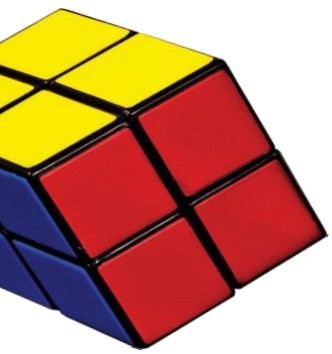


Rubik's Cube

PROJET DE MACHINE-LEARNING

SARI Mohand | SEBBAN Samuel | 2021/2022

1) INTRODUCTION

Lors de ce projet de Machine Learning (Rubik's Cube Data Challenge 2019), nous avons à notre disposition des jeux de données d'une grande taille (x_train =1.837.079 données ; y_train= 1.837.079 données ; x_test = 1.837.080 données) et qui nous ont causé de nombreux problèmes principalement au niveau du temps de compilation de chaque algorithme.

Cependant, nous avons donné notre maximum pour proposer des solutions et réaliser des prédictions à partir des différents algorithmes vus durant les Tps en venant même parfois à approfondir sur certains sujets.

Nous allons ainsi nous demander si nous avons assez de données à disposition dans notre x_train pour pouvoir entraîner le modèle, s'il est possible de faire de bonnes prédictions et si ces données sont de bonnes qualités.
Aussi nous pouvons nous demander s'il y a un meilleur algorithme pour prédire ce jeu de données et quels seront les bons paramètres de cet algorithme pour rendre plus performant le modèle.

2) VERIFICATION DES DONNEES

Peut-on supprimer un ou plusieurs attributs pour ce problème ?

Tout d'abord intéressons-nous aux différents attributs de notre ensemble x_train.
Concernant ses attributs, il nous est impossible d'en éliminer pour optimiser les prédictions, chacun des attributs est pertinent car ils correspondent aux couleurs des carrés présent sur chaque face.

La fonction verifdf(df)

Nous allons ensuite voir si nos jeux de données sont complets à l'aide de la fonction implémentée verifdf(df) dans notre dossier Jupyter qui vérifie si les données dans chaque colonne sont des entiers et qu'il n'y a pas de valeurs manquantes.
Après vérification, les jeux de données présents sur le site sont de bonnes qualités et on va pouvoir les utiliser sans les modifier.

One-Hot-encoding

Concernant les données, nous pouvons voir que pour chaque attribut des ensembles x_train ou x_test, sont attribuées des valeurs de 1 à 6 pour préciser les couleurs de chaque position sur le Rubik's Cube.

Or, de manière évidente, nous pouvons remarquer qu'il n'y a pas de relation d'ordre pour les couleurs ainsi à partir de ces valeurs attribuées à chaque couleur on pourrait dire par exemple qu'une couleur est supérieure à une autre, ce qui n'est pas cohérent et va peut-être fausser des prédictions pour certains algorithmes.

Pour pallier cela, la solution que nous proposons est le One-Hot-Encoding .
Elle permettra peut-être d'obtenir de meilleures performances et gagner en précision pour faire certaines prédictions.
Cette technique consiste à coder pour chaque couleur une matrice avec des 0 et un 1 pour signifier à quelle couleur correspond chaque position (voir l'exemple en Annexe).

Distribution des étiquettes :

Nous avons aussi remarqué que les observations de notre y_train ne sont pas réparties de manière à peu près égale. Cela peut donc perturber la précision de chaque modèle (voir la répartition en Annexe).

3) LE TRAVAIL REALISE

Nous avons décidé de réduire notre Trainset à l'aide de la fonction split (de skicit-learn) même si l'on sait que cela va réduire dans une moindre mesure les performances car on va entraîner sur moins de données.
Par ailleurs, de manière générale, le split divise notre Trainset en deux sous-ensembles :90% pour l'entrainement et en 10% pour le test.

Le but étant d'anticiper les performances de chaque modèle et sa précision.
Dans notre Trainset, nous avons des données déjà étiquetées donc cela va bien sûr nous aider à voir si un modèle est précis mais surtout à appliquer les différents algorithmes.

Après avoir fait des prédictions sur des données déjà étiquetées on pourra alors calculer la précision du modèle (accuracy,precision, recall,score F1).
Nous afficherons la matrice de confusion permettant de voir de manière plus précise les erreurs d'étiquettes pour les prédictions.

De plus, pour les algorithmes les plus performants, nous avons essayé d'utiliser le One-Hot-Encoding et cela nous a donné les mêmes résultats sur certains algorithmes et des résultats moins bons sur d'autres.
Nous avons donc décidé de ne pas l'appliquer sur tous les algorithmes (cf code sur Jupyter)

Nous avons également décidé de rééchantillonner les données (NearMiss de imbalanced-learn) d'entraînement car les données ne permettait pas aux algorithmes d'apprendre correctement étant donné que les nombre de coups { 10,11,12 } était sur représenté contrairement aux autres. Malheureusement, cela ne donne pas de meilleur score de rendu sur le site du challenge. Nous n'avons pas donné de suite à cette méthode.

A partir, des éléments cités précédemment que nous pouvons récolter, nous réduisons notre ensemble d'entrainement (cf fonction split) afin de pouvoir anticiper les performances des différents modèles et de pouvoir soumettre des prédictions sur le site du Challenge. Plus notre score se rapprochera de zéro, plus nos prédictions seront satisfaisantes.

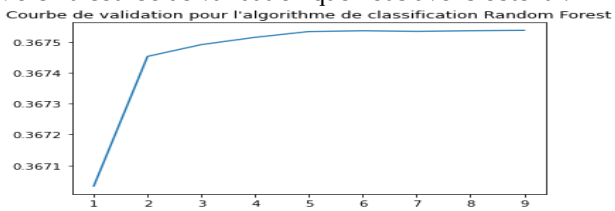
Nous allons maintenant expliquer les différents algorithmes utilisés pour ce projet. Nous avons décidé de nous focaliser et d'expliquer dans le détail un des algorithmes (Random Forest/ Réseau de neurones …) qui nous a donné le meilleur score sur le site du challenge.
Nous avons alors décidé de présenter en détail ce que nous faisons pour l'algorithme Random Forest.

4) UTILISATION DES ALGORITHME

a) Random Forest

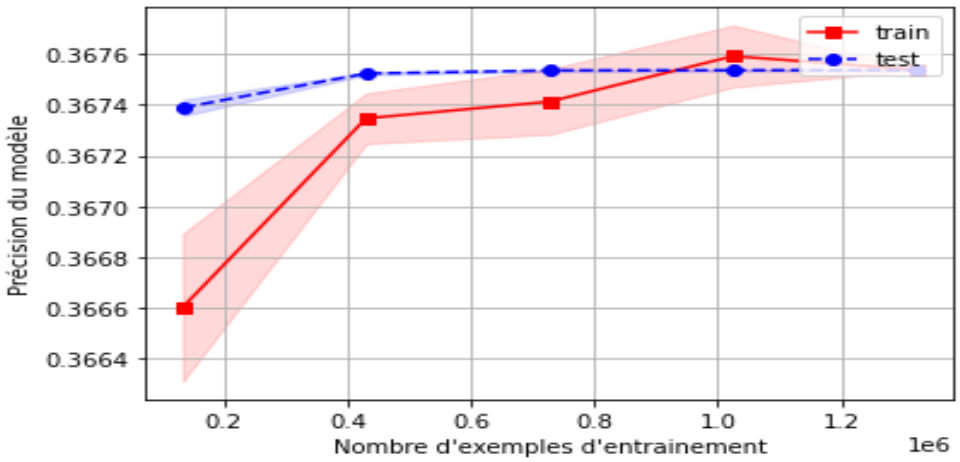
Après soumission sur le site du challenge, nous avons l'affichage suivant :
Random-Forest -> Your submission score is : 0.8589076142574085.
L'algorithme de Random Forest est un algorithme de classification basé sur différents paramètres (n_samples , max_depth …).
Afin de les optimiser, nous avons essayé d'utiliser la fonction GridSearchCV de skicit-learn qui permet d'afficher les meilleurs paramètres du modèle. Nous l'avons laissé tourner pendant plus d'une heure, pour ensuite recevoir un message d'erreur (n_itérations =100 cf. Annexe). Nous avons donc abandonné cette idée qui semblait être pourtant intéressante.

Ensuite, nous avons décidé d'utiliser la Cross-validation afin de pouvoir optimiser le paramètre n_samples dans notre algorithme.
Voici la courbe de validation que nous avons obtenu :



On remarque qu'à partir de n=5 la précision du modèle stagne, ainsi le choix que nous avions fait par tâtonnement(n=10) lors de notre implémentation s'avère cohérent.

Ensuite nous avons dessiné la courbe d'apprentissage du modèle :

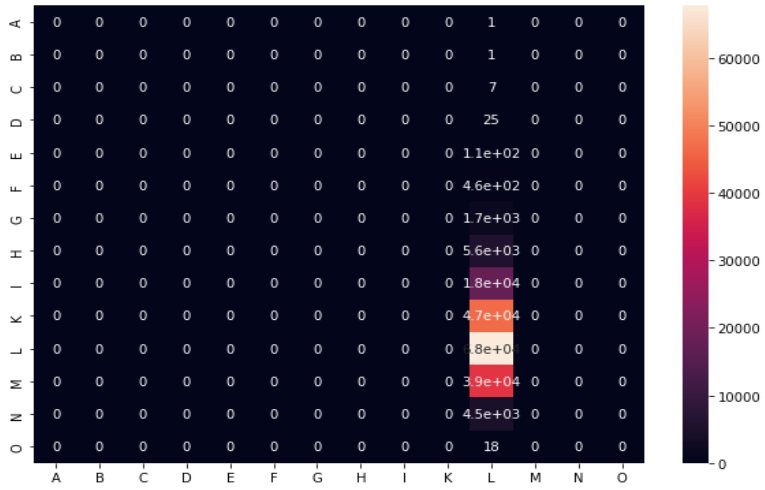


On remarque que plus l'algorithme s'entraîne, plus il est précis dans son apprentissage.
Quant à notre ensemble de tests, la précision au niveau des prédictions va stagner un peu après 400 000 données.
On peut alors en déduire qu'avec cet algorithme à partir de 400 000 données, nos prédictions sur notre ensemble de test varient très peu et qu'à partir de ce seuil il n'est plus nécessaire d'entraîner.
Les courbes semblent bien s'adapter, il y aura un petit espace entre notre courbe d'entraînement et notre courbe de test, ce qui correspond à un espace de généralisation.

Concernant la précision du modèle, nous avons l'accuracy suivante :

```
Out[53]: 0.3687917782567988
```

Nous voyons dans la matrice de confusion que l'algorithme apprend mal et ne prédit que des 11 (étiquette la plus présente dans notre ensemble d'entraînement).
Cela va donc fausser des prédictions, nous voyons alors que notre modèle a à peu près 37% d'accuracy.



b) Arbre de décision

Avec le même découpage de nos données d'entraînements déjà étiquetées (split avec un pourcentage 0.1 pour le test).
On remarque qu'en utilisant le critère par défaut Gini ou le critère Entropy, nous avons la même accuracy :

```
Out[84]: 0.16222483506434124
```

On peut dire que cette accuracy est faible et cela se reflète bien au niveau des scores recueillis sur le site qui dépasse 1 (résultat peu concluant).

On soumet sur le site du challenge pour vérifier notre score :
Your submission score is :
Gini -> Your submission score is : 1.3111971171641954
Entropy -> Your submission score is : 1.3127898621725782

c) Régression linéaire

Avec le même découpage de nos données d'entraînements déjà étiquetées (split avec un pourcentage 0.1 pour le test).
On obtient un score moyen (mse) de :

```
Out[16]: 1.361107386373843
```

On remarque que le mse est assez éloigné de 0 (valeur optimale) on en déduit que la précision n'est pas assez précise.
On soumet sur le site du challenge pour vérifier notre score :
Your submission score is : 0.928567484001525

d) SGD régression

Avec le même découpage de nos données d'entraînements déjà étiquetées (split avec un pourcentage 0.1 pour le test).
On obtient un score moyen (mse) de :

```
Out[146]: 1.4303884020189146
```

On remarque que le mse est assez éloigné de 0 (valeur optimale) on en déduit que la précision n'est pas assez précise.
On soumet sur le site du challenge pour vérifier notre score :
Your submission score is : 0.8794951805241301
On a également modulé les paramètres (learning_rate, eta0, alpha) par tâtonnement en vain.
Nous n'avons aucune amélioration significative à relever.

e) K plus proches voisins

Avec le même découpage de nos données d'entraînements déjà étiquetées (split avec un pourcentage 0.1 pour le test).
Pour la prédiction, il faut 2min pour prédire 1000 valeurs donc à peu près 2,5 jours pour les 1830080 valeurs. On conclut que l'algorithme prend trop de temps à compiler, on ne l'a donc pas utilisé pour générer un rendu sur le site du challenge. Cela semble dommage car la précision semblait satisfaisante par rapport aux autres algorithmes.
Malgré le temps extrêmement long pour prédire, on à chercher à savoir pour quel nombre de voisins l'algorithme donne la meilleure précision pour 1000 valeurs prédites par tâtonnement c'est pour k = 12 qu'on a la meilleure. On a un score de précision de 0,444.

```
accuracy_score(y1_test[:1000], yknn_pred)
```

```
Out[150]: 0.444
```

f) Classification naïve bayésienne

Il y a trois types d'algorithmes de classification naïves bayésienne :Gaussian/Multinomial/Categorial.
Nous les avons entraînés en découpant l'ensemble d'entraînement avec la fonction split (avec un pourcentage de 0.1).
Nous obtenons à peu près des accuracys équivalentes pour chaque type d'algorithme.
Pour Gaussien :

```
Out[153]: 0.36113832821651753
```

Pour Multinomial :

```
Out[38]: 0.3668158164042938
```

Pour Catégorial :

```
Out[65]: 0.3695865177346659
```

Ce modèle est peu contraignant au niveau de la compilation, il est moins complexe que les autres.
Pour chaque chose à ajouter dans notre code nous l'avions utilisé pour tester nos différentes idées.

Par exemple, pour le One-hot-Encoding, cela nous a permis de savoir rapidement que l'on obtenait les mêmes résultats en soumettant sur le site grâce à son temps de compilation rapide.

On a donc soumis les trois fichiers de prédictions pour voir si les scores sont dans le même ordre que les accuracys trouvées précédemment :

nbGauss -> Your submission score is : 0.9999608073682148
nbMulti -> Your submission score is : 0.8589076142574085
nbCateg -> Your submission score is : 0.8589076142574085

On a bien la confirmation que la méthode Naïve Bayésienne multinomiale et catégorielle sont meilleures que la gaussienne.

g) Réseaux de neurones

Avec le même découpage de nos données d'entraînements déjà étiquetées (split avec un pourcentage 0.1 pour le test).
Nous avons entraîné notre algorithme sur notre ensemble d'entraînement en modulant les Hyperparamètres de la fonction d'activation et du solver.

Pour la fonction d'activation 'relu', le solver qui donne le meilleur score de précision est le solveur 'lbfgs'.
Pour la fonction d'activation 'identity', le solver qui donne le meilleur score de précision est le solveur 'sgd'.
Pour la fonction d'activation 'logistic", le solver qui donne le meilleur score de précision est le solveur 'sgd'.

La combinaison qui donne la meilleure précision est la fonction d'activation 'identity' et le solveur 'sgd'.

```
Out[83]: 0.3679181091732532
```

Avec un score sur le site du challenge de :
Your submission score is : 0.8589076142574085
Nous avons également modulé le nombre de couches cachées ainsi que le nombre de neurones (hidden_layers) sans augmentation significative de la précision.
Nous avons le même résultat, pour la modulation du **learning_rate_init**.
Nous avons également tenté de trouver la combinaison des paramètres qui donne les meilleurs scores de précision avec GridSearchCV mais le temps de compilation était trop long pour l'utiliser. (La compilation d'un réseau de neurone met entre 25 et 35 minutes)

5) CONCLUSION

Lors de ce projet, nous avons dû faire face à une contrainte de temps de compilation important pour la majorité des algorithmes. Cela est principalement dû au nombre important de données à disposition.
Nous avons eu beaucoup de réflexions pour avoir des meilleures performances mais celles-ci n'ont pas abouties.
Au niveau des résultats obtenus, on peut dire qu'ils sont peu satisfaisants et nous pouvons trouver plusieurs raisons à cela :
-la valeur de chaque attribut ne suit pas une relation d'ordre (chaque couleur est codée par un chiffre) et le fait d'utiliser le OneHotEncoding n'a pas arrangé les choses au niveau des résultats
-des étiquettes mal réparties au niveau de nos observations dans notre ytrain de départ
-il est dur de trouver les bons paramètres (GridSearchCV qui n'a pas compilé)

Les algorithmes qui nous ont donnés les meilleurs scores sur le site du challenge sont les suivants : MLPClassifier(identity,sgd) / Random Forest/Naive bayse (categorical et multinomial)

Ce projet nous a bien sûr aussi permis de nous rendre compte qu'il est difficile de trouver le meilleur algorithme dans un problème de Machine-learning et nous avons trouvé cet aspect de recherche très intéressant et enrichissant en termes de connaissances mais aussi au niveau des différentes méthodes employées pour tenter de résoudre ce problème.