GONZALES Adrien 5SDBD A2

MEGA Adrian

TP Apprentissage Supervisé

# Introduction

Ce rapport a pour but de rendre compte de nos travaux en TP d’apprentissage supervisé. Chaque chapitre sera consacré à l’un des classifieurs étudiés à chaque TP (KNN, MLP, SVC) puis une synthèse comparera leur efficacité à chacun. Les scripts des programmes peuvent être retrouvés sur le dépôt suivant :

<https://github.com/dx07/5SDBD-Apprentissage>

Le jeu de données sur lequel nous travaillons est une base de données de chiffres écrits à la main. Nous avons également la vraie valeur représentée par ces chiffres incluse dans le dataset pour identifier quel chiffre correspond au dessin.

Selon les questions des sujets de TP, les tailles des données sélectionnées seront amenées à être changées pour des raisons de démonstration.

# I – Méthode KNN, les k plus proches voisins

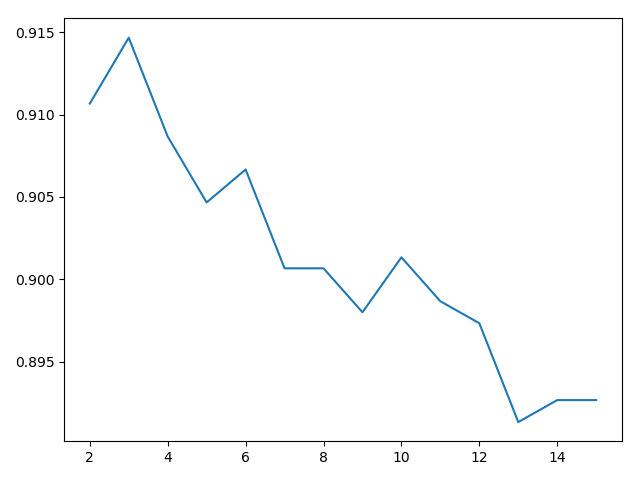
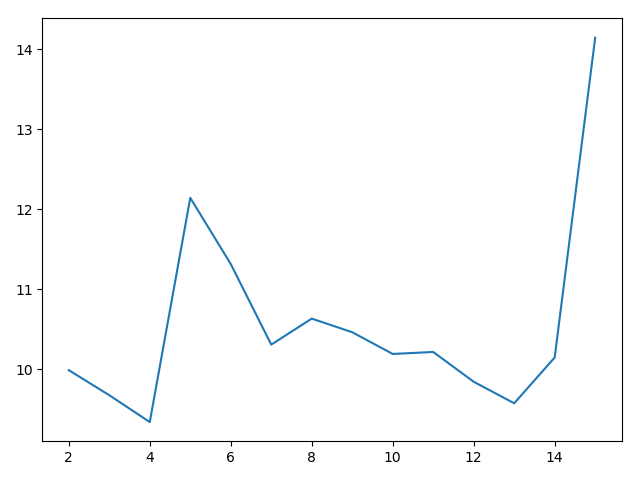
Cette méthode consiste à rechercher les k plus proches exemples labellisés pour identifier le label de l’instance analysée. Nous pouvons faire varier les paramètres suivants :

* K : Le nombre de voisins plus proches
* n\_jobs : Le nombre de jobs parallèles (1 par défaut, -1 pour tous)

Pour ces premiers tests, nous avons pris un extrait du dataset de taille 5000. L’algorithme étant très long à s’exécuter sur nos machines, nous sommes contraints d’utiliser une petite partie du dataset entier. Le partage du dataset entre données d’entraînement et données de test est respectivement de 70% et 30%.

## a) Variation de K

En faisant varier le nombre de plus proches voisins retenus, nous allons analyser les différents scores de fiabilité obtenus ainsi que les temps d’apprentissage nécessaires. Nous avons fait ces tests pour un K valant de 2 à 15. Voici les résultats obtenus.



Concernant le temps (figure de gauche), on ne constate pas de différence significative selon la valeur de K. En revanche, pour le score (figure de droite), nous pouvons observer un pic d’efficacité à K = 3. Il serait le paramètre le plus optimal pour ce jeu de données.

## b) Variation de n\_jobs

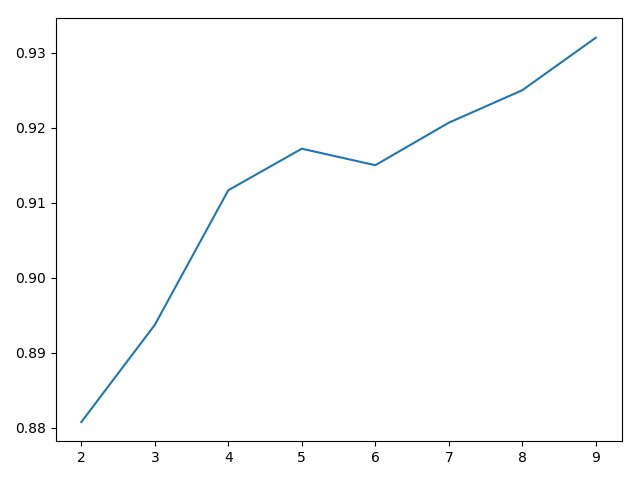
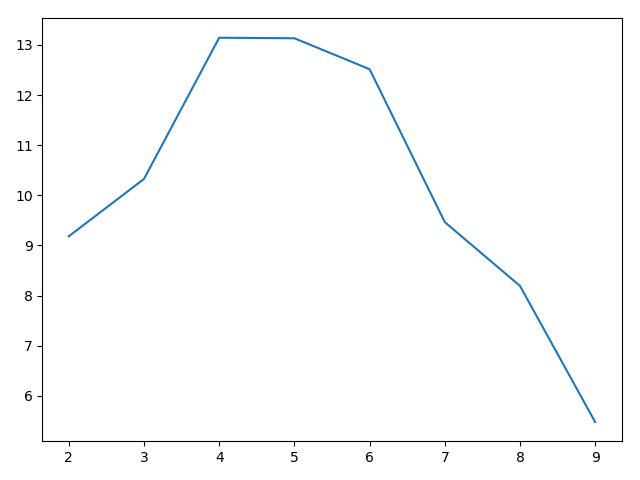
Cette étape va nous montrer si on gagne du temps en changeant le nombre de jobs parallèles utilisés. Pour cela nous allons montrer la différence entre n\_jobs = 1 et n\_jobs = -1, soit le maximum de jobs en parallèle. Nous gardons K = 3 pour cette étape ainsi que l’échantillon de données sélectionné précédemment. Nous obtenons les temps suivants :

* n\_jobs : -1 Temps : 4.365392208099365
* n\_jobs : 1 Temps : 9.597620725631714

Cela met clairement en évidence qu’avec davantage de ressources, la puissance de calcul est grandement augmentée et le temps de calcul en est inéluctablement réduit.

## c) Variation de la taille de train / test

Exceptionnellement pour cette partie, nous allons tester la variation de la taille d’entraînement pour apprécier l’impact qu’elle peut avoir sur les temps de calcul et le score d’efficacité. Nous conservons toujours K = 3.

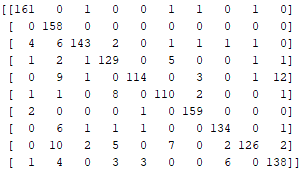


A gauche, nous pouvons voir que le temps de calcul se voit un peu augmenté au début, mais significativement diminué avec une plus grande taille de données d’entraînement (deux fois plus rapide avec 90% qu’avec 40%). Pour le score, celui-ci augmente forcement avec davantage de données pour s’entrainer, ce qui est logique dans le fond.

## d) Matrice de confusion

Voici à présent la matrice de confusion affichant sur les 10 chiffres possibles de lire le nombre d’occurrences ayant été correctement lues. On obtient les résultats suivants.

* score: 0.9146666666666666
* time: 10.216910362243652



Par exemple pour la ligne 6, qui correspond au chiffre 5, il y a eu 110 occurrences correctement identifiées contre 13 erronées. Cela met en évidence la bonne efficacité de KNN.

# II – Méthode MLP, les couches de neurones

TODO

# III – Méthode SVC, la machine à vecteurs de support

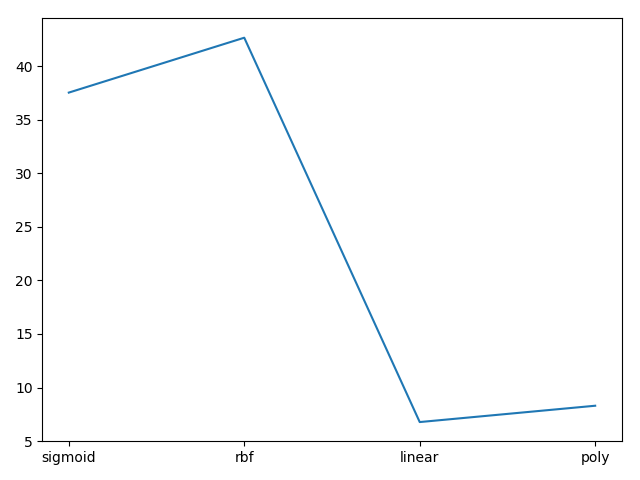
Ces derniers tests se feront sous la méthode d’apprentissage SVC, forme de SVM. Nous pourrons faire varier les paramètres suivants :

* Kernel : Le type de noyau utilisé
* C : La tolérance aux erreurs
* Gamma : Le coefficient utilisé dans les noyaux

Nous gardons également le jeu de données de taille 5000 pour accélérer les tests de calcul, avec une part de données d’entraînement toujours égale à 70%.

## a) Temps d’exécution des kernels

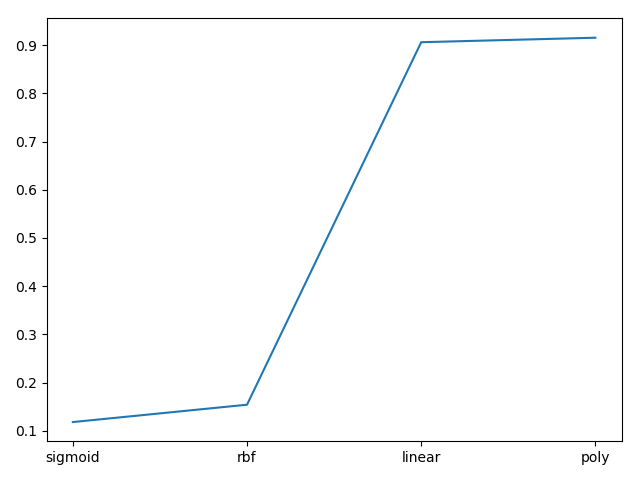
En testant chacun des kernels disponibles dans sklearn, nous avons comparé leur temps d’exécution respectifs afin de constater leur efficacité.



On constate rapidement que le kernel le plus efficace en termes de rapidité est sans conteste linear. Il devance les premiers d’un incroyable écart en s’exécutant en près de 7 secondes. Poly a cependant de très bon résultats également, légèrement supérieurs à linear. L’extrait de données étant resté le même entre les exécutions des 4 cas, nous en concluons tout de même que linear est le plus rapide. Mais cela ne veut pas dire qu’il obtient les meilleurs résultats en termes de fiabilité.

## b) Score des kernels

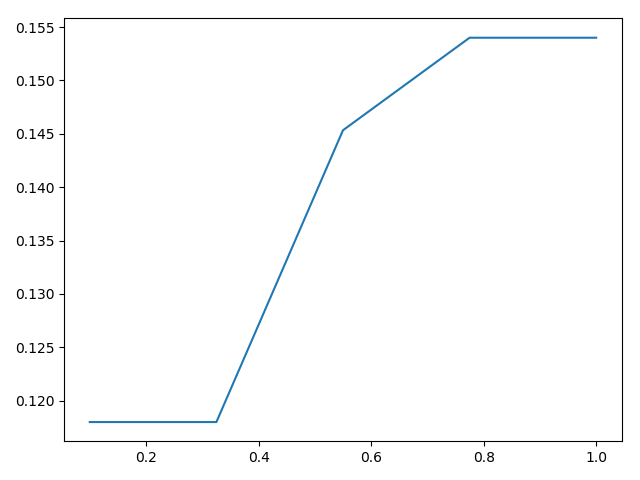
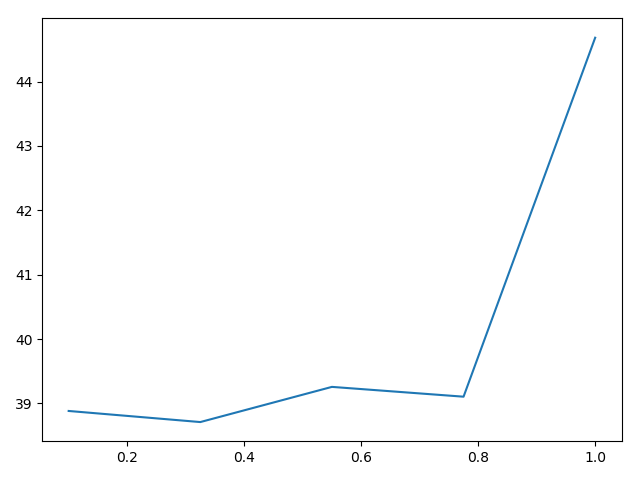
Ici nous allons comparer les scores de fiabilité obtenus par chacun de nos kernels. Sans changer l’échantillon de données test, voici les résultats obtenus.



Une fois de plus linear se démarque avec poly pour leur haut taux d’efficacité ! Nous ne dépassons même pas les 20% avec sigmoid et rbf. Cela laisse penser que nous avons tout intérêt à préférer les autres car ils sont meilleurs en tout point.

## c) Variation de la tolérance aux erreurs

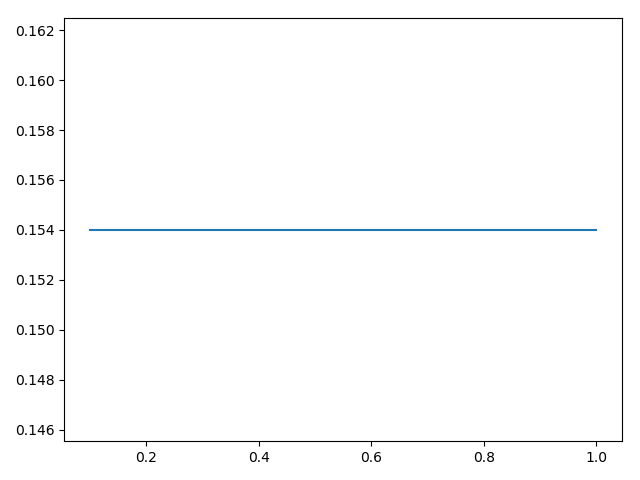
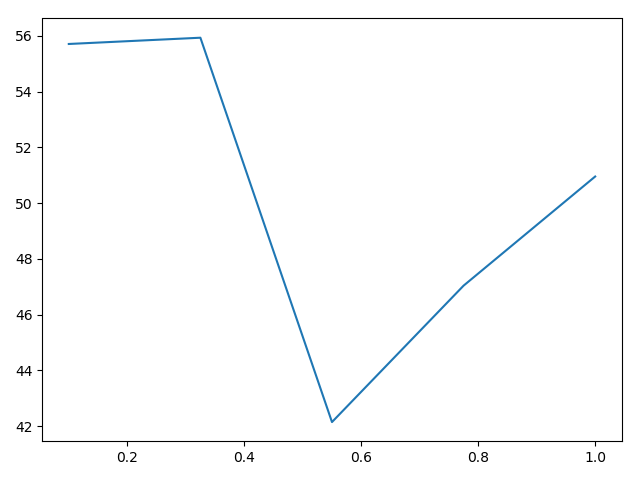
Pour ces tests, nous avons pris le kernel rbf pour tester l’impact de la variation de C sur l’exécution du solver. Nous avons pris 5 valeurs C entre 0.1 et 1.



Cela nous permet d’observer le phénomène de plus près concernant la fiabilité. Celle-ci est déjà à son maximum pour 0.8. Le temps de calcul quant à lui varie très peu selon C. Il est évident qu’avec une autre acceptation des erreurs on en arrive à une fiabilité qui n’augmente plus (cela étant rbf n’était pas le kernel le plus fiable …).

## d) Variation du coefficient Gamma

Ce coefficient sert pour le calcul via les kernels rbf, poly et sigmoide. Nous allons le tester avec rbf de nouveau en le faisant varier entre 0.1 et 1 à nouveu.



Cette fois, malgré un temps très variant (même assez haut dans les premiers cas) cela n’a aucune incidence sur l’efficacité du solver.

## e) GridSearchCV

Ce modèle est supposé trouver de lui-même les paramètres optimaux pour le solver. Nous lui avons donné le choix entre tous les kernels et un C entre 1 et 10. Voici les résultats :

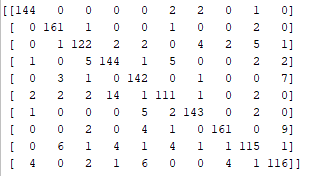
* score: 0.9153333333333333
* time: 441.23244857788086

En soi, ça a duré beaucoup de temps pour une durée équivalente à nos précédents tests avec linear et poly, donc pas plus fructueux que ça pour notre dataset.

## f) Matrice de confusion

Enfin, nous allons tester la matrice de confusion sur le kernel linear, sans paramètres additionnels et pour le même extrait de dataset qu’avant.

* score: 0.906
* time: 7.316744327545166



Cette matrice nous indique par exemple que pour la seconde ligne (correspondant au chiffre 1), 161 valeurs ont été correctement identifiées, tandis que 4 ont été erronées. Globalement, on peut constater l’efficacité de notre SVC sur les données de test (30% de 5000 valeurs).

# Synthèse

Nous avons conclu plusieurs choses concernant le travail des 3 méthodes sur notre jeu de données.

TODO