INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Rapport TP Machine Learning

$5^{ème}$ année SIEC

Fernando Pinheiro de Vasconcelos
Hugo Gamero

29 Novembre 2019

https://github.com/HugoGam/Machine-Learning

Contents

1 Introduction

Dans le cadre des cours de Machine Learning à l'INSA, nous avons étudié deux méthodes d'apprentissage supervisé très connues : la méthode des k plus proches voisins et le réseau de neurones.

La première partie de ce compte-rendu est consacrée à la méthode des k plus proches voisins. Nous verrons la prise en main de la bibliothèque scikit-learn puis l'application à la classification supervisée par la méthode des k-plus proches voisins (k-nn) avec la base de données MNIST.

Deuxièmement, nous appliquerons l'apprentissage par perceptron multi-couches (les réseaux de neurones). Le but étant de déterminer une architecture MLP performante pour le jeu de données MNIST, c'est-à-dire, trouver les meilleurs paramètres du modèle.

Enfin, dans une troisième partie, nous comparerons les deux techniques d'apprentissages supervisé en montrant leur similarités et leurs différences.

2 Méthodes des k-plus proches voisins

2.1 Les jeux de données

2.1.1 Le jeu données MNIST de Google

La base de données MNIST est une base de données de chiffres écrits à la main. Les données sont alors des chiffres écrits à la main allant de 0 à 9. Une entrée est donc une image de taille fixe 28x28 (donc 784 pixels au total pour une image).

Premièrement, nous avons appris à manipuler la base de données. Voici un exemple de commandes de bases :

- `print(mnist)` affiche le jeu de données complet composé des data et de leur étiquette associée
- `print(mnist.data)` nous donne les data du jeu MNIST
- `print(mnist.target)` nous renvoie les étiquettes de chaque data (ici ['5' '0' '4' ... '4' '5' '6'])
- `print(mnist.data.shape)` qui permet d'obtenir la taille des data, ici (7000, 784)
- `print(mnist.data[0][1]) ou print(mnist.data[:,1])` et leurs dérivées permettent de sélectionner qu'une partie des data du jeu de données, data étant une matrice (7000,784)

Le but par la suite est de visualiser les données à l'aide de la librairie matplotlib de SciKitLearn.

Premièrement, comme dit précédemment, les données contenues dans mnist.data sont dans le format (70000, 784). Afin de visualiser ces données qui sont des chiffres manuscrits sous formes d'images de taille fixe 28x28, on utilise la commande `mnist.data.reshape()`

On peut par exemple afficher le premier chiffre de la base de données à l'aide des commandes suivantes :

`images = mnist.data.reshape((-1, 28, 28))` (le "-1" est necessaire pour qu'on transforme que la colonne du mnist.data)

```
plt.imshow(images[0],cmap=plt.cm.gray_r,interpolation="nearest")
plt.show()
```

Ce qui nous donne :

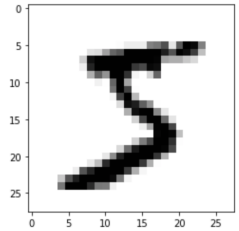

Figure 1: Représentation de l'image 0

Résultat qu'on peut aussi retrouver avec la commande `print(mnist.target[0])` qui affiche l'étiquette de la première image.

On peut maintenant essayer d'autres jeux de données.

2.1.2 Le jeu données Iris

Par exemple, nous avons pu essayer le jeu de données IRIS. Ce jeu de données regroupe les caractéristiques de 3 espèces de fleurs : setosa, versicolor et virginica. Ces fleurs sont caractérisées selon 4 propriétés : la longueur et largeur des sépales et la largeur et longueur des pétales. Ce jeu de données comporte 150 échantillons et permet donc de prédire l'espèce d'une fleur testée.

2.1.3 Le jeu données Breast Cancer

Ensuite, nous avons également découvert le jeu de données nommé Breast Cancer. Ce jeu de données est composé de prélèvements effectués sur des grosseurs au sein. Les prélèvements comportent 10 caractéristiques (rayon, aire, périmètre...) qui chacun se divisent en 3 attributs ce qui donne 30 mesures au total. Ce jeu de données permet de prédire si la grosseur est une tumeur maligne ou bénigne.

2.2 Application de la méthode K-nn

Cette méthode des k-plus proches voisins est un algorithme d'apprentissage supervisé permettant de traiter un problème de classification. C'est un apprentissage supervisé car on dispose d'une étiquette pendant l'apprentissage. Dans cette partie, nous utiliserons la bibliothèque `sklearn.neighbors`.

Premièrement, on prend un échantillon de données (data) égal à 5 000 aléatoirement à l'aide de la fonction `np.random.randint(total,size)` Ensuite, on divise la base de données en deux parties (80% pour l'apprentissage et 20% pour les tests).

On entraîne ensuite un classifieur k-nn avec k = 10. Puis on affiche la classe d'une image (par exemple la numéro 4) et on la compare à sa classe prédite.

On peut également calculer le score sur l'échantillon de test.

Enfin, on peut remarquer que le taux d'erreur sur nos données d'apprentissage et de test sont très faibles, 6.125 % et 7.1 % respectivement.

Dans cette dernière partie, on fait varier les paramètres.

2.2.1 Variation du nombre de voisins

En utilisant une boucle `for`, on fait varier le nombre de voisins de 2 jusqu'à 15. On utilise également une seconde boucle `for` à l'intérieur de la première pour avoir la meilleure précision pour chaque valeur de k (nombre de voisins). On trouve que $k = 3$ est le k optimal, obtenant une précision de 95.7 %.

2.2.2 Variation de la taille d'échantillon total

Dans ce cas, on s'intéresse à faire varier la taille de l'échantillon total (donc de training et de test) et de relever la précision. A l'aide d'une boucle for, on fait varier le nombre d'échantillons pris de 5 000 à 65 000. Pour cette analyse, on prends 80% des données pour l'apprentissage pour chaque itération.

On peut alors remarquer que nous avons une précision très bonne pour un échantillon de taille = 15 000. Ensuite, quand la taille augmente, le score diminue de plus en plus. On peut donc dire que de prendre trop d'échantillons complexifie la prédiction du classiffieur k-nn qui obitent alors une moins bonne précision.

2.2.3 Variation de la répartition des data (pourcentage pour le training)

Maintenant, on cherche à faire varier le pourcentage des échantillons (training et test) pour un échantillon de taille 5000. Pour cela, on utilise une boucle for en prenant train_size = 0.1 à 0.9 par pas de 0.1.
Puis on affiche le score de test et le score d'apprentissage pour chaque cas.

On remarque alors que le pourcentage pour l'apprentissage donnant le meilleur score est k=0.9 (soit 90% des données pour l'apprentissage et 10% pour les tests). Pour ce k, on obtient un score de test de 0.932 et un score d'apprentissage de 0.964.

2.2.4 Variation des types de distance

On cherche maintenant à faire varier les types de distance (p) de notre classifieur. Par défaut, c'est la distance Euclidienne (p=2) qui est utilisée comme métrique pour le calcul des plus proches voisins. Cependant, on peut choisir d'autres types de distance.

En effet, si on fixe le paramètre p à 1 lors de la création de notre objet de type classifieur, cela utilise la métrique nommée Manhattan distance dont la forme est la suivante : `sum(|x - y|)`. En comparant seulement ces deux types de distance, on remarque que la précision avec la distance Euclidienne est supérieure (92.9%) à celle de la distance Manhattan (91.8%).

Enfin, nous avons essayé des valeurs de p allant de 3 à 7 et nous remarquons que la précision augmente mais sans jamais vraiment dépasser celle de la norme Euclidienne. La différence notable est que le temps de calcul est beaucoup plus long, car la mesure de distance est beaucoup plus complexe.

2.2.5 Variation de n_job

On fixe maintenant n_job à 1 puis à -1 dans la définition du classifieur.
Ainsi, on ajoute ce paramètre comme ceci :

```
clf1 = neighbors.KNeighborsClassifier(n_neighbors=3, n_jobs=1)
```

On remarque alors que le temps est inférieur avec n_job = -1 (0.11s vs 0.13s environ). Ce qui est logique car n_job correspond au nombre de travaux parallèles à exécuter pour la recherche de voisins. Par défaut, si on ne spécifie pas ce paramètre, la valeur est considéré comme 1. Alors qu'en utilisant la valeur -1, on utilise tous les processeurs pour faire la recherche de voisins donc le temps est naturellement plus court.

3 Perceptron Multi-couches

Le perceptron multi-couches est un algorithme d'apprentissage supervisé fortement non-linéaire et capable d'atteindre un énorme degré de liberté, vu que nous pouvons augmenter arbitrairement le nombre des neurones dans chaque couche cachée.

Celui-ci est composé par un ensemble de pièces unitaires que l'on appelle neurones. Chaque neurone a un ensemble de poids et est relié à des entrées, qui seront responsables de le sensibiliser et, éventuellement, de le franchir.

Ce modèle possède plusieurs hyperparamètres qui devront être réglés pour que nous puissions trouver le meilleur modèle. Ces paramètres sont explicités dans l'instruction (1) ci-dessous :

```
MLPClassifier(hidden_layer_size, activation, solver, alpha, max_iter, tol)
```
(1)

Afin d'appliquer cette fonction, on va utiliser le jeu de données MNIST comportant un nombre assez grand de data (70 000). Le but était donc de faire varier tous les paramètres pour trouver l'ensemble des paramètres menant au meilleur modèle possible.

D'abord, nous avons commencé par utiliser 50 neurones dans la couche cachée et 10 neurones dans la couche de sortie, vu que nous avons 10 classes au total.

La précision du modèle obtenu sur l'ensemble de test était de 94.97%

Puis, nous avons fait varier chaque hyperparamètre :

3.1 Nombre de couches

Premièrement, le nombre de couches cachées. Pour étudier ce paramètre, nous avons fait varier le nombre de couches de 2 jusqu'à 50 couches.

Le meilleur modèle obtenu était celui avec 10 couches cachées, présentant une précision de 96.73%. Comme la fonction d'activation utilisée était la `relu`, le coût computationnel n'était pas excessif, même avec ce nombre de couches.

Le modèle avec 20 couches de 50 neurones donnait également une bonne précision (96.61%) très proche de celle du modèle à 10 couches. Cependant, le temps d'apprentissage est beaucoup plus important pour le modèle à 20 couches (75 secondes contre 41) donc nous choisissons de garder celui à 10 couches.

Lorsque le nombre de couches cachées est trop grand comme par exemple 40 couches de 50 neurones chacune, la précision du modèle sur la base de test et d'entraînement chute. Ceci s'explique par le fait que la taille de la base de données n'est plus assez grande pour la complexité du modèle. On obtient alors une précision plus petite qu'avec un modèle de 10 couches de 50 neurones qui sera plus adapté à la base de données MNIST.

De plus, l'algorithme d'apprentissage est basé sur une fonction-objectif, qui au fur et à mesure que le modèle se complique, se complique également. Donc, la fonction-objectif aura plus de minimums locaux. Or, si le nombre de données est trop faible, l'algorithme est alors piégé dans un minimum local et n'atteint pas le minimum global, ce qui explique la diminution de la précision du lorsqu'on augmente trop le nombre de couches.

3.2 Nombre de neurones dans chaque couche

Le nombre de neurones dans chaque couche est obtenu de forme très empirique. Nous avons essayé plusieurs types d'architectures, et nous avons trouvé que 50 neurones par couches était le meilleur à suivre, nous permettant de trouver des résultats de jusqu'à 96.19% de précision.

Le nombre de données d'entrées utilisé étant très important (70 000 données sur 784 entrées), il faut un nombre de neurones assez important pour obtenir une bonne précision. Sinon, le modèle ne sera pas assez précis et réalisera de trop grandes approximations.

3.3 Fonction d'activation

Les fonctions d'activation jouent un rôle très important, vu que les fonctions `tanh` et `logistic` ont un coût computationnel très grand par rapport à relu et l'identité. Donc, le choix de la fonction d'activation a un rapport direct avec le temps d'apprentissage.

Nous avons fait varier la fonction d'activation des neurones parmi les 4 types suivants : `relu, logistic, tanh, identity`.
Le meilleurs précisions obtenu avec chaque fonction étaient:

- `relu` : 96.19%;
- `logistic` : 95.04%;
- `tanh` : 93.07%;
- `identity` : 91.04%.

Ces résultats ont été obtenus avec 2 couches de 50 neurones. Nous avons remarqué que si le nombre de couches est trop important (10 par exemple), la fonction d'activation `logistic` n'est pas adaptée et fournie une précision très faible (11%)
On peut alors voir que la fonction d'activation la plus adaptée est `relu`.

3.4 Nombre maximum d'itérations

Au début, quand on variait les fonctions d'activations le nombre max d'itération par défaut (fixé à 200) n'était pas suffisant. Nous avons eu besoin de l'augmenter jusqu'à 400, pour que les algorithmes puissent converger.

3.5 Alpha (valeur de la régularisation L2)

Ensuite, nous avons cherché à déterminer la meilleure valeur pour le paramètre alpha, le terme de régulation.

A l'aide d'une boucle `for`, nous avons pris un modèle à 10 couches de 50 neurones puis nous avons fait varier la valeur de alpha de 0.0001 (valeur par défaut) à 0.1.

Nous avons alors pu remarquer que le meilleur résultat en terme de précision était pour des valeurs de alpha fixées à 0.01 (97.1%) ou 0.05(97.16%). Afin de les départagées, nous nous sommes basés sur le temps d'apprentissage qui est plus court pour un alpha à 0.01 (66 secondes vs 77 secondes).

Nous choisissons donc `alpha` = 0.01.

3.6 Solveur

Quand on fait varier le paramètre `solver`, le nombre d'itérations change ainsi que le temps d'apprentissage. Le meilleur solveur entre les 3 était le `adam`, en général, c'était lui qui finissait de converger plus rapidement que les autres. Le `lbfgs` et le `sgd` avaient une performance qui variait en fonction de la complexité du modèle et de la fonction d'activation. Parfois, un était meilleur que l'autre, mais ce n'était pas possible de les classifier.

3.7 Modèle choisi

Finalement, après avoir testé expérimentalement ces hyperparamètres, nous avons conclu que les meilleures valeurs étaient les suivantes :

- `nb_couches` : 10;
- `activation` : `relu`;
- `nb_neurones_par_couche` : 50;
- `alpha` : 0.01;
- `solveur` : `adam`;
- `max_iter` : 400;

Pour obtenir ces résultats, nous avons également cherché à combiner les différents paramètres ensembles à l'aide de plusieurs boucles `for` imbriquées les unes dans les autres.

4 Comparaison des modèles

4.1 Points communs

4.1.1 Apprentissage supervisé

Tout d'abord, les deux modèles précédemment vus (k-nn et réseau de neurones) fonctionnent par apprentissage supervisé.

Pour redéfinir ce qu'est un apprentissage supervisé, c'est un type d'apprentissage de l'intelligence artificielle où la base de données est étiquetée. En effet, l'algorithme est alors entraîné à effectuer une tâche à partir de bases de données étiquetées : des données qui sont associées à des mots qui les désignent.

Tandis que pour un apprentissage non supervisé, l'algorithme apprend seul en "regardant" les données, sans que celles-ci ne soient étiquetées. Le travail du modèle est alors plus complexe.

4.2 Différences

4.2.1 Temps de calcul du réseau de neurones

Les réseaux de neurones présentent un temps de calcul beaucoup plus grand que le k-nn. Cela vient du fait que la complexité du modèle peut être plus élevé que celle du k-nn. Dans le TP, pour améliorer la précision du modèle, nous avons essayé de le complexifier, en ajoutant plus de couches et en augmentant le nombre de neurones par couche, ce que augmentait toujours le temps d'apprentissage.

De plus, le choix des fonctions d'activations avait un rôle important dans le coût computationnel, vu que les fonctions `logistic` et `tanh` présentent un coût très élevé.

4.2.2 Complexité à choisir les hyperparamètres du réseau de neurones

Le réseau de neurone possède plusieurs paramètres qui sont difficiles à régler, vu que plusieurs d'entre eux sont interdépendant, comme la fonction d'activation et le alpha. Tous ces paramètres sont à régler minutieusement de sorte à avoir une précision la meilleure possible.

Or, la méthode des k plus proches voisins est beaucoup plus simple à paramétrer : il suffit d'indiquer le nombre de voisins qu'on veut considérer.

4.2.3 Simplicité de mise en place de la méthode k-nn

Comme vu lors de la première partie, l'algorithme des k-plus proches voisins est très simple à utiliser. Il est très intuitif.

Nous avons juste à lui spécifier le nombre de plus proches voisins sur lesquels on veut qu'il se base pour ensuite effectuer l'apprentissage.

4.2.4 Taille du dataset

Enfin, le réseau de neurones a besoin d'un grand nombre de données pour être précis. Nous avons pu entraîner un réseau de neurones avec 70 000 données (sur 784 entrées) et nous avons vu que ce nombre est parfois pas suffisant lorsque le nombre de couches est trop élevé.

La méthode des k plus proches voisins peut quand à elle, fonctionner avec des jeux de données de taille très faible (200 ou 500 par exemple) et donner de très bon résultats.

Le choix de l'algorithme utilisé dépend donc de l'application pour laquelle on souhaite l'utiliser. Dans notre cas de la base de données MNIST, la méthode k-nn semble plus appropriée.

5 Conclusion

Pour conclure, nous avons pu découvrir deux techniques très utilisées dans les applications de Machine Learning : la méthode des k plus proches voisins et le réseau de neurones.

Nous avons pu nous rendre compte des avantages et inconvénients de chacune de ces méthodes. La méthode k-nn est simple d'utilisation et permet d'obtenir de très bons résultats. Le paramètre principal qui est le nombre de voisins est facile à configurer et cette méthode permet d'obtenir des temps d'apprentissages très faibles.

Le réseau de neurones quand à lui, est plus complexe à paramétrer mais peut mener à une meilleur précision que la méthode k-nn. Cependant, la manière expérimentale pour trouver les bon paramètres et son temps de calcul plus long sont ses principaux inconvénients. Tout dépend donc de l'application souhaitée.