Siméon GAUMART

Nguyet Ha TRAN

5SDBD - B1

**Rapport TP Apprentissage Supervisé**

# **Introduction**

De nos jours, l’apprentissage est un domaine important des sciences informatiques qui permet aux ordinateurs d'apprendre sans être programmés. Les deux principales catégories de l'apprentissage utilisées sont apprentissage supervisé et apprentissage non supervisé.

Dans le cadre du TP Apprentissage Supervisé, nous travaillons avec le jeu de données MNIST contenant des chiffres de 0 à 9 manuscrits. Il s’agit d’un problème de classification multiclasse de 10 classes. l’objectif du TP est d’appliquer trois différentes méthodes d’apprentissage supervisé qui sont la méthode k plus proches voisins (kNN), les réseaux de neurones artificiels (ANN) et les machines à vecteurs supports (SVM) sur le jeu de données MNIST. D’une part, nous explorons différents hyperparamètres afin de connaître leurs influences sur le modèle construit avec le jeu de données, puis, nous mettons en place une modèle performant en termes de précision des prédictions et de temps d’exécution. D’autre part, nous comparons ces trois méthodes sur de multiples aspects afin de mieux comprendre l’efficacité de chaque méthode avec le jeu de données MNIST et l’extensibilité potentielle vers d’autres jeux de données de tailles plus grandes et de dimension plus grande

# **Méthodes implémentées**

## **KNN - K plus proches voisins**

Dans cette partie nous allons étudier la méthode d’apprentissage supervisé des K plus proches voisins (KNN).

Cette méthode consiste à classifier les données selon la classe la plus représentée dans leurs K plus proches voisins. C’est-à-dire que, au moment de l'entraînement du modèle, on ne fait que placer les données (d’entraînement) sur un repère de dimension égale au nombre de features de chaque données (par exemple un repère de dimension 784 pour les données de MNIST). Ainsi, au moment de la prédiction, le modèle calculera la distance entre la donnée à prédire et les données d’entraînement et attribuera le label le plus présent dans les k plus proches données d'entraînement à la donnée à prédire.

Pour utiliser la méthode des K plus proches voisins, nous avons utilisé la méthode neighbors.KNeighborsClassifier de ScikitLearn.

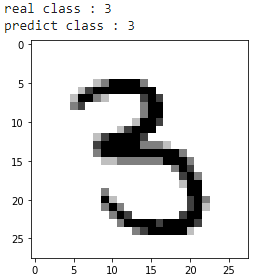
Tout d’abord nous avons pris aléatoirement 5000 données dans les 70000 du jeu de données MNIST, puis nous avons coupé ces 5000 données en deux afin d’avoir des données pour l’entraînement (apprentissage) du modèle (80% x 5000 = 4000) et des données pour l’évaluation (test) de notre modèle (20% x 5000 = 1000).

On entraîne ensuite un premier classifieur des k plus proches voisins avec un k de 10 en utilisant la distance euclidienne (p=2), et on effectue la prédiction sur les données de test tout en calculant le temps d’apprentissage et de prédiction. On obtient :



On voit ici que le temps d’apprentissage est bien inférieur au temps de prédiction. On peut expliquer cela par le fait que, comme on l’a dit précédemment, l’entraînement consiste uniquement à placer les données d’apprentissage dans un repère (fictivement, de les prendre en considération) là où la phase de prédiction nécessite de calculer les distances entre les données, ce qui demande beaucoup plus d’opérations.

On affiche ensuite l’image, la classe réelle et la classe prédite de la 4ème donnée des données d’évaluation :

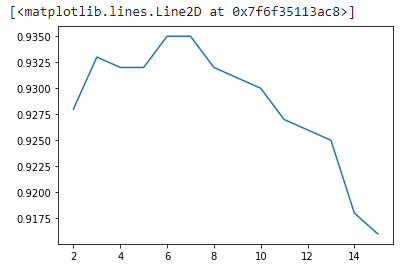


On remarque que la prédiction est correcte.

On trouve que le score pour ce classifieur est de 0.93. Ce qui veut dire que ce classifieur a 93% de chance de prédire le label d’une image (du jeu de données MNIST) correctement.

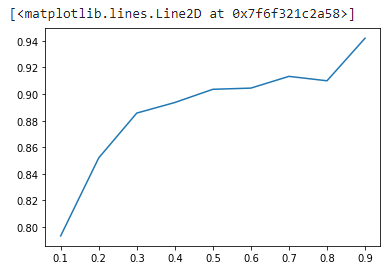
On va maintenant faire varier les paramètres du classifieur ainsi que les tailles des jeux de données utilisés pour l’entraînement et la prédiction avec de voir quels paramètres sont les plus adaptés pour le jeu de données MNIST.

On commence par faire varier K (n\_neighbors) entre 2 et 16 et on calcule le score pour chaque valeur de K (en utilisant les mêmes autres paramètres que précédemment):



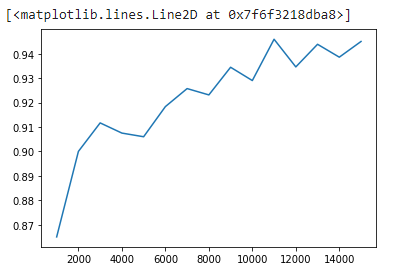
On remarque qu’on obtient le meilleur score pour k=6 ou k=7. On remarque que le score baisse ensuite drastiquement en augmentant k. On peut donc déduire (par bon sens, car les images du jeu de données se ressemblent un minimum) que pour des valeurs plus élevées de k, on n’atteindra plus un score aussi bon que pour k=6 et k=7.

On fait varier ensuite le pourcentage du jeu de données d’entraînement entre 10% et 90% (en gardant le même nombre de données totales de 5000, et en faisant donc varier le jeu de données d’évaluation par conséquent), en reprenant k=10 et les autre paramètres précédents :



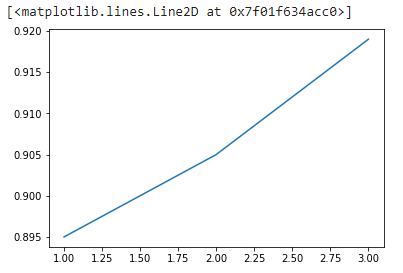
On remarque que plus le pourcentage est élevé, plus le score est bon, ce qui parait logique vu qu’on a plus de données pour l’entraînement et que notre modèle devient donc plus précis (tant qu’il n’y a pas d’overfitting). Cependant, augmenter la taille du jeu de données d’entraînement implique de diminuer la taille du jeu de données d’évaluation et donc de diminuer la précision du score.

On fait maintenant varier la taille du jeu de données utilisé (training 80% + prediction 20%) tout en gardant les paramètres précédents :



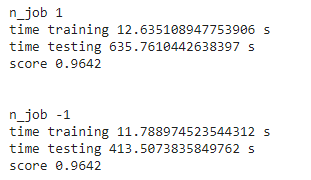
On remarque que globalement le score augmente pour des tailles du jeu de données plus grandes, ce qui paraît cohérent vu que la taille du jeu de données d’entraînement augmente avec et permet au modèle d’être plus précis. On remarque cependant que le score a l’air de converger petit à petit et pourrait même décroître avec trop de données à cause de l’overfitting.

On fait varier le type de distance p entre 1 et 3 (respectivement distance de Manhattan, distance euclidienne et distance de Minkowski) :



On remarque que la distance de Minkowski est plus adaptée pour le jeu de données MNIST.

Pour finir on calcule le temps d'exécution pour n\_job=1 (pas de parallélisation pour la recherche des voisins) et n\_job=-1 (parallélisation maximale possible) avec un jeu de données de 50000 images, avec un ratio 80% 20% pour le training et la prediction :



On remarque qu’avec la parallélisation le classifieur est plus rapide (mais pas 2 fois plus sûrement à cause de la fusion nécessaire après parallélisation) pour un résultat identique en termes de score.

On remarque donc que la méthode des K plus proches est une méthode pouvant être très efficace pour les jeux de données ayant des clusters assez distincts, se mélangeant peu, et que le temps de training du classifieur est très faible. Cependant la méthode est moins efficace sur les clusters plus proches les uns des autres (notamment sur les bords), ou les clusters se mélangeant. De plus le temps de prédiction est très élevé et “exponentiel” par rapport à la taille du jeu de données (en effet la complexité temporelle de l’algorithme permettant de trouver le plus proche voisin est en Ω(nlogn) (Θ(nlogn)

avec une méthode divide and conquer)).

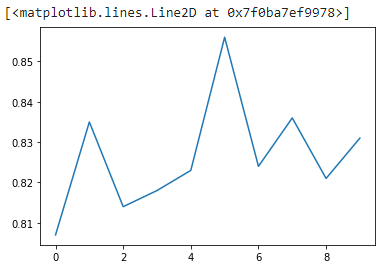
## **ANN - Réseaux de neurones artificiels**

Dans cette partie nous allons étudier la méthode des Réseaux de Neurones artificiels (ANN) en utilisant la méthode des perceptrons multicouches (MLP).

Cette méthode permet d’ajouter des couches de neurones cachées, entre les entrées et la couche de sortie. L’ajout de ces couches de neurones permet d’optimiser le modèle à apprendre et à effectuer des tâches.

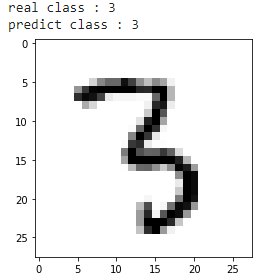
Pour utiliser MLP, nous allons utiliser la méthode neural\_network.MLPClassifier de scikitLearn. Et comme dans la partie précédente nous allons utiliser un jeu de données de 5000 images de MNIST avec 80% pour le training et 20% pour la prédiction.

On commence par entraîner un modèle ayant une seule couche cachée de 50 neurones et on calcule le score. On fait une boucle pour relancer le même programme plusieurs fois :



On remarque que le score varie malgrès qu’on lance le même programme avec les mêmes variables et le même jeu de données. Le score pour une couche cachée de 50 neurones est donc d’environ 0.83 en moyenne.

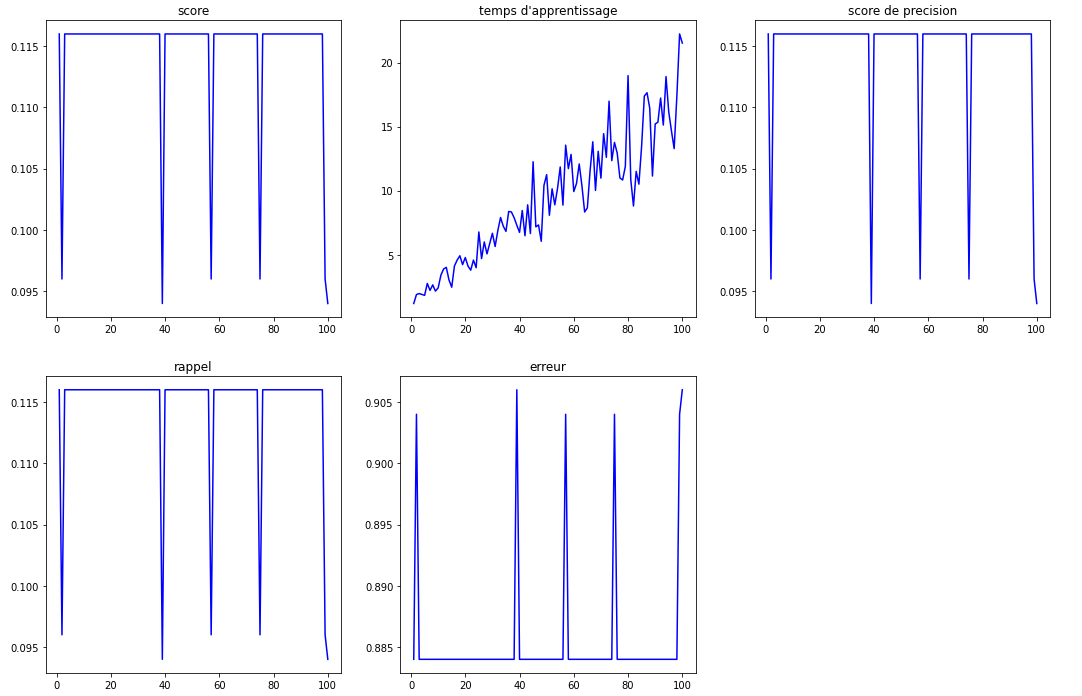
On affiche ensuite l’image, la classe réelle et la classe prédite de la 4ème donnée des données d’évaluation :



On remarque que la prédiction est correcte.

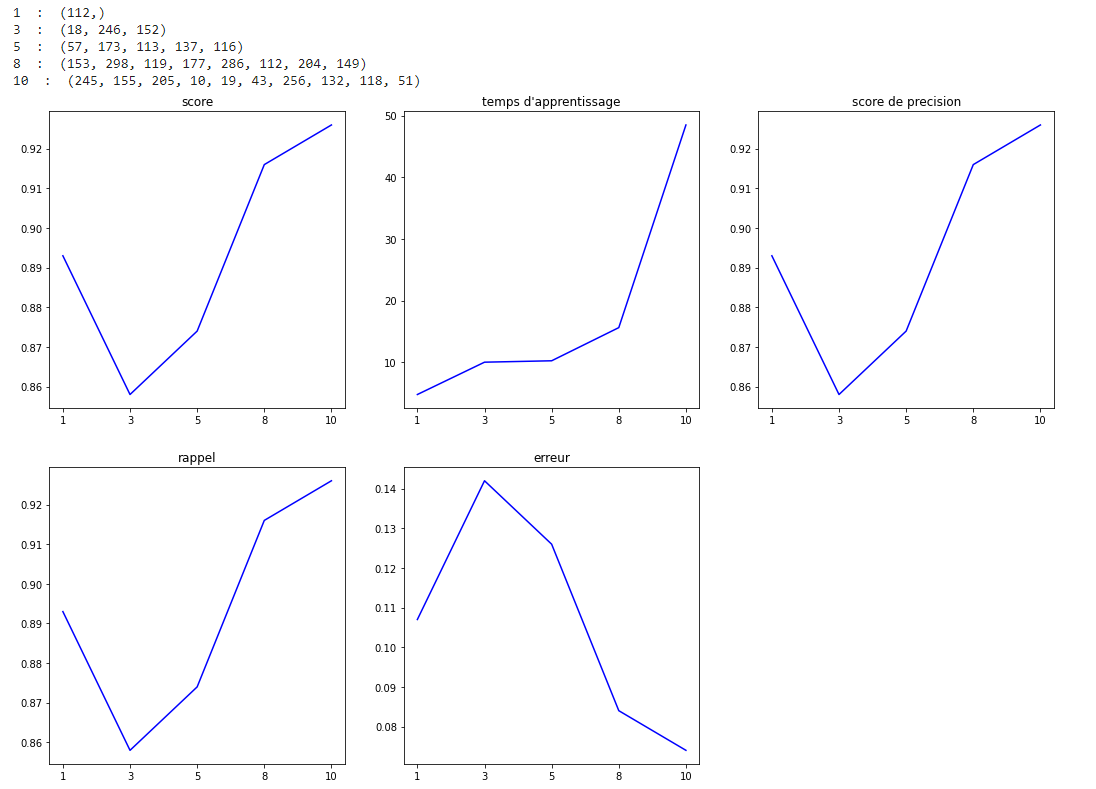
On fait maintenant varier les paramètres de MLP tout en calculant le score, la précision, le rappel, l’erreur et le temps d'apprentissage.

On fait d’abord varier le nombre de couches cachées de 1 à 100 (avec un neurone):



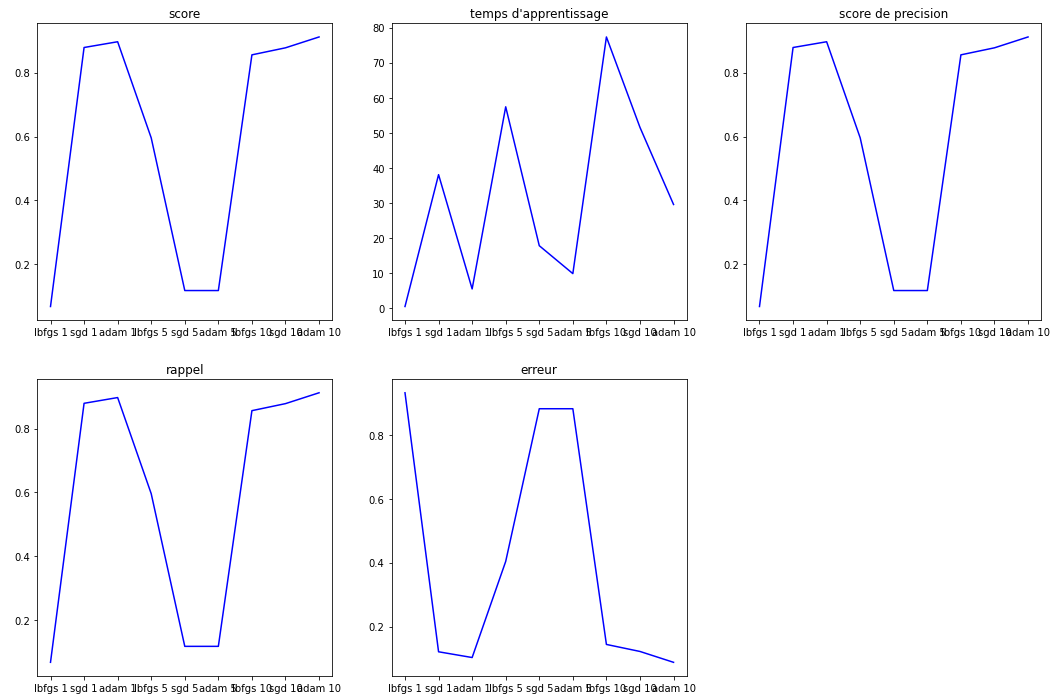
On remarque que malgré que le nombre de couches cachées augmentent, avec le même nombre de neurones (1) le score reste le même et est très faible (avec quelques pics encore plus faible pour certaines valeurs, mais qui changent quand on relance). On peut remarquer aussi que plus le nombre de couches cachées augmente, plus le temps d’apprentissage est long.

Ensuite, on construit 5 modèles de 1, 3, 5, 8 et 10 neurones avec des nombres de neurones aléatoires entre 10 et 300:



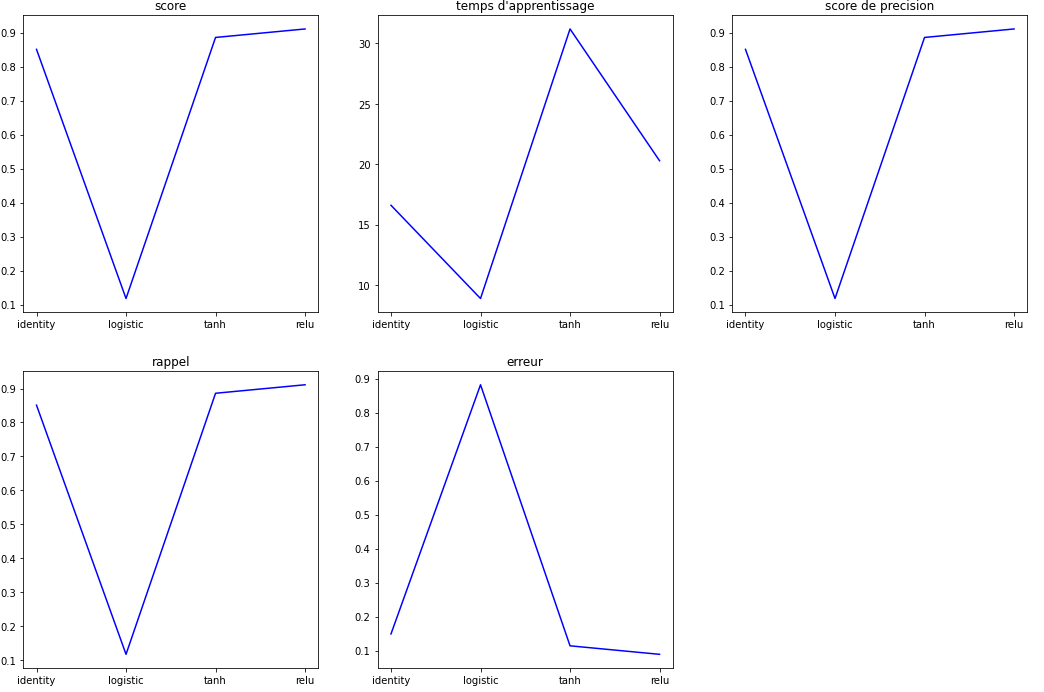
On peut remarquer que, pour des nombres de neurones suffisamment élevés, le score à tendance à augmenter avec le nombre de couches, mais ce n’est pas forcément vrai dans tous les cas (par exemple entre 1 et 3 dans ce cas là) et ça dépend du nombre de neurones de chaque couche (en relançant l’algorithme plusieurs fois on obtient pas les mêmes tendances, mais les 8 et 10 ont plus souvent des meilleurs résultats). Le nombre de couches et de neurones optimal dépend du jeu de données sur lequel est entraîné la méthode. On remarque encore que le nombre de couches influe sur le temps d’apprentissage et le fait augmenter.

On fait ensuite varier le solver entre ‘lbfgs’, ‘sgd’ et ‘adam’, en testant sur 3 réseaux de neurones de 1, 5 et 10 couches cachées (avec des nombres de neurones choisis précédemment de façon aléatoire):



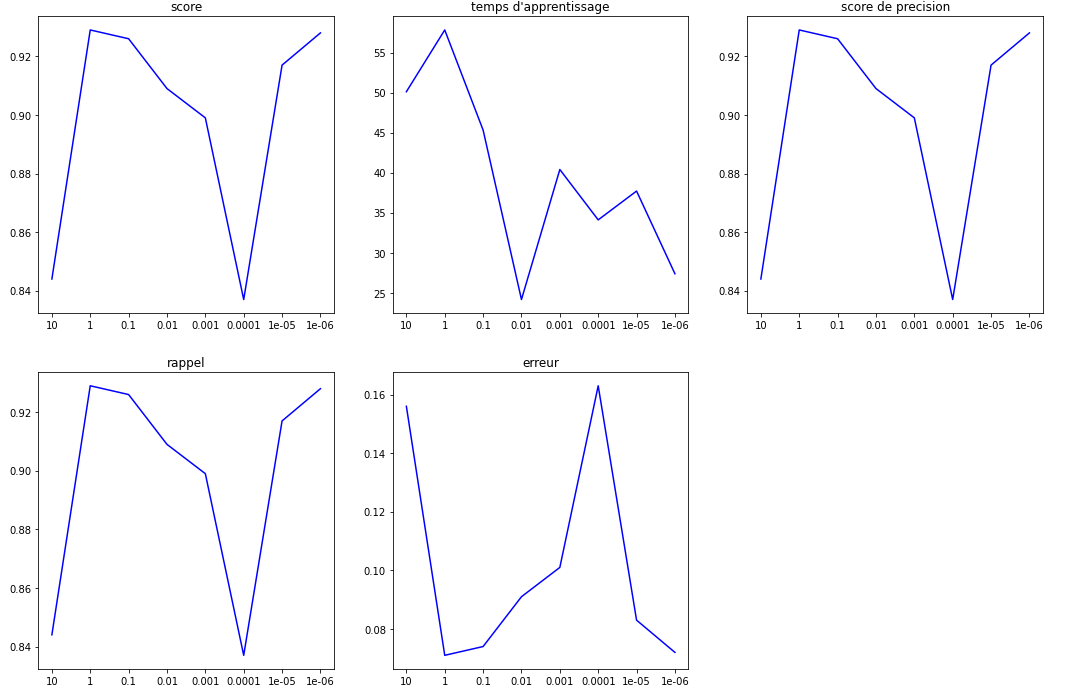
On remarque que selon le nombre de couches cachées que l’on a, ce n’est pas le même solver qui est le plus efficace. Par exemple avec 5 couches (dans notre cas) lbfgs est plus efficace que les autres mais pour 10 couches ou une couche c’est adam qu’il l’est le plus. C’est le cas aussi pour le temps d’apprentissage qui varie selon les solvers selon le nombre de couches (où le classement des 3 solvers n’est pas le même selon le nombre de couches).

On fait maintenant varier l’activation entre ‘identity’, ‘logistic’, ‘tanh’ et ‘relu’ en utilisant le set de couches cachées ayant eu le meilleur score lors du deuxième test :



On remarque que pour les paramètres choisis lors du test, identity, tanh et relu donnent de biens meilleurs résultats que logistic, avec relu qui donne le meilleur résultat. Mais comme précédemment, on ne peut pas affirmer que c’est le cas en changeant le nombre de couches et de neurones.

Pour finir on fait varier le paramètre alpha de 0.000001 à 10 par puissance de 10 :



Pour les paramètres choisis, la meilleure valeur d’alpha est 1. Mais il dépend encore une fois des autres paramètres du classifieur. On peut aussi remarquer que le temps d’apprentissage à tendance à diminuer quand alpha diminue (mais pas vrai pour toutes les valeurs d’alpha, on a notre minimum à 0.01).

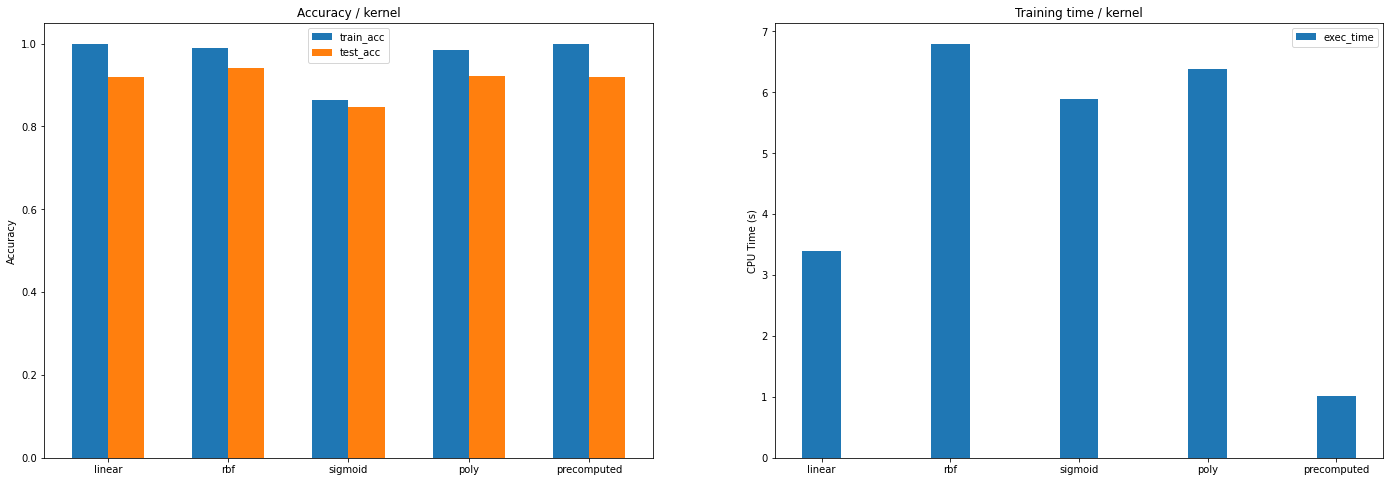
On remarque donc qu’il est très difficile avec MLP de trouver les paramètres optimaux pour maximiser le score. En effet, chaque paramètres dépendent des uns des autres dans leurs performances, du nombre de couches cachées et de neurones de chaque couche, et du jeu de données utilisé pour l’apprentissage. De plus, les combinaisons de paramètres étant infinies, il n’est pas possible de toutes les tester, et il est très dur de déduire des tendances par rapport à l’efficacité des paramètres car elles ne sont pas les mêmes selon les combinaisons choisies.

Le potentiel de la méthode est cependant très important et permet de surclasser beaucoup d’autres méthodes si les paramètres choisis sont optimaux. Le temps d’apprentissage n’est pas excessivement long (par rapport à d’autres méthodes) mais augmente avec le nombre de couches utilisées.

## **SVM - Machines à Vecteurs Supports**

* + 1. ***Varier la fonction noyaux - SVM***

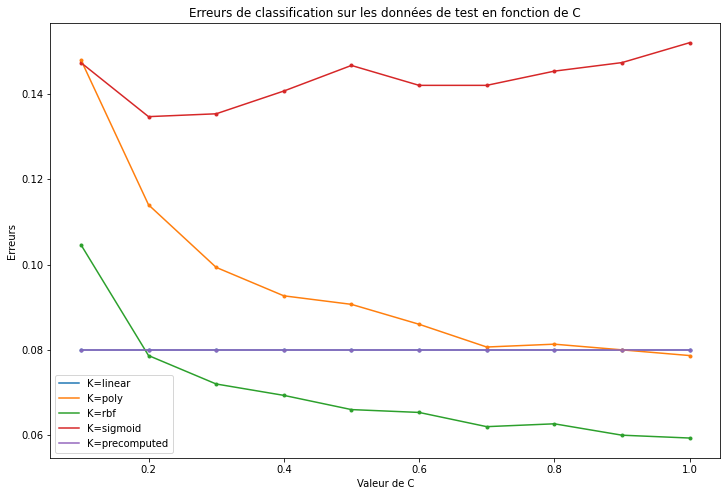
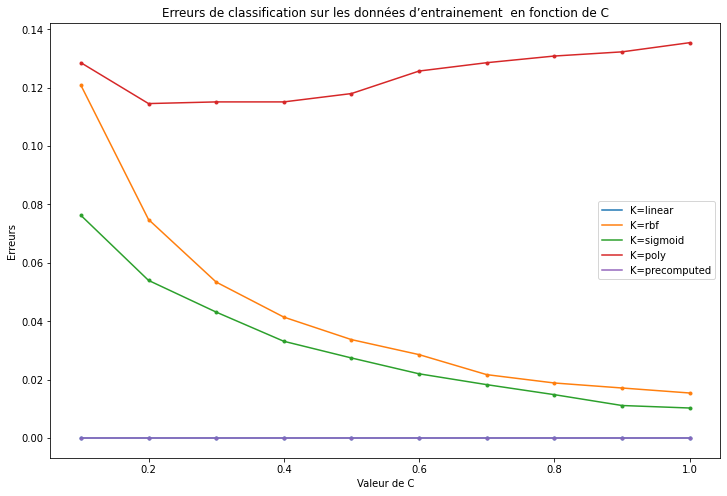
Lorsqu’on utilise des machines à vecteurs supports pour construire un modèle, le noyau - ‘***kernel***’ représente un paramètre très important. Puisque le principe du SVM est de créer des limites de décision qui séparent les différentes classes, ce paramètre aide à trouver efficacement ces limites qui sont adaptées à la façon dont les données sont distribuées à l’intérieur du jeu de données. Dans cette partie, nous voulons observer la précision et le temps d’apprentissage des modèles utilisant cinq types de noyaux différents : ***'linear'***, ***'poly'***, ***'rbf'***, ***'sigmoid'***, ***'precomputed'.***

******

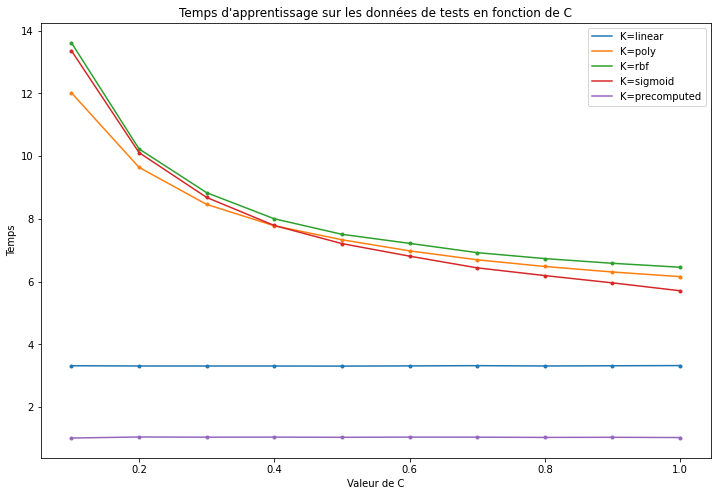
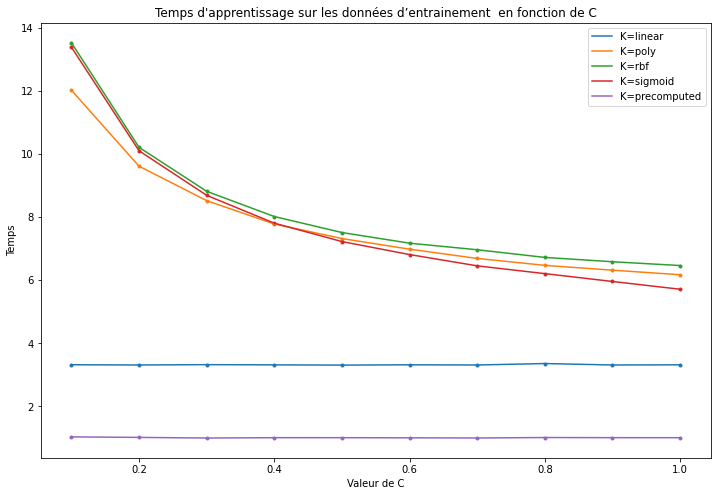
Le modèle avec le noyau ***‘rbf’*** donne la meilleure précision parmi les cinq modèles. Pourtant, comme les deux autres noyaux ***‘poly’*** et ***‘sigmoid’***, ils ont des temps d’exécution assez longs. La raison est que SVM avec un noyau non linéaire a un problème d’optimisation plus complexe avec plus variables. Ils ont donc besoin d’un grand nombre d’itérations pour résoudre ce problème et trouver les minimums globaux. SVM à noyau linéaire, en revanche, a un temps d’exécution plus court, mais sa précision n’est pas aussi bonne. Ce résultat montre que les classes du jeu de données MNIST semblent ne pas pouvoir être séparées linéairement, c’est-à-dire, elles ne peuvent pas être séparées géométriquement avec un hyperplan. Cependant, dans le cas d’un jeu de données important, de haute dimension mais assez clairsemé, le noyau linéaire deviendra très efficace car il offre un temps d'exécution beaucoup plus rapide.

* + 1. ***Varier le paramètre de tolérance aux erreurs C***

Lors de la construction d'un modèle SVM, il y a deux préoccupations : fixer une marge plus importante et réduire le taux d'erreurs de classification. À cette fin, le SVM propose un paramètre d’erreur C, qui permet d'ajuster la marge, évitant ainsi les erreurs de classification. Nous expérimentons donc ici sur 10 valeurs C différentes allant de 0,1 à 1 avec les cinq types de noyaux.



Les deux figures ci-dessus montrent les erreurs de classification sur les données d'entraînements et de tests en fonction de C. On observe que le paramètre d’erreur C n’a pas d’influence sur les modèles SVM dont le noyau est ***‘precomputed’*** et ***‘linear’***. Sur les données de test, le modèle avec noyau ***‘rbf’*** et C = 1,0 a le plus petit erreur (la plus grande précision). Cela suggère que l'hyperplan séparateur à marge optimale, qui provient de grandes valeurs de C, est très convenable au jeu de données du MNIST. Les grandes valeurs C amèneront le modèle à rechercher un hyperplan à marge plus faible, c'est-à-dire une limite de décision de forme courbe, donc mieux à classer correctement tous les points d'entraînement.



* + 1. ***Avantages et inconvénients***

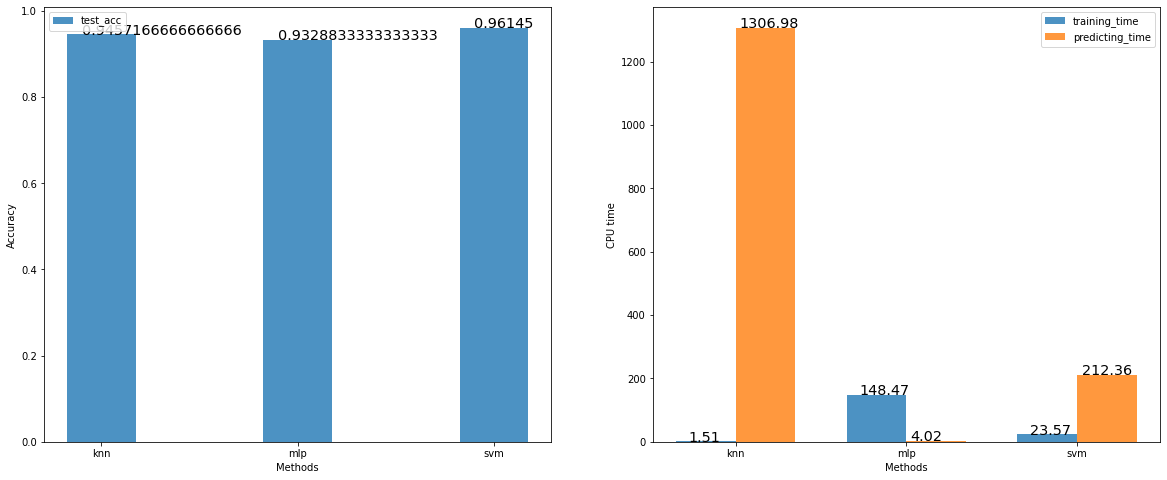
Grâce à cette expérience, nous constatons tout d'abord que le SVM est très utile pour une grande variété de données et fonctionne efficacement avec des données de faible dimension. Cependant, l'utilisation d'un noyau non linéaire dans ce cas augmentera la complexité pendant la phase d'entraînement, ce qui entraînera un temps d'exécution beaucoup plus long et même une surcharge de la mémoire. Une autre limitation du SVM est qu'il faut une étape de prétraitement des données ainsi qu'un réglage précis des hyperparamètres pour obtenir une meilleure précision.

Le SVM est très polyvalent car il fournit différentes fonctions du noyau afin d'être mieux adapté à l'ensemble de données choisi et d'améliorer la qualité des performances du modèle. Il permet également de spécifier des noyaux spéciaux lorsque cela est nécessaire. En outre, la nature du SVM est d'utiliser un sous-ensemble de points d'entraînement dans la fonction de décision - appelés vecteurs supports, de sorte qu'il est efficace en mémoire.

Néanmoins, dans le cas d'un ensemble de données dont la distribution est atypique, par exemple s'il n'est pas séparable linéairement, le choix de la fonction du noyau et du terme de régularisation est crucial, et le coût de l'utilisation d'un noyau non linéaire doit également être pris en compte.

# **Comparaison des méthodes**

## **Performance**



Pour comparer les performances des classificateurs d’apprentissage, nous pouvons nous référer à (i) la performance du classificateur lui-même, c'est-à-dire la vitesse d'exécution du constructeur du modèle ou le temps d’entraînement et (ii) la performance de l'étape de prédiction, c'est-à-dire la vitesse d'exécution du modèle lors de la prédiction

D'après nos expériences sur l'ensemble de données MNIST-784, nous nous rendons compte que les SVM et les ANN sont beaucoup plus lents que les kNN en termes de temps d'entraînement. Il y a plusieurs raisons à cela : La formation des SVM nécessite de résoudre le problème associé du double Lagrangien, qui est un problème d'optimisation quadratique avec un très grand nombre de variables. L'ANN nécessite également un temps de formation élevé, en particulier pour les modèles complexes et puissants en raison du grand nombre de couches et de neurones dans ces couches. C'est pourquoi les modèles SVM et KNN nécessitent un grand nombre d'itérations afin d'avoir une meilleure convergence vers l'optimum. Plus le nombre d'itérations est élevé, plus le temps d’entraînement est long.

Cependant, la performance de prédiction d'un SVM est nettement meilleure que celle d'un ANN ou d'un KNN. En effet, pour un MLP à 10 couches, la prédiction nécessite une propagation vers l'avant avec des multiplications successives, alors que pour un classificateur SVM, il s'agit simplement de déterminer de quel côté de la frontière de décision se trouve un point donné.

## **Précision**

En outre, un autre facteur important est la précision des prédictions, qui est utilisée pour comparer les classificateurs entre eux. Ainsi, d'après les travaux que l’on a réalisés durant ce TP , on se rend compte que les précisions de ces trois méthodes sont assez bonnes. Pour avoir un aperçu général : KNN fonctionne bien pour les petits ensembles de données (<100000 échantillons, non contextuels) en raison de son utilisation simple avec seulement deux propriétés importantes (nombre de voisins et distances). SVM fonctionne efficacement pour les petits et moyens ensembles de données ayant des caractéristiques similaires. ANN a évolué au fil du temps avec des modèles complexes et il est puissant pour les ensembles de données à grande échelle. Cependant, le SVM et l'ANN sont très sensibles aux hyperparamètres et ils nécessitent également que les données soient mises à l'échelle lors de l'étape de prétraitement avant la phase d'entraînement afin d'obtenir une meilleure précision.

En résumé, toutes les méthodes mentionnées ci-dessus (c'est-à-dire KNN, ANN ou MLP, SVM) sont trois stratégies populaires pour l'apprentissage supervisé et la classification. Cependant, il n'est pas souvent évident de savoir quelle méthode est la meilleure pour un projet spécifique. Par conséquent, lorsque nous travaillons sur un projet avec un nouveau jeu de données, il est souvent préférable de commencer par des modèles simples tels qu'un classificateur de k voisins plus proche ou un modèle linéaire, puis nous essayons de déterminer où nous devons aller ensuite. Lorsque nous pouvons mieux comprendre l'ensemble de données sur la base des méthodes simples mentionnées et de l'analyse exploratoire des données, nous pouvons passer à un meilleur algorithme capable de construire des modèles complexes tels que le SVM ou les réseaux neuronaux profonds (MLP est sa version la plus simple).

# **Conclusion**

Grâce à ce TP, nous avons découvert trois méthodes d'apprentissage machine différentes, de trois natures totalement différentes, plus précisément :

* La méthode k plus proches voisins qui est populaire car elle est très simple et intuitive à comprendre et à exécuter avec un petit ensemble de données.
* Les réseaux de neurones artificiels est une méthode très sophistiquée, capable de fournir la plupart du temps un modèle très performant pour des ensembles de données plus importants et de haute dimension, et constitue la base d'apprentissage profond.
* Les machines à vecteurs supports représentent l'approche géométrique de l'apprentissage et expriment sa complexité face à des ensembles de données complexes.

Ici, nous avons expérimenté trois techniques différentes non seulement pour voir quel est le modèle le plus optimal pour l'ensemble de données MNIST en particulier, mais nous avons aussi observé le comportement éventuel de chaque technique face à différents types d'ensembles de données. C'est pourquoi nous avons remarqué que l'analyse des données d'exploration constitue une étape cruciale avant de procéder à toute application de technique d'apprentissage machine afin d'obtenir un bon résultat en termes de performance du modèle.