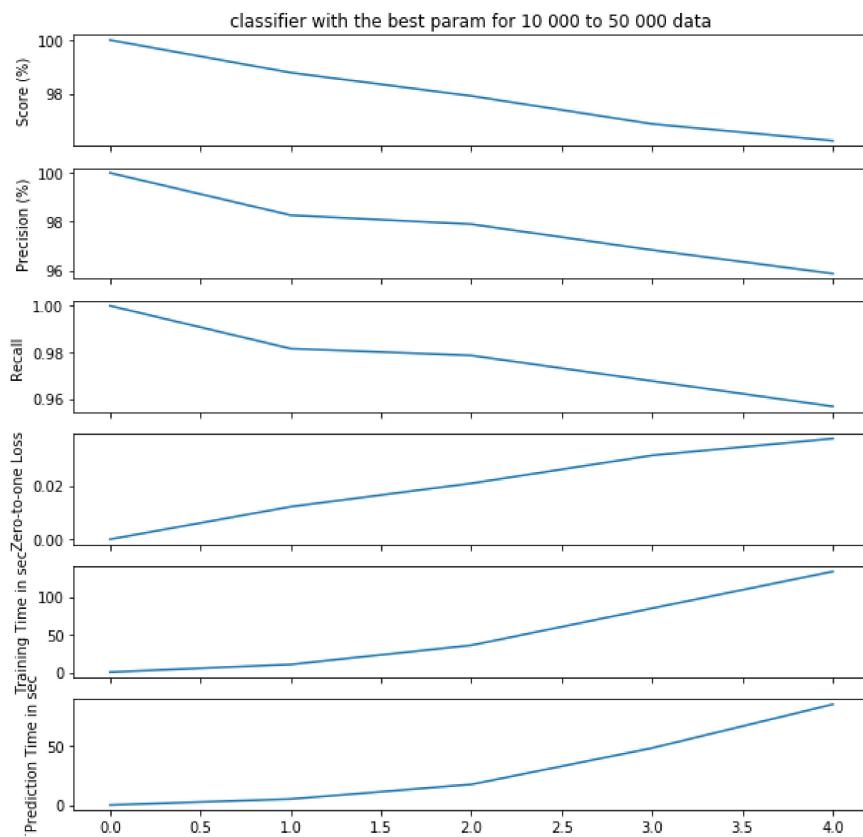
En faisant varier **gamma**, on a les même constats. Le graphe n'est donc pas rapporté ici, pour ne pas surcharger le rapport.

Sikit-learn nous offre un outils : GridSearchCV, qui va trouver la meilleure combinaison suivant les paramètres qu'on lui donne. (Le seul inconvénient de cet outil est qu'il est long à exécuter). Voici donc comment l'exécuter :

Entrée [ ]:

```
1 # PARTIE AVEC GridSearchCV
2
3 paramGrid = {'C': [0.001, 0.01, 0.1, 1, 10, 100],
4             'gamma': [0.0001, 0.001, 0.01, 0.1],
5             'kernel': ('linear', 'poly')}
6
7
8 grid = GridSearchCV(SVC(), paramGrid, cv=5)
9 grid = grid.fit(X=xtrain, y=ytrain)
10 print (grid.best_params_)
11
12 #The best param are : {'C': 0.001, 'gamma': 0.0001, 'kernel': 'Linear'} for 10
```

Avec les meilleurs paramètres choisi pour 10000 données, on le fait tourner en augmentant les données, jusqu'à 50000. On se rend compte que le modèle ne généralise pas bien. Il faudrait refaire une études pour changer les paramètres pour des données plus importantes. Cependant, cela demande du temps.



### 3. Analyse comparative

Pour pouvoir comparer les 3 méthodes rappelons les performances obtenues :

- pour KNN : pour 20000 données, 96,5% pour un temps de training négligeable et un temps de prédiction de 30sec
- pour ANN : pour 70000 données 98,5%, un temps d'entraînement de 5min et un temps de prédiction négligeable
- pour SVM : pour 20000 données, un score de 98,78% pour un temps de training de 10 sec et un temps de prédiction de 5 sec.

A première vu, SVM est donc le meilleur modèle en fonction du score et des temps d'entraînement et de training. Mais on a aussi vu que SVM ne généralisait pas bien car si l'on garde le même modèle, on tombe à 96% pour des temps d'entraînement et de prédiction de 4min.

On remarque aussi qu'entre KNN et MLP, le temps d'apprentissage et de prediction ont été inversé. En effet, KNN apprend très rapidement, mais il doit tester tous les voisins avant de pouvoir labéliser les données de test donc son temps de prédiction est long. Inversement, le Réseau de neuronne doit stabiliser ses poids à l'entraînement puis la prédiction est instantannée.

En ce sens, ANN est à préconnisé si on l'adapte pour des données reçues en streaming, ou si l'on doit prédire un grand nombre de données. Au contraire, on ne recommandera pas KNN qui est trop coûteux en temps de prédiction.

Globalement, les 3 modèles permettent bien de classifier nos données mnist.

Personnellement, je suis plus à l'aise avec les modèles de KNN et ANN. J'ai du mal à voir les effets de C et de gamma dans le modèle de SVM.

Entrée [ ]:



```
1 #Meilleur KNN
2 # K=3
3 # Toutes les ressources (njobs=-1)
4 # distance euclidienne par défaut
5
6 # imports mnist et KNN
7 from sklearn.datasets import fetch_mldata
8 from sklearn import datasets
9 from sklearn.model_selection import train_test_split
10 from sklearn import neighbors
11 import numpy as np
12 import matplotlib.pyplot as plt
13 from sklearn import metrics
14 import time
15
16 mnist=datasets.fetch_mldata('MNIST original')
17
18
19 #ECHANTILLONS DE 10000
20 indices = np.random.randint(70000, size=20000)
21 data = mnist.data[indices]
22 target = mnist.target[indices]
23 #pour séparer le dataset en 2 échantillons de train et de test
24 xtrain, xtest, ytrain, ytest =train_test_split(data, target, train_size=0.8)
25
26
27 #Classifier
28 clf=neighbors.KNeighborsClassifier(3,n_jobs=-1)
29 # temps de training et prédiction mesuré
30 startTrain =time.time()
31 clf.fit(xtrain,ytrain)
32 endTrain = time.time()
33 startpred= time.time()
34 clf.predict(xtest)
35 endpred = time.time()
36 # Metrics (score, precision, recall, zero-to-one loss, temps de training et de prediction)
37 predict = clf.predict(xtest)
38 score = clf.score(xtest,ytest)
39 precision = metrics.precision_score(ytest, predict, average='macro')
40 recall = metrics.recall_score(ytest, predict, average ='macro')
41 loss01 = metrics.zero_one_loss(ytest, predict)
42 timetrain = endTrain - startTrain
43 timePred = endpred - startpred
44
45 print("Ce modèle de KNN, K=3, à un score de ", score*100, "%.")
46 print("4eme image : prédiction ",predict[3], "reel : ", ytest[3])
47 print ("ce KNN à une précision de", precision*100, "%.")
48 print ("ce KNN à un recall de",recall*100, "%.")
49 print ("ce KNN à un zero-one_loss de",recall*100, "%.")
50 print("    temps apprentissage : ", timetrain, "sec , temps prediction = ", ti
51
52 #Ce modèle de KNN, K=3, à un score de 95.75 %.
53 #4eme image : prédiction 2.0 reel : 2.0
```

```
54 #ce KNN à une précision de 95.8171550275235 %.  
55 #ce KNN à un recall de 95.69581521050152 %.  
56 #ce KNN à un zero-one_loss de 95.69581521050152 %.  
57 #     temps apprentissage : 1.5378844738006592 sec , temps prediction = 29.13
```

Entrée [ ]:



```
1 # MEILLEUR MLP
2 # réseau 3 (50, 100, 200, 100,50)
3 # Relu et Adam (par défaut)
4 # petit alpha 0,1
5
6 #import nécessaires
7 import numpy as np
8 from sklearn import datasets
9 from sklearn.model_selection import train_test_split
10 from sklearn.neural_network import MLPClassifier
11 from sklearn import metrics
12 import time
13
14 mnist=datasets.fetch_mldata('MNIST original')
15 #randomise Data et target
16 indices = np.random.randint(70000, size=70000)
17 data = mnist.data[indices]
18 target = mnist.target[indices]
19 # on veut un training set de 49000
20 xtrain, xtest, ytrain, ytest =train_test_split(data, target, train_size=49000)
21
22 clf3 = MLPClassifier(hidden_layer_sizes=(50, 100, 200, 100,50),alpha= 0.1) # a
23 startTrain =time.time()
24 clf3.fit(xtrain, ytrain)
25 endTrain = time.time()
26 # Predict
27 startpred= time.time()
28 predict = clf3.predict(xtest)
29 endpred = time.time()
30 score = clf3.score(xtest,ytest)
31
32 recall = metrics.recall_score(ytest, predict, average ='macro')
33 precision = metrics.precision_score(ytest, predict, average='macro')
34 loss01 = metrics.zero_one_loss(ytest, predict)
35 timetrain = endTrain - startTrain
36 timePred = endpred - startpred
37
38
39 print("Ce modèle de MLP, de 5 couches de 50, 100, 200, 100, 50, a un score de
40 print("4eme image : prédiction ",predict[3], "reel : ", ytest[3])
41 print ("ce modèle de MLP à une précision de", precision*100, "%.")
42 print ("ce modèle de MLP à un recall de",recall*100, "%.")
43 print ("ce modèle de MLP à un zero-one_loss de",recall*100, "%.")
44 print("    temps apprentissage : ", timetrain, "sec , temps prediction = ", ti
45
46 #Ce modèle de MLP, d' 5 couches de 50, 100, 200, 100, 50, a un score de 98.50
47 #4eme image : prédiction 5.0 reel : 5.0
48 #ce modèle de MLP à une précision de 98.50503688809177 %.
49 #ce modèle de MLP à un recall de 98.49104431028593 %.
50 #ce modèle de MLP à un zero-one_loss de 98.49104431028593 %.
51 #    temps apprentissage : 322.9891335964203 sec , temps prediction = 0.6702
```

## Conclusion

Ce rapport propose des paramétrage de 3 méthodes d'apprentissage supervisé pour classifier les données mnist. Les 3 méthodes portent leur fruits et présentes toutes des avantages et inconvénient.

Il serait utile de poursuivre l'étude en l'étendant à d'autres méthodes supervisées comme le décision tree, les réseaux baésian, les CNN et RNN.

Entrée [ ]:



1	
---	--