Rapport de projet Machine Learning

Concours LumenAI

Groupe : **Glycope**

ZERHOUNI Salim

GREFFON Yanis

HENRY Nicolas

Dans ce rapport, nous expliquons l'évolution de notre travail ainsi que les différentes pistes suivies lors de ce projet, qu'elles soient concluantes ou non. Des commentaires expliquant certains détails sont donnés dans le notebook en annexe.

Première approche

Tout d'abord, le dataset proposé paraît dur à étudier en l'état. En effet, sur 1837079 observations, nous constatons que le nombre de coups associé à chaque rubik's cube est loin de suivre une loi uniforme, la majorité des observations se résolvant en 11, 10, 12 et 9 coups et seulement 3 cubes se résolvant en 1 coup.

Aussi, l'étude de ce dataset sans opérer de modifications sur les données est limitée et ne donne pas un résultat qui minimise l'erreur moyenne absolue : appliquer des modèles d'arbres de décision, de classification naïve bayésienne, ou même des réseaux de neurones nous permet uniquement de recevoir en output un vecteur constitué de 1837080 « 11 », n'offrant qu'une erreur moyenne absolue de 0.859.

Aussi, l'obtention d'un tel output nous a fait prendre conscience de l'importance d'une étude et d'une compréhension des données.

Des cubes similaires ?

Nous nous sommes interrogés sur l'existence de "faux" doublons, à savoir des observations différentes mais relatant de la même situation. En effet, le nombre de données est élevé et retirer de potentiels doublons nous aiderait à baisser l'erreur moyenne absolue.

Mais que signifient ces « pos0, …, pos23 » ? Après tout, il semble compliqué de pouvoir comparer différents rubik's cube vu que l'observation d'un cube dépend du point de vue de son observateur. Regarder frontalement et latéralement un rubik's cube devrait donner deux observations différentes, alors qu'il s'agit du même cube.

Pour cela, nous avons tenté de créer un programme (partie #Fonction de rotation du cube) qui permet de changer l'indexation du cube en fonction des mouvements de rotations que l'on fait. Nous avons ensuite testé ces rotations sur quelques combinaisons de notre base de données (le faire pour toute combinaison nous prendrait trop de temps à cause de la complexité) et cherché si ces rotations s'y trouvaient déjà, sans succès. Nous avons travaillé sur cette partie du code avec un autre groupe (Soufiane BOUSTIQUE et Johana LABOU), qui avait les mêmes idées que nous. Ils ont également testé des permutations aléatoires de couleur pendant que nous faisions les rotations de cube. Certaines permutations n'ont aucun

sens lorsque l'on observe le cube, et n'existent pas, mais en testant toutes les permutations possibles sur une combinaison, nous aurions pu trouver de "faux" doublons. Cela a été également un échec.

Cependant, nous avons remarqué que les colonnes 6, 14 et 22 sont constantes et affichent toujours la même couleur quel que soit le cube. Ils sont en fait déjà tous orientés de la même manière, en prenant comme référence le coin de couleur 4,5,6. La recherche de "faux" doublons ne pouvait donc pas être concluante.

Réarrangement des combinaisons par face

En voulant étudier les combinaisons pour avoir d'autres pistes, nous nous sommes demandé si les cubes étaient vraiment indexés face par face. Par exemple : 'pos0', 'pos1', 'pos2', 'pos3' correspondent-ils à la première face ? Pour nous simplifier la tâche, nous avons cherché les cubes de classe 1, car il était plus facile d'étudier des cubes dont un seul coup suffisait à leur résolution. Mais la combinaison [5 1 6 3 5 1 6 3 2 4 4 2 2 4 4 2 1 3 3 1 6 5 5 6] nous a montré que le cube n'était jamais indexé par faces, car avoir 4 couleurs différentes sur la première face n'était pas compatible avec une distance de 1 à la résolution.

De ce fait, nos diverses recherches nous ont permis de comprendre le sens de ces fameux "pos X". Nous avons ainsi procédé à un réarrangement des données. Pour cela, nous avons acheté un rubik's cube 2x2 et trouvé les combinaisons concrètes des cubes à distance 1 de la résolution. Nous nous sommes également inspirés d'un code source trouvé sur internet, d'un individu ayant réussi à décoder l'indexation des générations de problèmes. L'implémentation de ce code est dans notre fonction "faceCube()"

Nos problèmes étaient maintenant indexés par face. Par exemple, les 4 premiers indices composaient la face "front", les 4 suivants la face "back", etc.

Ainsi arrangée, chaque face est représentée par le nombre de couleurs qu'elle contient (nombre allant de 1 à 4, avec "1" représentant une face finie) et par une colonne "somme" représentant la somme des 6 faces, un rubik's cube fini prenant donc la valeur 6.

Cette piste s'est révélée non concluante et les nouveaux attributs inutiles puisque notre prédiction n 'affichait encore que des '11'.

Autre intuition : regroupement des classes, divisées en 3 "lots de classes". En effet, la disparité des observations nous a poussé à effectuer une pondération des colonnes, un regroupement des classes, dans le but d'équilibrer celles-ci. Par exemple, nous avons cherché à créer 3 nouvelles colonnes, regroupant d'une part les observations dont la distance était 12, 13, et 14, d'autre part les observations de distance 10, 9, et 8, et enfin le reste (à savoir 11 associé aux autres, qui sont peu représentatives numériquement). Nous avons aussi essayé d'associer d'une part les distances 12, 10, 8, 6 et d'autre part le reste, pour avoir une quasi-égalité de nombre de valeurs des classes. Ces exemples sont non exhaustifs par rapport à nos tests.

En dépit de cette idée, il s'est avéré que cette classification n'était pas pertinente pour la suite, ne permettant pas aux divers modèles de machine learning d'obtenir un résultat plus intéressant que celui associé à notre précédent vecteur de 11. Nous avons même combiné cette méthode avec la partie "**réarrangement des combinaisons par face**". En effet, nous

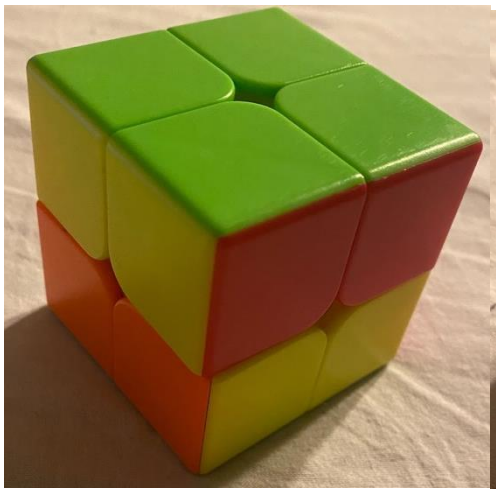
avons essayé des modèles d'arbre de décision, des random forests, des réseaux de neurones, le tout sans optimiser l'erreur moyenne absolue, même en optimisant les hyper-paramètres pour ne pas avoir d'overfitting ou d'underfitting.

Modélisation des données sous la forme de coins. Une autre piste que nous avons explorée est celle des coins. L'avantage du Rubik's Cube 2x2 est d'être composé exclusivement de coins, qui ne changent pas de couleurs mais seulement de positions. Les coins sont composés de trois couleurs chacun, alors on travaille avec des trinômes de données.

Pour arriver à une telle re-modélisation, nous sommes partis d'une base de données déjà réarrangée par face (expliquée dans les parties précédentes). Nous avons tâtonné et intuitivement trouvé par nous-même des positions représentant de façon cohérente nos triplets de coins grâce aux 3 cubes de distance 1 qui sont dans les données train, et à notre rubik's cube 2x2 acheté. La fonction implémentant ce code est "cornerCube()" qui fait elle-même appel à "faceCube()".

Par exemple. Nous prenons les faces dans l'ordre "front", "back", "left", "right", "up", "down". Nous choisissons les couleurs 1 = blanc, 2 = bleu, 3 = orange, 4 = vert, 5 = rouge, 6 = jaune. Les colonnes constantes dont nous avions parlé au préalable (6,14,22) sont sur cette base réarrangée dans les colonnes 5,13,17 et sont de couleurs 5,6,4. Notre coin de référence est donc le coin rouge, jaune, vert et se trouve toujours entre les faces "back", "right", "up". Si nous prenons donc cette combinaison de rubik's cube dans notre base de données réarrangée par faces,

[3 3 1 1 5 5 6 6 1 1 5 5 6 6 3 3 4 4 4 4 2 2 2 2]
alors notre cube ressemblera à cela (à noter que l'on voit le coin de référence dans la première image, mais pour l'observation nous devons le placer sur les faces "back","right","up" comme sur la deuxième image) :




Avant d'appliquer un algorithme, il faut maintenant numériser les valeurs de nos triplets. Nous avions au début considéré qu'un coin "(4,5,2)" était le même coin que "(5,2,4)" car dans un Rubik's Cube, pour trois couleurs données, il n'y a qu'un seul coin qui réunisse ces trois couleurs, et donc qu'il n'y aurait pas de confusion possible. Nous nous sommes donc seulement intéressés au placement des coins. Ainsi nous avons trié nos tuples pour pouvoir

les numériser plus facilement, et écrit notre propre fonction de numérisation “myNumerize()” qui a donné une valeur de 0 à 7 à nos triplets.

De ce fait, la modélisation sous formes de coins nous a permis d’obtenir un score minimisant l’erreur moyenne absolue (à 0,7896).

Nous avons ensuite cherché à éliminer la redondance de nos données en retirant les observations similaires et n’en gardant qu’une à chaque fois. Pourtant, recourir à cette idée nous a fait perdre un nombre important de données (passant d’1,8M à 20 000 données), et a beaucoup augmenté notre erreur moyenne absolue.

Quelle information perdions-nous ? Il s’agissait peut-être de l’orientation des couleurs de chaque coin, qui était auparavant compensée par la compression des valeurs qui rendaient l’apprentissage difficile dans la base de données d’origine.

Nous avons ainsi procédé à une représentation de nos données, incluant 8 nouvelles colonnes de nombres compris entre 0 et 2, proposant 3 orientations différentes, afin d’avoir une représentation plus synthétique de nos données. Pour cela, nous avons dû revoir notre fonction “myNumerize()” et créer une fonction “compressTuple()” qui donnait un nouveau tuple de deux valeurs cette fois-ci en fonction du triplet d’origine. La première valeur du couple correspond à la position du cube, comme pour la modélisation précédente. La seconde correspond à l’une des trois orientations possibles. Par exemple “(1,3,2)” est transformé en “(1,0)” alors que “(3,2,1)” est transformé en “(1,1)” car c’est en fait le même coin. Cependant, “(1,2,5)” s’écrira “(2,0)”. Ces couples étaient donc séparés en 2 colonnes. Cette méthode s’est avérée infructueuse car seul un vecteur composé de distances à 11 était prédit.

Une découverte nous a permis d’avancer : en conservant une représentation des données par les coins, il se trouve que tous les doublons révélés par notre modèle en faisant abstraction des orientations n’ont pas les mêmes distances. En effet, pour une même combinaison donnée, certaines observations peuvent être associées à une distance de 11 tandis que d’autres seront associées à 10. Pour ce faire, notre modèle privilégiera la distance la plus représentée parmi les observations identiques. En effet, sur les 20 000 valeurs après suppression des doublons prenant en compte la distance, il en reste seulement 5 000 si l’on fait abstraction de celle-ci.

Nous avons donc implémenté un programme permettant de mesurer le “poids” (“weight”) d’un des 20 000 cubes restants par rapport à son nombre de doublons dans la base d’origine et à sa distance. Par exemple, si X (distance 11) et X’ (distance 10) sont le même cube dans notre modélisation en coins - ce qui est possible vu que nous faisons abstraction de l’orientation des coins - et que nous retrouvons 900 doublons de X pour seulement 100 doublons de X’, alors X aura un poids de 901 et X’ un poids de 101. Nous trions ensuite notre base de 20 000 données par ordre croissant de poids puis nous supprimons les doublons cette fois-ci en faisant abstraction de la distance (ce qui devrait laisser 5000 valeurs au total sur les 1 800 000), mais tout en gardant le dernier doublon à chaque fois, qui sera toujours de plus fort poids parmi ses semblables, puisque cet attribut est classé par ordre croissant.

En conclusion, nous avons donc pu réduire notre base d'entraînement à 5 000 valeurs, gagnant ainsi en rapidité sur notre complexité de modèle, tout en conservant le même score.

À noter qu’une petite rectification de la prédiction a pu améliorer notre score final, le faisant passer de 0,7896 à 0,7894. Pour cela nous avons simplement cherché si des combinaisons de

la base test se trouvaient dans la base train, avec notre modèle en coins, et le cas échéant, corrigé sa distance dans la prédiction par rapport à celle qui se trouve dans notre base train à 5000 valeurs.

Le fait qu'une immense compression de données ait été faite avec notre modèle en "coin" facilite l'apprentissage, mais cette modélisation a ses limites. En effet, nous avons répété maintes fois qu'il faisait abstraction de l'orientation des coins. Cela constitue quand même une grosse perte de données. Nous aurions pu améliorer notre modèle en trouvant un moyen de représenter les orientations de façon pertinente sans trop augmenter la quantité de données.

Erreur moyenne absolue finale : **0,7894**

Nom dans le classement de l'ENS : **Glycope**

L'annexe est rendue sous forme de fichier pdf (« PojetML.pdf ») avec ce rapport. Ce fichier contient notre notebook code.