Classification des données MNIST à deux digits

Une comparaison de plusieurs méthodes

Équipe

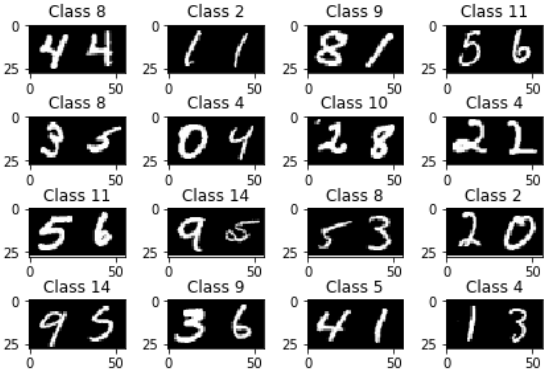
Nom de l'équipe : Othmane&Lucas

Membres :

- Lucas Gaspaldi (matricule : 20177597)
- Othmane Sajid (matricule : 20135964)

Introduction

Les données MNIST, parfois considérées comme le "Hello World" de l'univers du Machine Learning, présentent un défi intéressant pour l'entraînement d'algorithmes de classification. Les données MNIST utilisées habituellement sont des images d'un seul chiffre (de 0 à 9) avec le label correspondant. Toutefois, pour les données avec lesquelles ce projet a été réalisé, les images contiennent non pas un seul chiffre mais plutôt deux chiffres. Les labels sont donc au nombre de 19 et vont de 0 à 18 (la somme deux chiffres).
Ceci ajoute évidemment une difficulté supplémentaire. Le nombre de labels étant presque le double, la tâche de classification en devient d'autant plus complexe.



Nous avions donc un jeu d'entrainement de 50 000 observations, composées du couple (image, label). Chaque image est représentée par 1568 features, qui sont les valeurs prises par chaque pixel dans un format d'image de 56x28 pixels. Il s'agissait alors d'entraîner un modèle sur ce jeu d'entraînement, de sorte que le modèle puisse ensuite classifier efficacement 10 000 images du jeu de test dont le label est inconnu.

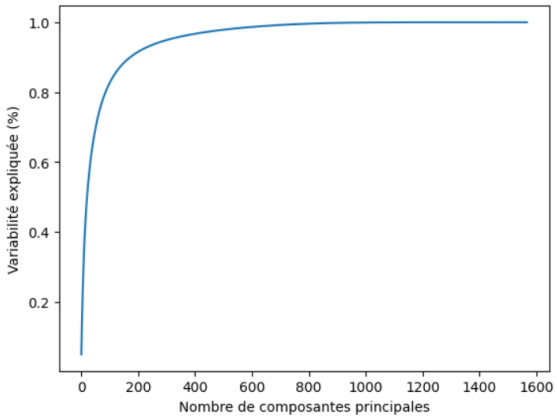
Pour cette tâche de classification, nous avons testé la régression logistique en plus de plusieurs autres modèles. Il s'en est dégagé que la régression logistique n'est pas très adaptée à ce jeu de données pour plusieurs raisons : avec le grand nombre de features (1578) et d'observations du jeu d'entraînement (50 000), l'entraînement prend un temps considérable. Également, la performance de la régression logistique est loin d'être optimale, puisqu'elle avoisine les 20-25% (quoi qu'elle peut être légèrement supérieure avec une meilleure implémentation, comme celle de la librairie Scikit-learn, qui a donné un score d'environ 34% sur un jeu de test isolé sélectionné aléatoirement dans le jeu de données).

D'autres modèles ont permis des performances significativement meilleures. Notamment, les réseaux de neurones sont parfaitement adaptés à ce type de problème et nous ont permis d'avoir un score de 95,7% sur le jeu de test.

Feature design et data preprocessing

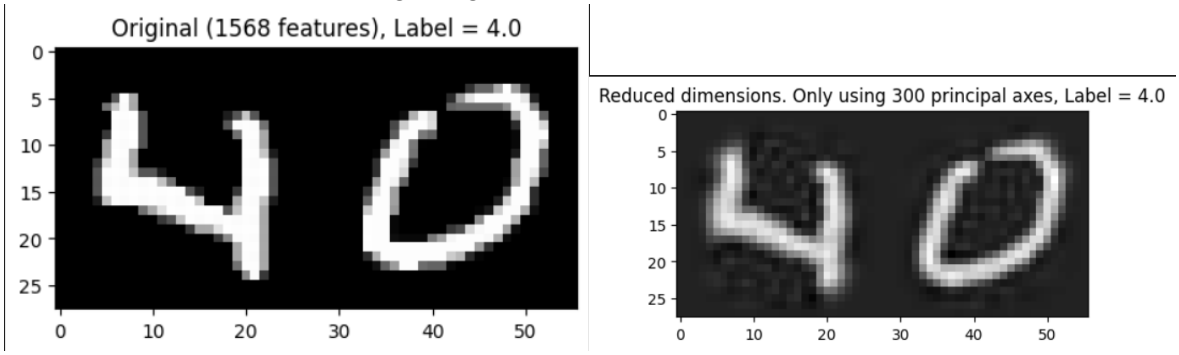
Dans l'étape d'analyse préliminaire, on a centré les features et effectué une Analyse en Composantes Principales (PCA) pour observer les relations au sein des features. Il en est ressorti notamment que la réduction du jeu de données à 300 composantes principales (plutôt que les 1568 features initiales) permettait de conserver quelque 95% de la variabilité, tout en divisant le nombre de variables par 5.

Figure - Variabilité expliquée (%) selon le nombre de composantes principales (PCA)



En effet, en conservant seulement 300 axes principaux (95% de la variabilité), les images conservent somme toute leurs structures, comme on peut le voir dans l'exemple suivant où l'on voit affichées, respectivement, l'image originale et celles avec les dimensions réduites.

Figure - Comparatif d'une image originale et de sa représentation en 300 axes principaux



En plus de la valeur descriptive de l'ACP effectuée, nous avons tenté d'effectuer la régression logistique avec 300 composantes plutôt que les 1568 features mais ceci n'a pas eu un impact significatif sur la qualité des prédictions. Nous avons donc abandonné cette démarche et avons privilégié d'effectuer la régression logistique avec les données originales.

Néanmoins, pour le modèle Naive Bayes discuté plus loin, nous avons entraîné le modèle sur les composantes principales et ceci a joué un rôle central dans la qualité de prédiction du modèle.

Overview des algorithmes utilisés

Nous avons expérimenté avec plusieurs modèles de classification. Notamment, nous avons utilisé :

- Régression logistique
- K-NN
- SVM (Support Vector Machine)
- Naïve Bayes (combiné avec PCA)
- Réseaux de neurones avec diverses architectures.

On a constaté une grande diversité dans les performances. Certains de ces modèles (et notamment avec les bons hyperparamètres) sont beaucoup plus performants que d'autres. Le travail que nous avons effectué a été fort utile pour comprendre à quel point il est essentiel de choisir un modèle adapté aux spécificités du jeu de données.

Méthodologie

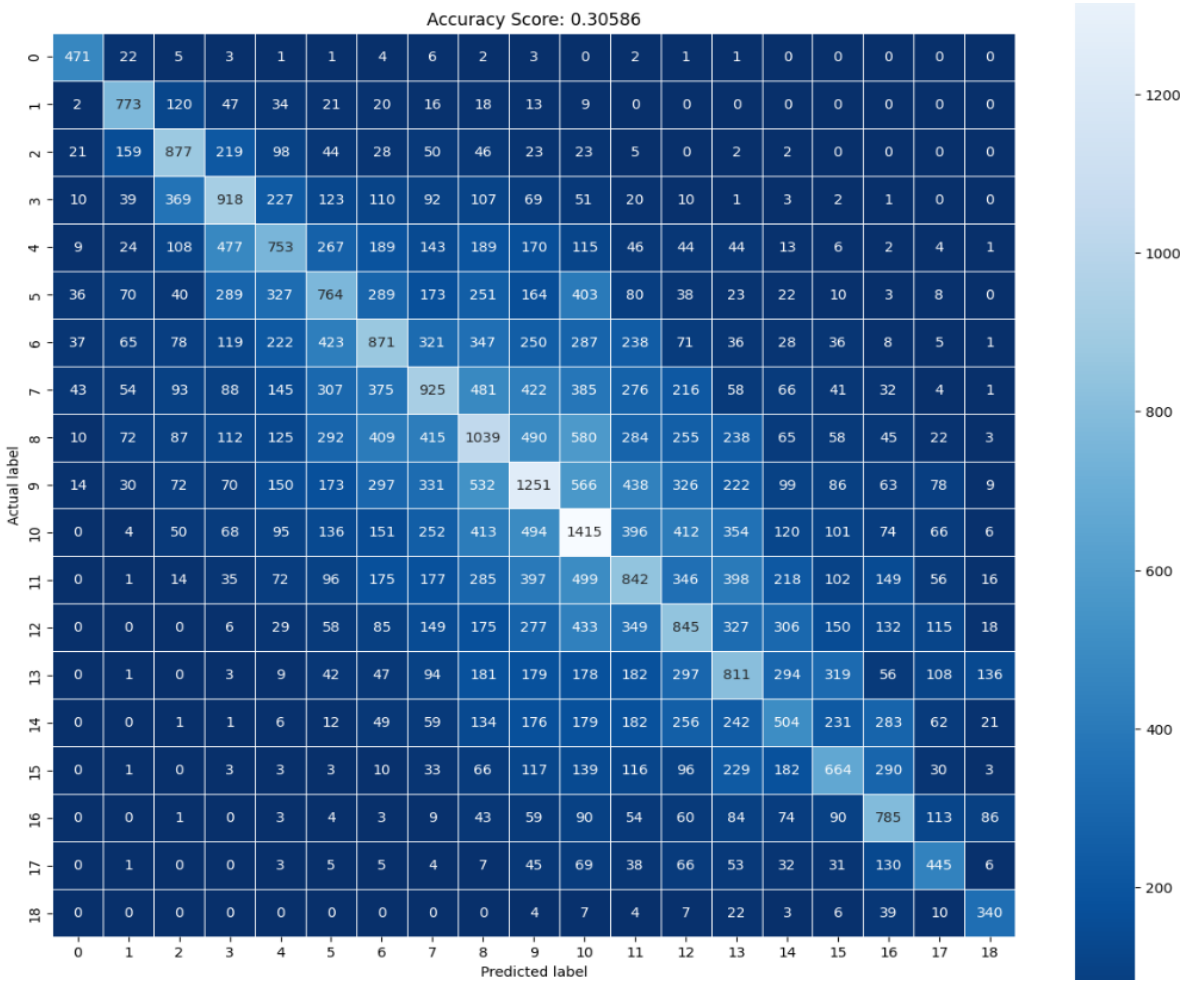
Pour chacun des modèles, lors de l'étape d'optimisation, nous avons partagé le jeu de données d'entraînement en un ensemble d'entraînement, et un ensemble de validation (et dans certain cas un ensemble de test aussi). Le partage était généralement 70-30. Le jeu de validation a permis notamment de choisir les hyper paramètres optimaux et d'améliorer in fine la performance du modèle lors de l'étape de test.
N. B. Ces optimisations sont vues en détail dans les parties dédiées aux modèles utilisés.

Résultats

Régression logistique

Pour la régression logistique que nous avons implémenté, en fonction des paramètres de la descente de gradient (nombre d'itérations, "Learning rate" et paramètre mu de régularisation), nous obtenions des scores d'environ 20-25% sur un jeu de test isolé, et d'environ 30% sur le jeu d'entraînement lui-même. Nous avons fait de la cross-validation pour optimiser les paramètres de descente de gradient.
Nous avons également tenté d'améliorer ces performances en faisant du Feature Design (notamment à l'aide d'une réduction PCA), mais cela n'a pas été très significatif. Nous avons donc préféré garder les données originales. Avec un grand nombre d'itérations, nous avons obtenu un score de 22.48% sur le jeu de test de Kaggle, battant ainsi le baseline de régression logistique.

Mentionnons quelques difficultés avec la régression logistique. Pour l'entraînement sur le jeu d'entraînement complet, le travail computationnel est important. La descente de gradient prend beaucoup de temps, et le grand nombre de features (1568) et de classes (19) n'aide pas du tout les choses. Il est clair que la régression logistique n'est pas bien adaptée à cette tâche de classification. Il n'est donc pas étonnant qu'elle a été la moins performante des 5 modèles que nous avons utilisé ici. On a remarqué que la régression logistique parvenait à bien classer certains labels, mais que d'autres, moins bien, notamment ceux qui se ressemblent. Le classement sur le label "0" par exemple était assez bon. On peut le constater dans la matrice de confusion



SVM

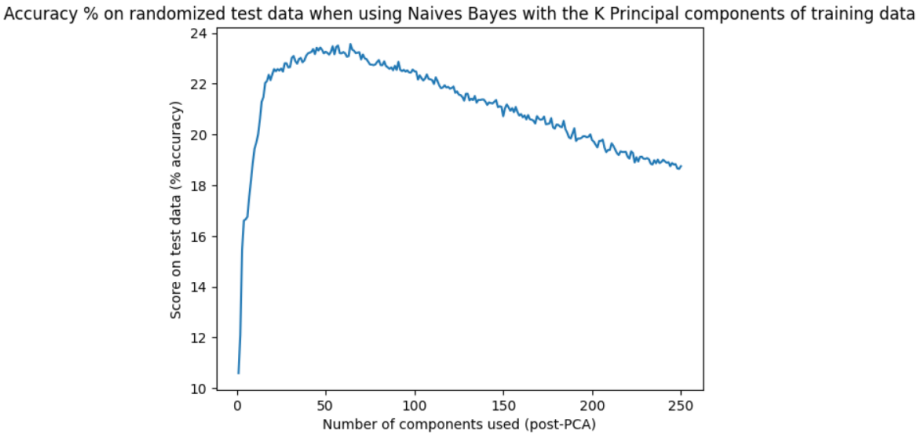
Les SVM sont réputés être efficaces sur les datasets avec un grand nombre de features, et sont “memory-efficient”. C’est pourquoi nous avons essayé ce modèle. Nous avons divisé le jeu de données en un ensemble d'entraînement (70% des données) et un ensemble de validation (30%). Nous avons expérimenté avec 3 kernels : radial gaussien (rbf), polynomiale et sigmoid (sachant que sigmoid allait être inadapté, nous voulions voir à quel point).
Les résultats ont été très dispersés et le kernel polynomial a fait les meilleures prédictions.

Kernel utilisé pour le SVM	**Score sur le jeu de validation**
RBF (gamma = 0.05, C=5)	44.57 %
Poly	73.96 %
Sigmoid	6.5 %

Généralement, la standardisation des données (centrées-réduites) permet d’améliorer les performances des SVM. Nous n’avons toutefois pas constaté d’amélioration en standardisant les données pour ce dataset.

Naive Bayes

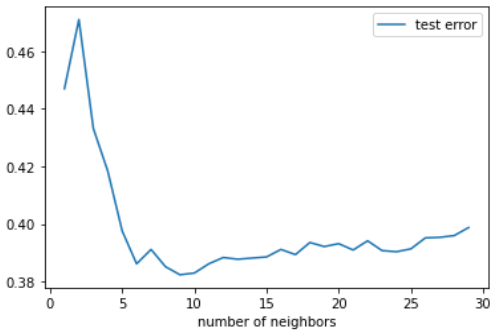
On a commencé par séparer les données en un jeu d'entrainement (70%) et un jeu de test (30%). Dans un premier temps, nous avons effectué la classification par Naive Bayes directement sur les données sans aucune transformation. Le résultat était médiocre et de seulement 7.4 % sur le jeu de validation. Nous avons alors choisi de combiner ce modèle avec une PCA. Pour le choix du nombre de composantes optimales, on a procédé par cross-validation.



On a vu ainsi qu'en conservant 64 composantes principales, on obtient la meilleure performance avec Naive Bayes. Le score était alors de 23,56 %. Il y a donc eu une augmentation significative en combinant la PCA avec Bayes, pour passer d'un faible score de 7,4 % à 23,56%, ce qui démontre bien l'importance de l'étape de Feature design.

KNN

L'avantage des k- plus proches voisins est que c'est un algorithme très simple, et malgré sa simplicité il peut très bien être capable d'apprendre des systèmes non linéaire (voronoi diagramme). On a donc essayé de l'implémenter sur les données MNIST pour voir ce qu'il était capable de prédire. Nous avons commencé par un KNN avec un k = 1 parce que ça paraissait avoir les meilleurs résultats en prenant une toute petite partie des données. Puis on s'est vite rendu compte qu'en agrandissant le jeu données, certaines valeurs de k avaient des meilleurs résultats. Ce qui nous lança dans la recherche du meilleur hyperparametre k.



-40000 données d'entrainement
-10000 données de validation

On voit donc que le k optimal est **k = 9**

Obtention d'une erreur similaire (0.35 d'erreur) sur kaggle avec les données de test

Nous avons testé que sur un nombre de voisin allant de 1 à 30, donc notre k optimal est dans cet intervalle, mais c'est très probable que celui soit notre k optimal global. Même si nous avons trouvé le k optimal, pourquoi nous n'avons pas eu de meilleurs résultats ?

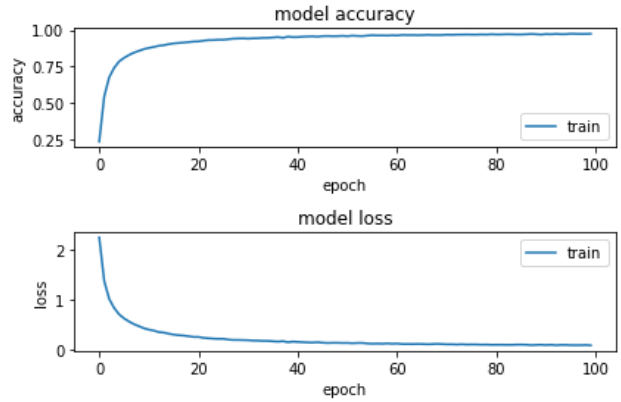
C’est surement parce pour un jeu comme MNIST avec un nombre de dimension assez élevé, le fléau de la dimension agit considérablement sur les distances euclidiennes entre les points.

Neural Network

D’après les benchmarks, les meilleurs résultats obtenus sur MNIST venaient de modèles basés sur des réseaux de neurones. Il fallait donc à tout prix, si on voulait avoir très peu d’erreurs sur nos prédictions, essayer nous même un réseau de neurones sur MNIST. Après avoir trouvé certaines ressources pour nous guider sur les types d’architectures et des informations complétaires, nous sommes arrivés à faire les prédictions :

Hyperparametres au départ:
Nombre de Hidden layers : 3
Nombre de neurones sur les hidden layers : 172 (en suivant le calcul sqrt(input layer nodes * output layer nodes))
Fonctions d’activations : -Relu pour les hidden layers (conseillé dans le cours à la place de la sigmoid)
-Softmax pour l’output

Test sur des données de validation:



Hyperparamètres de base, 100 epoch:Accuracy sur données de validation : 0.915066659450531

Notre première impression des réseaux de neurones est que c’était incroyablement rapide, en quelques secondes on pouvait avoir déjà plus de 90% de succès sur nos données de validation. Pourtant aller chercher le 99% n’était pas une tâche si facile, mettre plus d’epoch pour essayer de l’atteindre ne servait à rien, on arrivait juste à de l’overfitting
Donc bien sur il fallait qu’on joue avec les hyperparametres :

En essayant de changer le nombre de neurones des hidden layers :
100 : 0.8896 | 200 : 0.9226
300 : 0.9303 | 500 : 0.9391
600 : 0.9367 | 700 : 0.9308
800: 0.9434 | 900 : 0.9409
1000: 0.9398

On trouve une erreur minimale à 800 ->
0.9532 sur données de test avec 800 neurones sur les hidden layers et avec 400-500 epoch

Il était certainement très possible d’aller au delà de 99% mais notre manque de connaissance de chacun des hyperparametres et ne pas savoir comment trouver les meilleurs d’entre eux en un temps qu’il ne nous restait pas nous a empecher d’aller plus loin.

Discussion

La régression logistique n'obtient pas un résultat très bon pour notre classification sur nos données Mnist. Quelques hypothèses sur la mauvaise performance sont le grand nombre de features et de classes, ainsi que le fait que ces données MNIST sont fort probablement non linéairement séparables.

Nous avons mis en œuvre plusieurs techniques pour optimiser nos modèles, telles que la cross-validation, du Feature Design (notamment PCA), de la régularisation (L2 pour la régression logistique, etc.). Un enjeu important était de limiter l'overfitting pour maintenir les capacités de généralisation de nos modèles. Il est fort probable que l'on aurait pu obtenir des gains de performance avec des meilleures techniques de Feature Design, un aspect que nous n'avons pas suffisamment développé.

En effet, pour ce qui est des limites de cette étude, il importe de mentionner que notre approche de tester un grand nombre de modèles a certes eu un grand apport pédagogique pour nous, mais en vertu duquel nous avons possiblement sacrifié l'optimalité de la performance. Nous avons expérimenté avec la régression logistique, SVM, KNN, Naive Bayes et les réseaux de neurones, mais pour chacun de ces modèles, nous avons effectué des analyses intéressantes sans forcément chercher à optimiser la performance au maximum. Si nous avions plus de temps ou si nous avions plutôt focalisé notre attention par exemple sur un seul ou bien deux algorithmes seulement, nous aurions pu davantage améliorer leur performance (avec par exemple un meilleur "Feature engineering", plus de régularisation, optimisation poussée des hyper-paramètres, etc.). Le choix de privilégier la "quantité plutôt que la qualité" était essentiellement pour avoir la chance de tester tous ces modèles intéressants et de comparer leurs performances avec des implémentations plus ou moins élémentaires; c'était un choix pédagogique.

Liens Google Collab pour le code

Analyse préliminaire et régression logistique :
https://colab.research.google.com/drive/19PQNHsCtU_bMeHrdPgOgPAUQN3sehldD?usp=sharing

Tous les autres modèles :
https://colab.research.google.com/drive/17v-XDT_FbhuqLrY1HR78mnpwgrC4NoSE?usp=sharing

Division des contributions

Pour la rédaction du rapport, nous l'avons fait ensemble. Pour ce qui est des modèles, nous avons séparé les tâches. L'un de nous s'est concentré sur l'écriture du code de la régression logistique pendant que l'autre a testé divers modèles et tentait d'obtenir la meilleure performance avec un réseau de neurones. Somme toute, le partage du travail d'équipe était égalitaire et complémentaire et nous avons fait du version control via GitHub.

Nous déclarons par la présente que tous les travaux présentés dans ce rapport sont ceux des auteurs.

Références

Voici quelques articles dont nous nous sommes inspirés pour la compréhension/et ou astuces pour l'implémentation des différents modèle et/ou leur optimisation.

https://www.pycodemates.com/2022/03/multinomial-logistic-regression-definition-math-and-implementation.html (bonne explication détaillée de la régression logistique)

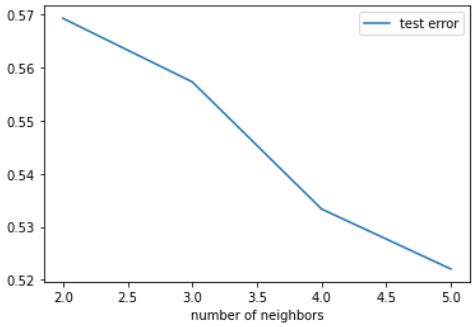
Inspiration pour Bien choisir les nombres de neurones: https://www.linkedin.com/pulse/choosing-number-hidden-layers-neurons-neural-networks-sachdev/

Pour le One-hot encoding :

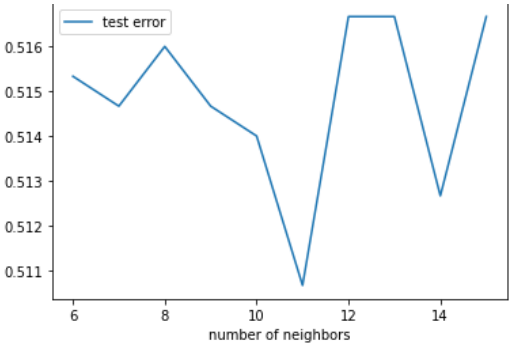
https://stackoverflow.com/questions/38592324/one-hot-encoding-using-numpy

Annexes

Premiers essais de recherche du meilleur k:



1er essai sur petit échantillon:
- 4000 données d'entrainement
- 1000 données de validation



On se rend compte que ça décroit, donc on exécute sur plus de voisins :

On trouve un optimal **k = 11**