Machine Learning – Rapport

Johana LABOU
Soufiane BOUSTIQUE

Sommaire

I. Description du problème et des données

Ce projet est tiré du challenge des données ENS et s'intitule « Solve 2x2x2 Rubik's cube ». Nous disposons d'un jeu de données décrivant un rubik's cube 2x2x2. Le but de ce projet est de prédire le nombre de coups minimal pour résoudre un rubik's cube. On appelle distance ce nombre de coups minimal. Nous invitons le lecteur à se référer à notre code mis en annexe tout au long de sa lecture pour une meilleure compréhension de ce rapport et de nos solutions.

II. Observation des données / Pré-traitement des données

a. Données manquantes

Nous avons commencé par afficher les données. Elles se présentent de la manière suivante (une fois la colonne id supprimée) :

```
Entrée [202]: X
```

Out[202]:

	pos0	pos1	pos2	pos3	pos4	pos5	pos6	pos7	pos8	pos9	...	pos12	pos13	pos15	pos16	pos17	pos18	pos19	pos20	pos21	pos23
0	4	1	2	6	5	2	4	5	1	1	...	4	2	6	1	3	3	5	2	3	3
1	6	5	2	2	4	4	1	6	2	1	...	4	1	6	3	1	5	3	5	3	2
2	5	3	1	3	1	1	6	6	3	2	...	1	6	5	4	4	4	3	2	2	2
3	5	5	1	2	2	4	4	3	4	1	...	2	3	2	1	6	6	3	6	3	5
4	4	2	3	1	3	4	2	3	1	5	...	3	5	4	6	6	2	1	2	1	5
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1837074	2	1	3	5	1	4	3	2	3	3	...	5	1	4	5	4	2	6	6	1	2
1837075	2	3	4	6	3	4	5	5	3	5	...	1	2	3	1	4	6	1	2	6	2
1837076	3	3	4	2	1	6	3	5	3	2	...	4	4	5	2	6	1	5	1	1	2
1837077	5	3	3	5	6	3	4	1	5	1	...	4	1	2	2	6	4	2	2	6	1
1837078	3	1	5	1	6	3	1	3	4	2	...	2	5	6	4	5	1	6	2	2	3

1837079 rows × 21 columns

Le jeu de données comporte 24 attributs (colonnes). Chaque colonne représente une case d'un rubik's cube (4 cases forment une face). Il n'y a aucune donnée manquante. La question que nous nous sommes posés est la suivante : quelles positions (c'est-à-dire colonnes) forment une face ?

b. Réarrangement des colonnes

Pour y répondre, nous sommes partis de l'observation et de la compréhension d'un vrai rubik's cube dont la distance était égale à 1. Nous avons recensé les différentes façons dont pourrait se trouver ce rubik's cube en regardant précisément la position des couleurs.

Nous avons remarqué les choses suivantes :

1. Les rubik's cube qui ont une distance égale à 1 ont forcément 2 faces opposées résolues.
2. Une face non résolue n'a que 2 couleurs : deux cases d'une certaine couleur en haut et deux cases d'une autre couleur en bas, OU deux cases d'une certaine couleur à gauche et deux cases d'une autre couleur à droite, selon s'il faut effectuer une rotation vers le côté ou une rotation vers le haut pour résoudre le rubik's cube.
3. Les faces non résolues alternent systématiquement les couleurs d'une face adjacente à une autre : une couleur en haut se retrouve en bas dans la face suivante, et ainsi de suite.

En affichant tous les rubik's cube de notre jeu de données qui avaient une distance égale à 1 (il y en avait 3), seule une respectait l'observation numéro 1. À partir de là, nous en avons déduit les colonnes qui représentaient une face déjà faite, c'est-à-dire 4 colonnes consécutives de la liste qui avaient le même numéro (un numéro représente une couleur).

En suivant une la logique des points cités pas à pas, nous avons réarrangé les colonnes pour avoir les 4 premières colonnes qui représentent une face, les 4 suivantes une face adjacente et ainsi de suite jusqu'au 8 dernières colonnes qui représentent les faces du haut et du bas.

c. Recherche de redondances

Une fois les colonnes réarrangées, nous avons cherché des redondances dans nos données afin de réduire le nombre de lignes de notre dataframe.

Méthode 1

Nous avons cherché, pour chaque ligne, s'il existait une autre ligne équivalente dont les couleurs seraient inter-changées. Plus précisément, nous avons testé pour chaque ligne de notre jeu de données d'inter-changer chaque couleur par toutes les combinaisons possibles de couleur ; nous avons fixé une liste [1,2,3,4,5,6] qui représentent nos différentes couleurs et nous avons écrit une fonction récursive qui génère toutes les permutations de cette liste (il y a 720 permutations).

Par exemple, une permutation possible est : [1,2,3,4,5,6] → [5,3,1,6,4,2]. Et donc, pour chaque ligne, tous les 1 deviennent des 5, tous les 2 deviennent des 3, et ainsi de suite.

Nous avons cherché si toutes ces lignes générées se trouvent dans notre X_train ou X_test. Mais nous n'en avons trouvé aucune.

Note : Nous sommes conscients que certaines des combinaisons générées ne sont pas compatibles avec une combinaison possible d'un rubik's cube, mais nous l'avons fait pour être exhaustif.

Méthode 2

Nous avons par la suite eu une nouvelle idée pour trouver des redondances. Nous avons pensé, pour chaque ligne, à retourner notre rubik's cube de toutes les façons possibles. Pour ce faire, il faut fixer pour chaque ligne toutes les combinaisons possibles de deux faces opposées (il y a 3 possibilités) qui représentent les faces du haut et du bas selon un axe vertical (d'où l'importance de l'étape précédente). Ensuite, il faut retourner les autres faces selon un axe horizontal au maximum 3 fois car la quatrième fois on revient à notre position de départ. Pour les faces du haut et du bas, les cases pivotent.

Exemple : [face1, face2, face3, face4, [1,2,3,4], [5,4,3,6]]
Avec une seule rotation, ce rubik's cube devient : [face4, face1, face2, face3, [4,1,2,3], [6,5,4,3]]

Finalement, pour chaque ligne, on a 3x6 combinaisons possibles.
Cette méthode ne nous a pas non plus permis de trouver de doublons dans nos données.

Note pour les 2 méthodes : Chercher les redondances dans le X_train prenait beaucoup de temps. Pour aller plus vite, nous avons regroupé les données de notre X_train suivant leur distance (car 2 redondances ont forcément la même distance). Nous avons donc divisé le X_train en 14 dataframes et avons cherché les redondances dans chaque dataframe, ce qui a permis une exécution plus rapide de notre code.

Combinaison méthodes 1 et 2

Nous avons ensuite combiné les méthodes 1 et 2, c'est-à-dire que pour chaque ligne on a 9x720 combinaisons, mais n'avons rien trouvé.

Enfin, nous avons affiché les rubik's cube du X_test dont la distance était égale à 1. Puis, nous avons écrit un code afin d'afficher les rubik's cube du X_train dont la distance est égale à 1, code assez facile à implémenter une fois la logique des rubik's cube comprise. Nous avons trouvé 8 données en tout dont la distance était égale à 1. Nous avons par la suite regardé « à la main » s'il y avait des redondances dans ces données-là, mais cette fois encore, nous n'en avons trouvé aucune, ce qui nous a conforté (non pas confirmé, car il aurait fallu vérifier cela pour toutes les autres distance) dans l'idée qu'il n'y avait définitivement aucune donnée redondante dans notre jeu de données.

En revanche, nous avons remarqué que trois colonnes avaient toujours les mêmes valeurs : nous les avons donc supprimées car elles n'apportaient aucune information supplémentaire.

d. Ajout de features

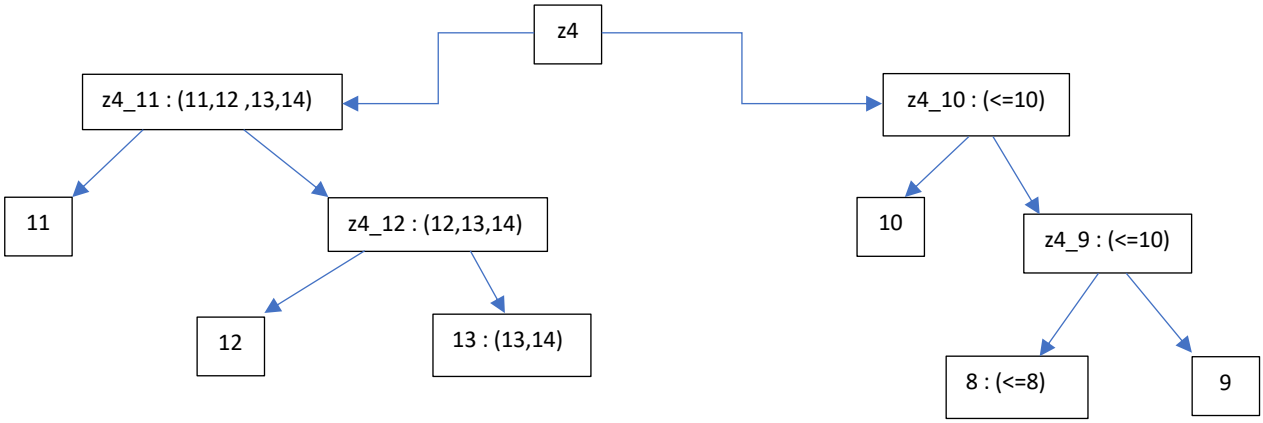
Par la suite, nous avons cherché à trouver des indicateurs qui pouvaient nous aider à améliorer nos modèles. Nous avons pensé à la chose suivante : un rubik's cube ayant une face déjà faite est forcément plus facile à résoudre qu'un rubik's cube complètement mélangé. Nous avons donc ajouté 6 colonnes, une pour chaque face d'un rubik's cube, qui indique si une face est faite (1) ou non (0). Nous avons appelé ces données X_F1.

En suivant la même logique, nous avons également pensé à compter le nombre de couleurs similaires par face : un rubik's cube qui aurait une ou plusieurs faces qui ont 3 cases de la même de la couleur serait peut-être plus facilement solvable qu'un rubik's cube dont les faces sont de couleurs différentes. Pour traduire ceci, nous avons ajouté 6 nouvelles colonnes également, une pour chaque face toujours, qui compte le nombre de couleurs similaires de chaque face. Ces colonnes peuvent prendre les valeurs [0,2,3,4].

e. Traitement de la disparité des classes

La chose la plus remarquable dans nos données est la disproportion des classes. Sur 1 837 079 données, seules 3 ont une distance égale à 1 contre 675 426 qui ont une distance égale à 11. La classe 11 est de loin la classe majoritaire (d'où notre recherche acharnée de redondances pour diminuer le nombre de données, entre autres). Il faut donc faire tout un travail de traitement des classes, autrement nos modèles prédiront systématiquement une distance égale à 11 pour n'importe quel rubik's cube. Ainsi, nous avons réorganisé les classes selon 5 découpages différents (qu'on a appelé z, z1, z2, z3 et z4). Nous avons gardé le 4ème découpage, qu'on appelle z4, qui nous donnait les meilleurs résultats et que nous allons détailler ici. Nous avons réparti toutes nos données dans cette nouvelle variable z4 selon 2 classes (nous avons essayé de répartir les données selon 3 classes mais les résultats étaient moins intéressants) : nous avons mis les données des lignes qui ont une distance égale à 11,12,13 et 14 dans la classe 11, et le reste dans la classe 10.

À partir de là, nous redécoupons chacune des classes 11 et 10 en plusieurs sous-classes (cf. le schéma ci-dessous) :



III. Implémentation des modèles de classification : optimisation des hyper paramètres et analyse des matrices de confusion

Note : pour chaque modèle, nous avons essayé de tracer et d'analyser les courbes d'apprentissage mais cela prenait trop de temps à s'exécuter donc nous ne les avons pas gardées. Idem pour la sélection de features.

1. Données non réduites

a. Random forest

Le premier modèle de classification que nous avons implémenté est le random forest. C'est le modèle qui nous a donné les meilleurs résultats, c'est pourquoi nous le présentons en détails dans cette partie. Nous l'avons d'abord implémenté simplement avant de chercher à optimiser les hyper-paramètres à l'aide d'un GridSearchCV. Le nombre de combinaisons de valeurs à tester étant beaucoup trop important, le GridSearchCV n'arrivait jamais à terme. De même, nous n'avons pas pu utiliser de cross validation score car cela n'arrivait pas à terme.

Nous faisons 2 prédictions pour savoir si notre découpage apporte réellement une plus-value. La première fois, nous faisons une prédiction selon le découpage z4. Ensuite, nous faisons une

prédiction avec nos données d'entrainement « brutes ». Notons que nous séparons nos données de test et nos données d'entraînement en parties égales (50%). Nous avons essayé d'optimiser les hyper-paramètres manuellement pour diminuer l'overfitting mais n'avons pas trouvé de meilleurs résultats. Nous n'avons donc spécifié aucun paramètre, que ce soit n_estimators ou max_depth, qui sont par défaut puisqu'ils nous donnaient de meilleurs résultats ainsi.

Voici ce que nous obtenons :

```
Avec regroupement de classes on obtient les predictions suivantes

11    739129
10    179411
dtype: int64
```

Résultats1

```
Si on les(y_pred_test1) regroupe selon z4 on obtient
11 :  701200
10 :  217340
```

Résultats2

```
La vraie prédiction est la suivante regroupé selon z4
11    555964
10    362576
dtype: int64
```

Résultats3

On remarque clairement en comparant Résultats1 et Résultats2 avec Résultats3 (qui donne la vraie prédiction) que la première prédiction nous donne de moins bons résultats que la seconde, tout simplement parce que le nombre de 11 et de 10 dans la deuxième image est plus proche du vrai nombre de 11 et de 10 (Résultats3). En conclusion, découper nos données n'a pas apporté d'amélioration.

À ce moment-là de notre avancement, nous nous sommes concertés avec un autre groupe (celui de Salim Zerhouni, Yanis Greffon et Nicolas Henry) pour réfléchir à une autre façon de voir les données. Nous avons trouvé une autre solution que nous présentons dans la prochaine partie.

2. Exploration d'une nouvelle approche : réduction des données sans perte d'information

Dans cette partie, nous testons une nouvelle approche en créant un nouveau dataframe qui stocke les informations d'un rubik's cube non pas en termes de faces mais en termes de coins. Un rubik's cube comporte 8 coins, et chaque coin fournit une information sur 3 couleurs. Comme nous avons arrangé correctement les colonnes dans la partie b, il devient facile de retrouver les coins et de les stocker dans des tuples (un tuple sera donc un triplet de couleurs).

Nous avons écrit deux fonctions qui permettent de transformer nos tuples : une première qui permet de conserver l'orientation des coins (c'est une bijection) et une autre fonction qui ne prend pas en compte l'orientation (surjection) : elle permet de transformer nos tuples en entiers en multipliant les entiers représentants les couleurs. Exemple : (1,2,3) devient 1x2x3 = 6. Avec cette deuxième méthode, on perd l'information sur l'orientation car un autre tuple (2,1,3) donnerait également 6.

À partir de ces deux fonctions, nous créons 2 nouveaux dataframes : un premier qui prend en compte l'orientation X_ao (pour « avec orientation »), et un second X_so (« sans orientation ») qui présente des doublons. ATTENTION : il ne faut pas enlever ces doublons car ils vont rajouter du poids à nos y lors de la phase d'apprentissage. Nous aurions pu exprimer ces poids en ajoutant une nouvelle colonne pour supprimer plusieurs lignes, mais nous ne l'avons pas fait ici.

a. Random Forest

On remarque qu'en entraînant nos données avec orientation, on a un overfitting important et un moins bon score au test qu'en entraînant les données sans orientation. Finalement, les meilleures données correspondent aux données sans orientation. Pour le data_test final, on applique les mêmes modifications en regroupant les colonnes par coins et en appliquant ensuite

la fonction qui ne tient pas compte des orientations. En entraînant notre modèle sur le data_test final, on obtient un score de 0,7891, ce qui est le meilleur résultat que nous ayons obtenu.

3. Modèles testés mais non concluants

Dans cette partie, nous listons les autres modèles que nous avons essayés, mais qui n'ont pas fournis de meilleurs résultats que le random forest. Nous expliquons brièvement pourquoi ils n'ont pas été concluants :

- KNN : ne compile pas avec la première approche car trop de données.
- Réseau de neurones (par défaut, 100 neurones) : pourrait être optimisé mais on ne sait pas comment définir une architecture de couches optimale.
- Régression Logistique multi-classe : ne donnait pas de meilleur résultat.
- Descente de gradient stochastique (SGD) : ne donnait pas de meilleur résultat.
- Voting Classifier : il s'agit d'un modèle qui prend en entrée plusieurs modèles et qui renvoie en sortie la classe ayant la plus grande probabilité d'être choisie par l'agrégat de ces différents modèles. Chaque modèle a le même poids. Nous voulions utiliser le voting classifier avec en paramètre les 4 modèles cités précédemment pour que les forces des uns compensent les faiblesses des autres, ce qui aurait potentiellement pu être possible si on avait des matrices de confusion différentes, mais ce n'est pas le cas.
- K-means : on a remarqué que les 14 clusters créés ne correspondaient pas aux 14 clusters définis par la distance.

4. Conclusion

D'autres pistes auraient certainement pu être explorées ; nous aurions pu, par exemple, implémenter des réseaux de neurones à partir d'autres librairies que scikit-learn comme TensorFlow ou Keras par exemple, mais faute de temps et de connaissances de ces librairies, nous n'avons pas pu le faire.

5. Annexes

Machine Learning & Application

2021-2022

Projet Rubik's cube

Le projet porte sur un problème d'apprentissage supervisé. Le problème fait parti des données du challenge des données ENS et s'intitule "Solve 2x2x2 Rubik's cube" et est présenté par la société LumenAI. Une vidéo décrivant le problème se trouve sur le site du collège de France.

Les autres challenges sont aussi très intéressants, mais nécessitent plus de connaissances en machine learning (par exemple de l'apprentissage sur des séries temporelles, sur des images, des sons, du texte, etc...). D'où le choix de ce challenge dont les données sont très proches de problèmes sur lesquels on a travaillé.

La résolution d'un rubik's cube peut être vu comme un problème d'intelligence artificielle (par exemple en utilisant des techniques de recherches avec des heuristiques). On peut même étudier le graphe du jeu du point de vue de la théorie des graphes et découvrir qu'en fait il existe toujours un chemin relativement court à une solution.

ex dans la litérature:

- "The Diameter of the Rubik's Cube Group Is Twenty", *T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge*, SIAM Review, 2014, Vol. 56, No. 4 : pp. 645-670.
- "Solving Rubik's Cube Using Graph Theory", Khemani C., Doshi J., Duseja J., Shah K., Udmale S., Sambhe V. (2019) in: Verma N., Ghosh A. (eds) Computational Intelligence: Theories, Applications and Future Directions - Volume I. Advances in Intelligent Systems and Computing, vol 798. Springer, Singapore.

Ici, on a une base de données qui contient la description de rubik's cubes ainsi que le nombre minimal de coups pour les résoudre. On ne sait pas comment les problèmes ont été générés (est-ce que cela entraine un biais dans les problèmes, est-ce que plusieurs problèmes similaires sont présents dans la base? (Ici par similaire, on pourrait peut être avoir deux problèmes qui apparaissent dans la base, mais en permuttant certaines couleurs, on aurait peut-être exactement le même problème!)). Cependant, on vous demande de constuire un modèle pour nous aider à prédire le nombre de coups minimal pour un problème donné. Ensuite, vous pourriez utiliser ce modèle dans un algorithme de recherche étudié en cours d'IA.

On peut le voir comme un problème de regression où il faut deviner le nombre minimal de coups pour résoudre le rubik's cube, ou comme un problème de classification où la classe d'un état du rubik's cube est le nombre minimal de coups pour le résoudre (donc on pourrait avoir au plus 19 classes). Toutes les méthodes que l'on a vu en cours peuvent s'appliquer.

Les données et le site du challenge

Le projet s'effectue en binôme. Vous devez ouvrir un compte pour le binôme sur le site du challenge, choisissez de participer seul (le binôme sera un seul participant au challenge), puis inscrivez-vous au challenge du cours *M1 MIAGE Dauphine - PSL - 2021-2022*.

Vous aurez accès à trois ensembles:

- `x_train` qui contient la description de 1.837.079 différents rubik's cubes. Chaque rubik's cube est représenté par 25 attributs (lisez la description sur le site du challenge).
- `y_train` qui contient le nombre minimal de coups pour résoudre chacun des 1.837.079 différents rubik's cubes. Ces données sont vos données d'entrainement.
- enfin `x_test` qui contient la description de 1.837.080 nouveaux rubik's cubes. Vous ne connaissez pas le nombre minimal pour chacun de ces problèmes.

Pour participer aux challenges, il vous faudra uploader sur le site votre prédiction sur les rubik's cubes du fichier `x_test` et le site du challenge vous donnera un score. Pour ce score, le site utilise l'erreur moyenne absolue: pour les n=1.837.080 exemples du fichier test, on fait la moyenne entre le vrai nombre minimal de coups y_i et votre prédiction z_i:

$$\frac{\sum_{i=1}^{n} |y_i - z_i|}{n}$$

Malheureusement (pour vous), le site ne vous donnera pas plus d'information que votre score, vous ne pourrez pas savoir quelles sont vos bonnes prédictions et quelles sont vos erreurs. Pire, le site vous permettra d'uploader une prédiction que deux fois par jour!

Soumission et rapport

Un des membres du binôme devra remplir le formulaire Forms de l'équipe Teams du cours (onglet General) pour enregistrer les membres du binôme et le login du binôme qui utilisé sur le site du challenge ENS. Avant le **jeudi 2 décembre à 12h** vous devez 1) avoir créé votre compte pour le binôme et inscrit le binôme sur le challenge du cours, et 2) rempli les informations sur le formulaire Forms.

Vous pouvez faire le projet seul, mais vous serez évalué comme un binôme. Si vous tenez absolument à former un trinôme, contactez moi par email, mais sachez que dès lors, les attentes seront plus élevées.

Le deadline pour le projet sera le **dimanche 20 décembre 23:59** Vous devrez à ce moment là avoir fait trois choses:

- avoir rendu un rapport
- avoir rendu un notebook jupyter ou collab contenant le code pour générer votre solution
- avoir soumis une solution sur le site du challenge ENS.

Le notebook et le rapport seront à soumettre sous myCourse.

Le rapport est un document **pdf** et devra être un document structuré qui explique vos choix, explique votre solution et donne votre résultat. Ne présentez ni le cours, ni le contexte, seul votre travail est important. Le rapport est de **6** pages maximum au format A4 (sans utiliser une taille de police inférieure strictement à 12). Vous pouvez ajouter une annexe à ce rapport (au format pdf ou sous la forme d'un notebook jupyter), étant entendu que le lecteur n'est pas obligé de lire l'annexe. Votre mission est de proposer un modèle de prédiction pour ce problème, votre rapport doit justifier comment vous avez répondu (complètement ou pas) à cette mission (par exemple, vous pouvez décrire ce que vous avez essayé, ce qui a marché ou non, pourquoi vous avez essayé autre chose...). Une autre façon de décrire ce qu'on attend du rapport est la suivante: votre manageur a donné à plusieurs équipes la même tâche d'apprentissage supervisé. Vous devez lui présenter dans ce rapport des arguments qui justifient la qualité de votre approche et de vos résultats. Votre manageur connait le problème, mais n'est pas forcément un expert du domaine. A vous de le convaincre d'utiliser votre solution! (attention, si vous connaissez aussi les limitations de votre solution, il est bon de les exposer aussi!).

L'évaluation portera sur la qualité de votre analyse, même si vos résultats sont peu concluant. Pour caricaturer, un modèle qui gagnerait la compétition sans pouvoir expliquer ce qu'il a fait n'aura pas une bonne note pour le projet du cours (mais bravo, il a gagné la compétition!). Autre caricature, un projet qui applique un seul algorithme et conclue que ça ne fonctionne pas bien n'aura pas non plus une bonne note.

Une soutenance sera organisée lors de la première semaine de cours (après les examens). Elle permettra de compléter l'évaluation et de vous donner un retour sur votre travail. Cette soutenance ne demande aucune préparation de votre part. Elle durera une douzaine/quinzaine de minutes par groupe. La soutenance consistera en un échange au sujet de vos résultats et votre rapport. Si la soutenance fait apparaître qu'un des membres n'a pas beaucoup contribué, sa note pourra être revue à la baisse. Egalement, on pourra vous demander de montrer le code et de fournir les résultats que vous avez obtenu lors des exercices d'implémentation des TDs.

Ce projet compte pour 40% de la note de l'UE. Il est donc souhaitable que la note corresponde au travail de votre groupe, et non aux conseils d'autres groupes, d'autres étudiants ou d'internet. Si vous utilisez des sources (articles de recherche, posts sur internet, etc...), vous devez mentionnez vos sources dans le rapport (sinon, cela s'appelle du plagiat, et cela peut être puni par un conseil de discipline).

Les quelques lignes de code ci-dessous lisent simplement les fichiers sources et affiche la taille des données.

IMPORTATION DES LIBRAIRIES

In [1]:
```
from sklearn.metrics import fbeta_score, make_scorer,accuracy_score
from sklearn.linear_model import SGDClassifier
from sklearn.ensemble import  VotingClassifier
import pandas as pd
import numpy as np
from sklearn.model_selection import GridSearchCV
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
from sklearn.preprocessing import StandardScaler
```

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import confusion_matrix
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import GridSearchCV
import sklearn
from sklearn.feature_selection import VarianceThreshold
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import cross_validate
from sklearn.model_selection import validation_curve
from sklearn.feature_selection import SelectFromModel
from itertools import combinations
from sklearn.metrics import confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib as mpl
import seaborn as sns
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import mean_absolute_error
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.tree import DecisionTreeClassifier
import itertools
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

1. OBSERVATION ET PRE-TRAITEMENT DES DONNÉES

1.1 Observation des données

In [2]:

```
X = pd.read_csv("train_input.csv")
y = pd.read_csv("train_output.csv")
data_test = pd.read_csv("test_input.csv")

X=X.drop(['ID'],axis=1)
y=y.drop(['ID'],axis=1)

data = X
data['distance'] = y['distance']
```

```
X=X.drop(['distance'],axis=1)
data_test=data_test.drop(['ID'],axis=1)
```

In [3]:
```
data_tmp=data.copy()
```

In [4]:
```
data
```

Out[4]:

	pos0	pos1	pos2	pos3	pos4	pos5	pos6	pos7	pos8	pos9	...	pos15	pos16	pos17	pos18	pos19	pos20	pos21	pos22	pos23	distance
0	4	1	1	1	6	2	6	6	5	4	...	4	1	3	3	5	2	3	5	3	11
1	6	5	2	1	2	2	6	3	4	4	...	1	3	1	5	3	5	3	5	2	11
2	5	3	3	2	3	1	6	5	1	1	...	6	4	4	4	3	2	2	5	2	11
3	5	5	4	1	2	1	6	1	2	2	...	4	1	6	6	3	6	3	5	5	9
4	4	2	1	5	1	3	6	6	3	3	...	2	6	6	2	1	2	1	5	5	12
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1837074	2	1	3	3	5	3	6	6	1	5	...	3	5	4	2	6	6	1	5	2	11
1837075	2	3	3	5	6	4	6	1	3	1	...	5	1	4	6	1	2	6	5	2	9
1837076	3	3	3	2	2	4	6	6	1	4	...	3	2	6	1	5	1	1	5	2	12
1837077	5	3	5	1	5	3	6	3	6	4	...	4	2	6	4	2	2	6	5	1	11
1837078	3	1	4	2	1	5	6	4	6	2	...	1	4	5	1	6	2	2	5	3	11

1837079 rows × 25 columns

In [5]:
```
data_test
```

Out[5]:

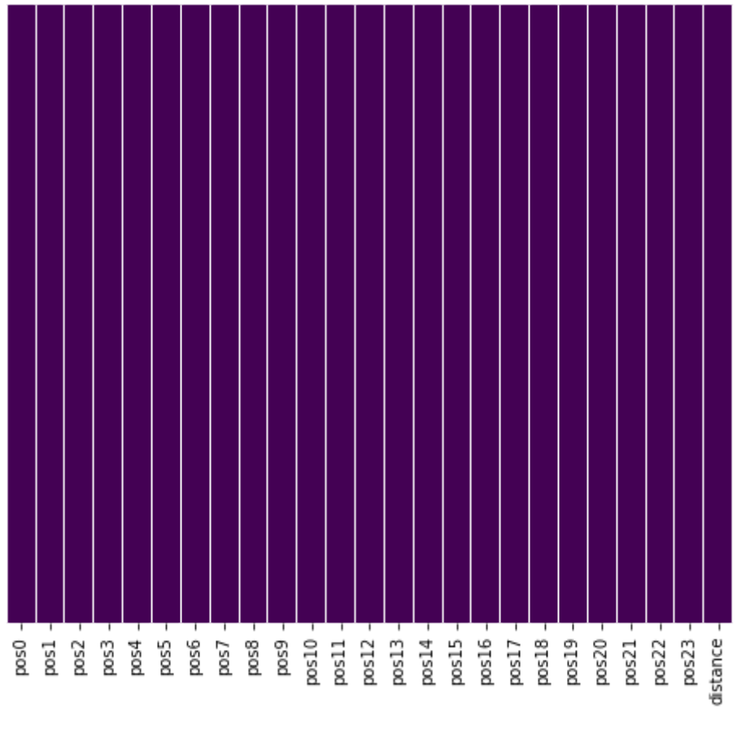
	pos0	pos1	pos2	pos3	pos4	pos5	pos6	pos7	pos8	pos9	...	pos14	pos15	pos16	pos17	pos18	pos19	pos20	pos21	pos22	pos23
0	2	3	6	1	1	4	6	5	6	1	...	4	1	3	4	5	5	2	6	5	4
1	3	1	1	5	1	2	6	3	2	4	...	4	6	6	3	3	1	5	6	5	4
2	2	4	6	1	3	1	6	5	1	6	...	4	1	5	3	5	3	2	3	5	4
3	3	4	2	2	5	1	6	5	4	6	...	4	6	1	3	3	3	1	2	5	2
4	6	4	1	6	6	2	6	3	2	1	...	4	4	5	5	2	4	3	1	5	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1837075	6	5	1	5	3	2	6	6	2	4	...	4	4	5	1	3	2	4	3	5	3
1837076	6	1	1	5	4	6	6	3	3	2	...	4	6	2	5	3	1	1	5	5	4
1837077	1	6	3	5	2	6	6	2	3	5	...	4	3	4	2	6	1	1	3	5	1
1837078	1	2	4	1	5	5	6	4	5	3	...	4	1	2	6	6	2	6	4	5	3
1837079	4	5	1	2	5	6	6	3	1	4	...	4	6	3	1	2	3	6	2	5	4

1837080 rows × 24 columns

1.2 Données manquantes

In [6]:
```
# On constate qu'il n'y a pas de données manquantes
plt.figure(figsize=(8, 7))
sns.heatmap(data.isnull(), cbar = False , yticklabels = False , cmap = 'viridis',)
```

Out[6]: <AxesSubplot:>



In [7]:
```
data.isnull().sum()
```

Out[7]:
```
pos0        0
pos1        0
pos2        0
pos3        0
pos4        0
pos5        0
pos6        0
pos7        0
pos8        0
pos9        0
pos10       0
pos11       0
pos12       0
pos13       0
pos14       0
pos15       0
pos16       0
pos17       0
pos18       0
pos19       0
pos20       0
pos21       0
pos22       0
pos23       0
distance    0
dtype: int64
```

1.3 Ré-arrangement des colonnes

In [8]:
```
#Ré-arrangement grace à l'observation des distances égale à 1
#Liste des distances égales à 1 avant changement
data[data['distance']==1].values[:,:24].tolist()
```

Out[8]:
```
[[5, 1, 6, 3, 5, 1, 6, 3, 2, 4, 4, 2, 2, 4, 4, 2, 1, 3, 3, 1, 6, 5, 5, 6],
 [1, 1, 6, 6, 1, 1, 6, 6, 3, 5, 4, 2, 3, 5, 4, 2, 4, 4, 3, 3, 2, 2, 5, 5],
 [2, 2, 4, 4, 1, 1, 6, 6, 6, 1, 1, 6, 2, 4, 4, 2, 3, 3, 3, 3, 5, 5, 5, 5]]
```

In [9]:
```
#####CHANGEMENT DATA

#ETAPE 0
F1_3=data['pos4'].copy()
F1_4=data['pos5'].copy()
F2_1=data['pos2'].copy()
F2_2=data['pos3'].copy()
data['pos2']=F1_3
data['pos3']=F1_4
data['pos4']=F2_1
data['pos5']=F2_2
F1_2=data['pos11'].copy()
F1_3=data['pos12'].copy()
F1_4=data['pos15'].copy()
F2_1=data['pos9'].copy()
F2_2=data['pos10'].copy()
F2_3=data['pos13'].copy()
F2_4=data['pos14'].copy()
data['pos9']=F1_2
data['pos10']=F1_3
data['pos11']=F1_4
data['pos12']=F2_1
data['pos13']=F2_2
data['pos14']=F2_3
data['pos15']=F2_4
#ETAPE 1
tmp1=data['pos10'].copy()
tmp2=data['pos11'].copy()
tmp3=data['pos14'].copy()
tmp4=data['pos15'].copy()
data['pos10']=tmp2
data['pos11']=tmp1
data['pos14']=tmp4
data['pos15']=tmp3
tmp1=data['pos2'].copy()
tmp2=data['pos3'].copy()
tmp3=data['pos6'].copy()
tmp4=data['pos7'].copy()
data['pos2']=tmp2
data['pos3']=tmp1
data['pos6']=tmp4
data['pos7']=tmp3
#ETAPE 2
tmp1=data[['pos4','pos5','pos6','pos7']].copy()
tmp2=data[['pos8','pos9','pos10','pos11']].copy()

data[['pos4','pos5','pos6','pos7']]=tmp2
data[['pos8','pos9','pos10','pos11']]=tmp1


#####CHANGEMENT DATA TEST

#ETAPE 0
F1_3=data_test['pos4'].copy()
```

```
F1_4=data_test['pos5'].copy()
F2_1=data_test['pos2'].copy()
F2_2=data_test['pos3'].copy()
data_test['pos2']=F1_3
data_test['pos3']=F1_4
data_test['pos4']=F2_1
data_test['pos5']=F2_2
F1_2=data_test['pos11'].copy()
F1_3=data_test['pos12'].copy()
F1_4=data_test['pos15'].copy()
F2_1=data_test['pos9'].copy()
F2_2=data_test['pos10'].copy()
F2_3=data_test['pos13'].copy()
F2_4=data_test['pos14'].copy()
data_test['pos9']=F1_2
data_test['pos10']=F1_3
data_test['pos11']=F1_4
data_test['pos12']=F2_1
data_test['pos13']=F2_2
data_test['pos14']=F2_3
data_test['pos15']=F2_4
#ETAPE 1
tmp1=data_test['pos10'].copy()
tmp2=data_test['pos11'].copy()
tmp3=data_test['pos14'].copy()
tmp4=data_test['pos15'].copy()
data_test['pos10']=tmp2
data_test['pos11']=tmp1
data_test['pos14']=tmp4
data_test['pos15']=tmp3
tmp1=data_test['pos2'].copy()
tmp2=data_test['pos3'].copy()
tmp3=data_test['pos6'].copy()
tmp4=data_test['pos7'].copy()
data_test['pos2']=tmp2
data_test['pos3']=tmp1
data_test['pos6']=tmp4
data_test['pos7']=tmp3
#ETAPE 2
tmp1=data_test[['pos4','pos5','pos6','pos7']].copy()
tmp2=data_test[['pos8','pos9','pos10','pos11']].copy()
data_test[['pos4','pos5','pos6','pos7']]=tmp2
data_test[['pos8','pos9','pos10','pos11']]=tmp1
```

In [10]:

```
#Liste des distances égales à 1 après changement
data[data['distance']==1].values[:,:24].tolist()
```

Out[10]:

```
[[5, 1, 1, 5, 2, 2, 2, 2, 6, 3, 3, 6, 4, 4, 4, 4, 1, 3, 3, 1, 6, 5, 5, 6],
 [1, 1, 1, 1, 3, 2, 2, 3, 6, 6, 6, 6, 5, 4, 4, 5, 4, 4, 3, 3, 2, 2, 5, 5],
 [2, 2, 1, 1, 6, 6, 2, 2, 4, 4, 6, 6, 1, 1, 4, 4, 3, 3, 3, 3, 5, 5, 5, 5]]
```

1.4 Recherche de redondances

In [11]:

```
# On remarque qu'il n'y a pas deux fois la meme valeur dans la dataFramepo
data.duplicated().sum()
```

Out[11]:

```
0
```

In [12]:

```
#                                  -------   Méthode 1  -------
```

```python
# On génère des dataFrame regroupés par distance
# data_i[i] nous donne les lignes de notre dataFrame avec distance==i
data_i=[[]]
for i in range(1,15):
    tmp=data[data['distance']==i]
    data_i.append(tmp)

# Exemple:
print('data_i avec distance =1 :')
print(data_i[1].values)
print('\n\n')

# On génere permutation de [1,2,3,4,5,6]
def permutations(start,L,end=[]):
    if len(start) == 0:
        L.append(end)
    else:
            for i in range(len(start)):
                permutations(start[:i] + start[i+1:],L ,end + start[i:i+1])

L=[]
permutations([1,2,3,4,5,6],L)
L.pop(0)

# Code qui renvoie pour chaque ligne du dataFrame toutes ses permutations de couleurs

def permut_ligne(X,L):
    faces_similaire=[]
    for l in L:
        tmp=[]
        for x in X:
            if x ==1 :
                tmp.append(l[0])
            if x ==2 :
                tmp.append(l[1])
            if x ==3 :
                tmp.append(l[2])
            if x ==4 :
                tmp.append(l[3])
            if x ==5 :
                tmp.append(l[4])
            if x ==6 :
                tmp.append(l[5])
        faces_similaire.append(tmp)
    return faces_similaire



def existe_permut(data_i):
    R=[]
    i=0
    X_i_list=data_i.iloc[:,:24].values.tolist()
    for Xi in X_i_list :
        if(list(Xi) in R):
            continue
        P=permut_ligne(Xi,L)
        for p in P :
            if (p in X_i_list):
                R.append(p)
                print('ici : ',p,'origine : ',Xi)
        if(len(R)==0):
            print('existe pas pour ligne ',i,', pour data avec distance = ',data_i['distance'].iloc[0])
```

```
        i=i+1


# on teste l'existence des permutations pour data_i[i] avec i allant de 1 à 14. Ici on prend l'exemple i=1 :
existe_permut(data_i[1])
```

```
data_i avec distance =1 :
[[5 1 1 5 2 2 2 2 6 3 3 6 4 4 4 4 1 3 3 1 6 5 5 6 1]
 [1 1 1 1 3 2 2 3 6 6 6 6 5 4 4 5 4 4 3 3 2 2 5 5 1]
 [2 2 1 1 6 6 2 2 4 4 6 6 1 1 4 4 3 3 3 3 5 5 5 5 1]]



existe pas pour ligne  0 , pour data avec distance =  1
existe pas pour ligne  1 , pour data avec distance =  1
existe pas pour ligne  2 , pour data avec distance =  1
```

In [13]:

```
#                                        -------   Méthode 2  -------

#Ici on genere toutes les facons possibles de retourner un rubiks cube (pour chaque ligne)

#on a à la fin 3*15 = 3*(2 parmi 6)=45 combinaison pour chaque ligne
#(2 parmi 6) pour savoir quelle est la face du haut et la face du bas
# Nb : lorsqu'on a écrit ce code, on ne savait pas encore distinguer les faces opposées,
#      donc au lieu de (2 parmi 6)=15, on a 6 possibilités donc en tout pour chaque ligne on a: 3*6=18 possibilités

# i fait reference au numéro de la ligne dans X
def Permut_changement_pv(i,X):
    combi_2_6=list(combinations([0,1,2,3,4,5], 2))
    Face = []
    Face.append(X.values[i].tolist()[0:4])
    Face.append(X.values[i].tolist()[4:8])
    Face.append(X.values[i].tolist()[8:12])
    Face.append(X.values[i].tolist()[12:16])
    Face.append(X.values[i].tolist()[16:20])
    Face.append(X.values[i].tolist()[20:24])
#############################################################################
    L1=[]
    for i in range(1,4):
        for c in combi_2_6:
            F_tmp=Face.copy()
            F_tmp[c[0]]=list(np.roll(F_tmp[c[0]],i))## on tourne verticalement la face du haut
            F_tmp[c[1]]=list(np.roll(F_tmp[c[1]],i))## on tourne verticalement la face du bas
            ## On récupère le reste des faces dans une liste
            F_reste=[]## les faces à tourner en meme temps horizentalement; il y en a 4
            for e in range(0,6):
                if(e!=c[0] and e!=c[1]):
                    F_reste.append(F_tmp[e])
            ## on applatit F_reste
            F_reste=list(itertools.chain(*F_reste))
            ## on la tourne horizentalement 1 fois
            F_reste=list(np.roll(F_reste,4*i)) ## F_reste va de 0 à 15
            ## on les remets dans leur places
            j=0
            for n in range(0,6):
                if(n!=c[0] and n!=c[1]):
                    F_tmp[n]=F_reste[j:j+4]
                    j=j+4
            #print(F_tmp)
            F_tmp=list(itertools.chain(*F_tmp))
            L1.append(F_tmp)
    return L1
```

```
# i fait référence au numéro de la ligne dans data_i et j la distance
def existe_Permut_changement_pv(j,i,data_i):
    L=Permut_changement_pv(i,data_i[j].iloc[:,:24])
    cmpt=0
    for l in L:
        if(l in data_i[j].values.tolist()):
            print('yes, voici la redondance: ',l)
            cmpt=cmpt+1
    if(cmpt==0):
        print('non trouvé pour ligne ',i, 'dans data avec distance =',j)



#exemple on teste l'existence de toutes les permutations dans la dataFrame avec distance = 1, dans la ligne 0
# pour chercher dans tout la data il faut j allant de 1 à 14 et i parcourant data_i[i]
existe_Permut_changement_pv(1,0,data_i)
```

```
non trouvé pour ligne  0 dans data avec distance = 1
```

In [14]:

```
#                                        -------   Méthode 1 et 2 combinées  -------
# Il suffit juste de combiner existe_Permut_changement_pv et existe_permut mais on n'a pas trouvé non plus
```

In [15]:

```
# Detecte toutes les lignes dans data test avec distance = 1

D_1=[]
i=0
for d in data_test.values.tolist():
    cmpt=0
    tmp=[]
    for i in range(0,21,4):
        if(d[i]==d[i+1]==d[i+2]==d[i+3]):
            cmpt=cmpt+1
            tmp.append(i)
    if(cmpt==2):
        cmpt1=0
        for j in range(0,21,4):
            if(j==tmp[0] or j==tmp[1]):
                continue
            if((d[j]==d[j+1] and d[j+2]==d[j+3] ) or (d[j]==d[j+3] and d[j+1]==d[j+2] )):
                cmpt1=cmpt1+1
        #print(cmpt1)
        if(cmpt1==4):
            D_1.append(d)
    i=i+1

D_1
```

Out[15]:

```
[[3, 1, 1, 3, 2, 2, 2, 2, 6, 5, 5, 6, 4, 4, 4, 4, 6, 3, 3, 6, 1, 5, 5, 1],
 [1, 1, 1, 1, 5, 2, 2, 5, 6, 6, 6, 6, 3, 4, 4, 3, 2, 2, 3, 3, 4, 4, 5, 5],
 [6, 6, 1, 1, 4, 4, 2, 2, 1, 1, 6, 6, 2, 2, 4, 4, 3, 3, 3, 3, 5, 5, 5, 5],
 [4, 4, 1, 1, 1, 1, 2, 2, 2, 2, 6, 6, 6, 6, 4, 4, 3, 3, 3, 3, 5, 5, 5, 5],
 [1, 1, 1, 1, 4, 2, 2, 4, 6, 6, 6, 6, 2, 4, 4, 2, 5, 5, 3, 3, 3, 3, 5, 5]]
```

1.5 Ajout de features

In [16]:

```
def F1(L):
    if L[0]==L[1]==L[2]==L[3] :
        return 1
    return 0
```

```
def F2(L):
    cmpt=[0,0,0,0,0,0]
    for l in L:
        for i in range(1,7):
            if(l==i):
                cmpt[i-1]=cmpt[i-1]+1
    maxi = max(cmpt)
    if maxi==1:
        return 0
    return maxi

########################
Face_F1 = [[],[],[],[],[],[]]
Face_F2 = [[],[],[],[],[],[]]
for x in np.array(X):
    Face_F1[0].append(F1(x[0:4]))
    Face_F1[1].append(F1(x[4:8]))
    Face_F1[2].append(F1(x[8:12]))
    Face_F1[3].append(F1(x[12:16]))
    Face_F1[4].append(F1(x[16:20]))
    Face_F1[5].append(F1(x[20:24]))
    Face_F2[0].append(F2(x[0:4]))
    Face_F2[1].append(F2(x[4:8]))
    Face_F2[2].append(F2(x[8:12]))
    Face_F2[3].append(F2(x[12:16]))
    Face_F2[4].append(F2(x[16:20]))
    Face_F2[5].append(F2(x[20:24]))

X_F1=X.copy()
X_F2=X.copy()

X_F1['Face1']=Face_F1[0]
X_F1['Face2']=Face_F1[1]
X_F1['Face3']=Face_F1[2]
X_F1['Face4']=Face_F1[3]
X_F1['Face5']=Face_F1[4]
X_F1['Face6']=Face_F1[5]

###############################################

X_F2['Face1']=Face_F2[0]
X_F2['Face2']=Face_F2[1]
X_F2['Face3']=Face_F2[2]
X_F2['Face4']=Face_F2[3]
X_F2['Face5']=Face_F2[4]
X_F2['Face6']=Face_F2[5]

###############################################
X_F2_faces=X_F2[['Face1','Face2','Face3','Face4','Face5','Face6']]
```

In [17]:

```
#On ajoute 6 colonnes pour indiquer si une face est faite: 1, sinon :0
X_F1
```

Out[17]:

	pos0	pos1	pos2	pos3	pos4	pos5	pos6	pos7	pos8	pos9	...	pos20	pos21	pos22	pos23	Face1	Face2	Face3	Face4	Face5	Face6
0	4	1	1	1	6	2	6	6	5	4	...	2	3	5	3	0	0	0	0	0	0
1	6	5	2	1	2	2	6	3	4	4	...	5	3	5	2	0	0	0	0	0	0
2	5	3	3	2	3	1	6	5	1	1	...	2	2	5	2	0	0	0	0	0	0
3	5	5	4	1	2	1	6	1	2	2	...	6	3	5	5	0	0	0	0	0	0

	pos0	pos1	pos2	pos3	pos4	pos5	pos6	pos7	pos8	pos9	...	pos20	pos21	pos22	pos23	Face1	Face2	Face3	Face4	Face5	Face6
4	4	2	1	5	1	3	6	6	3	3	...	2	1	5	5	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1837074	2	1	3	3	5	3	6	6	1	5	...	6	1	5	2	0	0	0	0	0	0
1837075	2	3	3	5	6	4	6	1	3	1	...	2	6	5	2	0	0	0	0	0	0
1837076	3	3	3	2	2	4	6	6	1	4	...	1	1	5	2	0	0	0	0	0	0
1837077	5	3	5	1	5	3	6	3	6	4	...	2	6	5	1	0	0	0	0	0	0
1837078	3	1	4	2	1	5	6	4	6	2	...	2	2	5	3	0	0	0	0	0	0

1837079 rows × 30 columns

In [18]:
```
#On ajoute 6 colonnes pour indiquer le nombre de cases similaires sur une face
X_F2
```

Out[18]:

	pos0	pos1	pos2	pos3	pos4	pos5	pos6	pos7	pos8	pos9	...	pos20	pos21	pos22	pos23	Face1	Face2	Face3	Face4	Face5	Face6
0	4	1	1	1	6	2	6	6	5	4	...	2	3	5	3	3	3	2	2	2	2
1	6	5	2	1	2	2	6	3	4	4	...	5	3	5	2	0	2	3	2	2	2
2	5	3	3	2	3	1	6	5	1	1	...	2	2	5	2	2	0	3	2	3	3
3	5	5	4	1	2	1	6	1	2	2	...	6	3	5	5	2	2	2	2	2	2
4	4	2	1	5	1	3	6	6	3	3	...	2	1	5	5	0	2	2	2	2	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1837074	2	1	3	3	5	3	6	6	1	5	...	6	1	5	2	2	2	2	2	0	0
1837075	2	3	3	5	6	4	6	1	3	1	...	2	6	5	2	2	2	0	2	2	2
1837076	3	3	3	2	2	4	6	6	1	4	...	1	1	5	2	3	2	2	2	0	2
1837077	5	3	5	1	5	3	6	3	6	4	...	2	6	5	1	2	2	0	2	2	0
1837078	3	1	4	2	1	5	6	4	6	2	...	2	2	5	3	0	0	0	0	0	2

1837079 rows × 30 columns

1.6 Traitement de la disparité des classes

In [19]:
```
#On remarque que la colonne 11 , 14 et 22 ont toujous la meme valeurs, donc on les supprime
print(data['pos11'].duplicated().sum())
print(data['pos14'].duplicated().sum())
print(data['pos22'].duplicated().sum())
```

```
1837078
1837078
1837078
```

In [20]:
```
X_drop=pd.DataFrame()
X_drop=X.drop(['pos11','pos14','pos22'],axis=1)

data_test_drop=pd.DataFrame()
data_test_drop=data_test.drop(['pos11','pos14','pos22'],axis=1)
```

```
data_drop=pd.DataFrame()
data_drop=data.drop(['pos11','pos14','pos22'],axis=1)


#X_F2_drop=pd.DataFrame()
#X_F2_drop=X_F2.drop(['pos11','pos14','pos22'],axis=1)
```

In [21]:

```
# On remarque la disparité des classes
y.value_counts()
```

Out[21]:

```
distance
11          675426
10          465294
12          391268
9           180254
8            57074
13           45140
7            16529
6             4485
5             1128
4              267
14             138
3               60
2               13
1                3
dtype: int64
```

In [22]:

```
y1=y.values
```

Chaque z constitue un un facon de regrouper nos classes

In [23]:

```
"""
z=[]
for i in range(np.size(y1)):
    if y1[i]==10 or y1[i]==12 or y1[i]==8 or y1[i]==6:
        z.append(10)
    else :
        z.append(11)
z=np.array(z)
z=pd.DataFrame(z,columns=['distance'])

"""
```

Out[23]:

```
"\nz=[]\nfor i in range(np.size(y1)):\n    if y1[i]==10 or y1[i]==12 or y1[i]==8 or y1[i]==6:\n        z.append(10)\n    else :\n        z.append(11)\nz=np.array(z)\nz=pd.DataFram
e(z,columns=['distance'])    \n\n"
```

In [24]:

```
#z.value_counts()
```

In [25]:

```
"""
z1=[]
for i in range(np.size(y1)):
    if y1[i]==10:
        z1.append(10)
    elif y1[i]==11:
        z1.append(11)
    else :
        z1.append(12)
z1=np.array(z1)
z1=pd.DataFrame(z1,columns=['distance'])
"""
```

Out[25]:
```
"\nz1=[]\nfor i in range(np.size(y1)):\n    if y1[i]==10:\n        z1.append(10)\n    elif y1[i]==11:\n        z1.append(11)    \n    else :\n        z1.append(12)\nz1=np.array(z1)\nz1=pd.DataFrame(z1,columns=['distance'])\n"
```

In [26]:
```
#z1.value_counts()
```

In [27]:
```
"""
z2=[]
for i in range(np.size(y1)):
    if y1[i]==11:
        z2.append(11)
    elif y1[i]==12 or y1[i]==9:
        z2.append(12)
    else :
        z2.append(10)
z2=np.array(z2)
z2=pd.DataFrame(z2,columns=['distance'])
"""
```

Out[27]:
```
"\nz2=[]\nfor i in range(np.size(y1)):\n    if y1[i]==11:\n        z2.append(11)\n    elif y1[i]==12 or y1[i]==9:\n        z2.append(12)    \n    else :\n        z2.append(10)\nz2
=np.array(z2)\nz2=pd.DataFrame(z2,columns=['distance']) \n"
```

In [28]:
```
#z2.value_counts()
```

In [29]:
```
"""
z3=[]
for i in range(np.size(y1)):
    if y1[i]==11 or y1[i]==12:
        z3.append(11)
    else :
        z3.append(10)
z3=np.array(z3)
z3=pd.DataFrame(z3,columns=['distance'])
"""
```

Out[29]:
```
"\nz3=[]\nfor i in range(np.size(y1)):\n    if y1[i]==11 or y1[i]==12:\n        z3.append(11)    \n    else :\n        z3.append(10)\nz3=np.array(z3)\nz3=pd.DataFrame(z3,columns=
['distance']) \n"
```

In [30]:
```
#z3.value_counts()
```

In [31]:
```
z4=[]
for i in range(np.size(y1)):
    if y1[i]==11 or y1[i]==12 or y1[i]==13 or y1[i]==14:
        z4.append(11)
    else :
        z4.append(10)
z4=np.array(z4)
z4=pd.DataFrame(z4,columns=['distance'])
```

In [32]:
```
z4.value_counts()
```

Out[32]:
```
distance
11          1111972
10           725107
dtype: int64
```

Dans cette partie nous avons choisi z4 comme 1ere découpe car elle donne les meilleurs résultats. Ensuite il faut continuer les découpage par chacune des sous-classes et ré-entrainer notre modèle pour différencier les sous-classes.

In [33]:
```
z4_11=[]
z4_10=[]
for i in range(np.size(y1)):
    if y1[i]==11:
        z4_11.append(11)
    elif y1[i]==12 or y1[i]==13 or y1[i]==14:
        z4_11.append(12)
    elif y1[i]==10:
        z4_10.append(10)
    else:
        z4_10.append(9)

z4_11=np.array(z4_11)
z4_11=pd.DataFrame(z4_11,columns=['distance'])
z4_10=np.array(z4_10)
z4_10=pd.DataFrame(z4_10,columns=['distance'])
```

In [34]:
```
z4_9=[]
z4_12=[]
for i in range(np.size(y1)):
    if(y1[i]==9):
        z4_9.append(9)
    elif(y1[i]<9):
        z4_9.append(8)
    if(y1[i]==12):
        z4_12.append(12)
    elif(y1[i]>12):
        z4_12.append(13)

z4_9=np.array(z4_9)
z4_9=pd.DataFrame(z4_9,columns=['distance'])
z4_12=np.array(z4_12)
z4_12=pd.DataFrame(z4_12,columns=['distance'])
```

In [35]:
```
X_11=X[y['distance']>=11]
z4_11.value_counts()
```

Out[35]:
```
distance
11          675426
12          436546
dtype: int64
```

In [36]:
```
X_12=X[y['distance']>=12]
z4_12.value_counts()
```

Out[36]:
```
distance
12          391268
13           45278
dtype: int64
```

In [37]:
```
X_10=X[y['distance']<=10]
z4_10.value_counts()
```

Out[37]:
```
distance
10          465294
```

```
9           259813
dtype: int64
```

In [38]:
```
X_9=X[y['distance']<=9]
z4_9.value_counts()
```

Out[38]:
```
distance
9           180254
8            79559
dtype: int64
```

2. IMPLEMENTATION DES MODELES DE CLASSIFICATION

2.1 Données non réduites

a. Découpage

In [39]:
```
#Découpage z4
X_train, X_test, y_train, y_test = train_test_split(X.values, z4['distance'].values, test_size=0.5,shuffle=True,random_state=1)

#Découpage data set initial
X_train1, X_test1, y_train1, y_test1 = train_test_split(X.values, y.values, test_size=0.5,shuffle=True,random_state=1)
```

b. Random Forest

In [40]:
```
regressor = RandomForestClassifier(n_jobs=32,random_state=6)
regressor1 = RandomForestClassifier(n_jobs=32,random_state=6)
```

In [41]:
```
regressor.fit(X_train,y_train)
regressor1.fit(X_train1,y_train1)
```

Out[41]:
```
RandomForestClassifier(n_jobs=32, random_state=6)
```

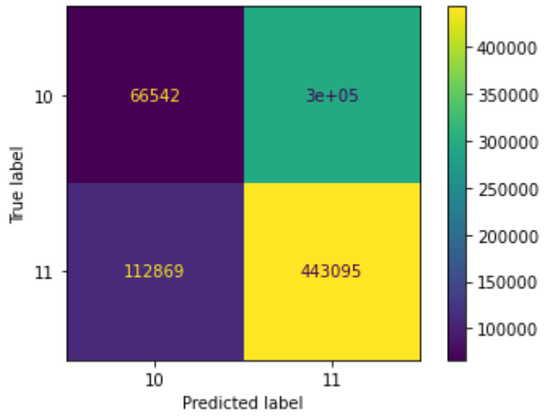
In [42]:
```
y_pred_test=regressor.predict(X_test)
y_pred_test1=regressor1.predict(X_test1)
```

In [43]:
```
print('n:',mean_absolute_error(y_pred_test,y_test))
print('1:',mean_absolute_error(y_pred_test1,y_test1))
```

```
n: 0.4451662420798223
1: 1.0586136695190194
```

In [44]:
```
plot_confusion_matrix(regressor,X_test,y_test)
```

Out[44]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7fef7c82aee0>
```



In [45]:
```
# Sans regroupement de classes on obtient les predictions suivantes ( suffit juste de sommer values(11)+values(12)
# + values(13)+ values(14) )
print(pd.DataFrame(y_pred_test1).value_counts(),'\n')
tmp=pd.DataFrame(y_pred_test1).value_counts().tolist()
```

```
11    596634
10    203905
12    104180
9      12121
8       1167
13       384
7        115
6         28
14         2
4          2
5          1
3          1
dtype: int64
```

In [46]:
```
print("Si on les(y_pred_test1) regroupe selon z4 on obtient")
print('11 : ',tmp[0]+tmp[2]+tmp[5]+tmp[9])
print('10 : ',sum(tmp)-(tmp[0]+tmp[2]+tmp[5]+tmp[9]))
```

```
Si on les(y_pred_test1) regroupe selon z4 on obtient
11 :  701200
10 :  217340
```

In [47]:
```
print("Avec regroupement de classes on obtient les predictions suivantes")
pd.DataFrame(y_pred_test).value_counts()
```

```
Avec regroupement de classes on obtient les predictions suivantes
```

Out[47]:
```
11    739129
10    179411
dtype: int64
```

In [48]:
```
print("\nLa vraie prédiction est la suivante regroupé selon z4")
print(pd.DataFrame(y_test).value_counts())
```

```
La vraie prédiction est la suivante regroupé selon z4
11    555964
10    362576
dtype: int64
```

2.2 Exploration d'une nouvelle approche : réduction des données sans perte d'information

Dans cette partie, on va regrouper notre rubik's cube selon un coin fixé.

In [49]:
```
tmp = data_tmp.copy()
data_c = data_tmp.copy()
data_c['pos0'] = tmp['pos18']
data_c['pos1'] = tmp['pos17']
data_c['pos2'] = tmp['pos19']
data_c['pos3'] = tmp['pos16']
data_c['pos4'] = tmp['pos21']
data_c['pos5'] = tmp['pos22']
data_c['pos6'] = tmp['pos20']
data_c['pos7'] = tmp['pos23']
data_c['pos8'] = tmp['pos1']
data_c['pos9'] = tmp['pos5']
data_c['pos10'] = tmp['pos0']
data_c['pos11'] = tmp['pos4']
data_c['pos12'] = tmp['pos2']
data_c['pos13'] = tmp['pos6']
data_c['pos14'] = tmp['pos3']
data_c['pos15'] = tmp['pos7']
data_c['pos16'] = tmp['pos14']
data_c['pos17'] = tmp['pos13']
data_c['pos18'] = tmp['pos10']
data_c['pos19'] = tmp['pos9']
data_c['pos20'] = tmp['pos11']
data_c['pos21'] = tmp['pos8']
data_c['pos22'] = tmp['pos15']
data_c['pos23'] = tmp['pos12']
```

In [50]:
```
######################################## Pour le test final
tmp = data_test.copy()
data_test['pos0'] = tmp['pos18']
data_test['pos1'] = tmp['pos17']
data_test['pos2'] = tmp['pos19']
data_test['pos3'] = tmp['pos16']
data_test['pos4'] = tmp['pos21']
data_test['pos5'] = tmp['pos22']
data_test['pos6'] = tmp['pos20']
data_test['pos7'] = tmp['pos23']
data_test['pos8'] = tmp['pos1']
data_test['pos9'] = tmp['pos5']
data_test['pos10'] = tmp['pos0']
data_test['pos11'] = tmp['pos4']
data_test['pos12'] = tmp['pos2']
data_test['pos13'] = tmp['pos6']
data_test['pos14'] = tmp['pos3']
data_test['pos15'] = tmp['pos7']
data_test['pos16'] = tmp['pos14']
data_test['pos17'] = tmp['pos13']
data_test['pos18'] = tmp['pos10']
data_test['pos19'] = tmp['pos9']
data_test['pos20'] = tmp['pos11']
data_test['pos21'] = tmp['pos8']
data_test['pos22'] = tmp['pos15']
data_test['pos23'] = tmp['pos12']
```

In [51]:
```
tmp = data_c.copy()
data_c=pd.DataFrame()
data_c['coin1']=tmp[['pos5','pos13','pos16']].apply(tuple , axis=1)
data_c['coin2']=tmp[['pos9','pos4','pos17']].apply(tuple , axis=1)
```

```
data_c['coin3']=tmp[['pos12','pos0','pos18']].apply(tuple , axis=1)
data_c['coin4']=tmp[['pos1','pos8','pos19']].apply(tuple , axis=1)
data_c['coin5']=tmp[['pos14','pos2','pos20']].apply(tuple , axis=1)
data_c['coin6']=tmp[['pos3','pos10','pos21']].apply(tuple , axis=1)
data_c['coin7']=tmp[['pos7','pos15','pos22']].apply(tuple , axis=1)
data_c['coin8']=tmp[['pos11','pos6','pos23']].apply(tuple , axis=1)

data_c['distance']=y
X_c=data_c[['coin1','coin2','coin3','coin4','coin5','coin6','coin7','coin8']]
```

In [52]:

```
##################################### Pour le test final
tmp = data_test.copy()
data_test=pd.DataFrame()
data_test['coin1']=tmp[['pos5','pos13','pos16']].apply(tuple , axis=1)
data_test['coin2']=tmp[['pos9','pos4','pos17']].apply(tuple , axis=1)
data_test['coin3']=tmp[['pos12','pos0','pos18']].apply(tuple , axis=1)
data_test['coin4']=tmp[['pos1','pos8','pos19']].apply(tuple , axis=1)
data_test['coin5']=tmp[['pos14','pos2','pos20']].apply(tuple , axis=1)
data_test['coin6']=tmp[['pos3','pos10','pos21']].apply(tuple , axis=1)
data_test['coin7']=tmp[['pos7','pos15','pos22']].apply(tuple , axis=1)
data_test['coin8']=tmp[['pos11','pos6','pos23']].apply(tuple , axis=1)
```

In [53]:

```
data_c
```

Out[53]:

	coin1	coin2	coin3	coin4	coin5	coin6	coin7	coin8	distance
0	(5, 6, 4)	(2, 3, 6)	(1, 3, 2)	(3, 1, 4)	(1, 5, 2)	(1, 4, 5)	(3, 6, 4)	(6, 2, 5)	11
1	(5, 6, 4)	(2, 3, 6)	(2, 5, 1)	(1, 5, 4)	(1, 3, 4)	(3, 6, 4)	(2, 3, 1)	(2, 5, 6)	11
2	(5, 6, 4)	(1, 2, 5)	(3, 4, 6)	(4, 3, 1)	(2, 3, 1)	(4, 5, 1)	(2, 5, 6)	(3, 2, 6)	11
3	(5, 6, 4)	(1, 3, 2)	(4, 6, 3)	(6, 5, 2)	(1, 3, 4)	(1, 5, 2)	(5, 1, 4)	(2, 6, 3)	9
4	(5, 6, 4)	(3, 1, 4)	(1, 2, 5)	(6, 2, 3)	(5, 1, 4)	(6, 4, 3)	(5, 6, 2)	(1, 2, 3)	12
...	...	...	...	...	...	...	...	...	...
1837074	(5, 6, 4)	(3, 1, 4)	(3, 2, 1)	(4, 1, 5)	(3, 6, 4)	(5, 2, 1)	(2, 6, 3)	(5, 6, 2)	11
1837075	(5, 6, 4)	(4, 6, 3)	(3, 6, 2)	(4, 3, 1)	(5, 1, 4)	(1, 2, 3)	(2, 1, 5)	(6, 2, 5)	9
1837076	(5, 6, 4)	(4, 1, 5)	(3, 1, 4)	(6, 3, 4)	(2, 5, 6)	(2, 3, 1)	(2, 6, 3)	(2, 1, 5)	12
1837077	(5, 6, 4)	(3, 6, 2)	(5, 4, 1)	(6, 3, 4)	(1, 2, 3)	(2, 5, 6)	(1, 3, 4)	(5, 2, 1)	11
1837078	(5, 6, 4)	(5, 2, 6)	(4, 1, 5)	(5, 1, 2)	(2, 6, 3)	(4, 3, 6)	(3, 4, 1)	(1, 2, 3)	11

1837079 rows × 9 columns

In [54]:

```
# Fonction qui permet de coder nos tuples en prenant compte l'orientation ie l'ordre dans le tuple
def transformer_ao(t):
    s=t[0]*t[1]*t[2]*100
    return s+(t[0]+t[1]*2+t[2])
# On l'applique à data
X_ao=pd.DataFrame()
X_ao=X_c.applymap(transformer_ao)
```

In [55]:

```
# On enlève coin1 car se répète
X_ao['coin1'].duplicated().sum()
```

Out[55]: 1837078

In [56]:
```
X_ao=X_ao.drop(['coin1'],axis=1)
```

In [57]:
```
X_ao
```

Out[57]:

	coin2	coin3	coin4	coin5	coin6	coin7	coin8
0	3614	609	1209	1013	2014	7219	6015
1	3614	1013	2015	1211	7219	609	6018
2	1010	7217	1211	609	2015	6018	3613
3	609	7219	6018	1211	1013	2011	3617
4	1209	1010	3613	2011	7217	6019	608
...	...	...	...	...	...	...	...
1837074	1209	608	2011	7219	1010	3617	6019
1837075	7219	3617	1211	2011	608	1009	6015
1837076	2011	1209	7216	6018	609	3617	1009
1837077	3617	2014	7216	608	6018	1211	1010
1837078	6015	2011	1009	3617	7216	1212	608

1837079 rows × 7 columns

In [58]:
```
# Pas de doublon entre nos differentes ligne on a donc gardé l'information sur l'orientation
X_ao.duplicated().sum()
```

Out[58]: 0

In [59]:
```
# Fonction qui permet de coder nos tuples sans prendre compte l'orientation mais on perd de l'information
def transformer_so(t):
    return t[0]*t[1]*t[2]

# On l'applique à data
X_so=pd.DataFrame()
X_so=X_c.applymap(transformer_so)
```

In [60]:
```
# On enlève coin1 car il a la même valeur dans chaque ligne. Il n'apporte aucune information.
X_so.duplicated()
```

Out[60]:
```
0          False
1          False
2          False
3          False
4          False
           ...
1837074     True
1837075     True
1837076     True
1837077     True
1837078     True
Length: 1837079, dtype: bool
```

In [61]:
```
X_so=X_so.drop(['coin1'],axis=1)
```

On trouve des doublons dans nos lignes car on a perdu de l'information sur l'orientation. ATTENTION: il faut laisser ces doublons car ils rajoutent du poid à nos y. Pour une meilleure prédiction, on pourrait rajouter ces poids dans une nouvelle colonne en tenant compte des y, mais nous ne le faisons pas ici.

In [62]:
```
tmp=X_so.copy()
tmp['distance']=y
tmp[tmp.duplicated()]
```

Out[62]:

	coin2	coin3	coin4	coin5	coin6	coin7	coin8	distance
42	72	36	60	12	20	10	6	12
104	36	6	60	20	72	12	10	11
143	20	6	72	12	36	60	10	11
178	12	6	36	60	20	72	10	7
230	72	36	12	60	20	6	10	10
...	...	...	...	...	...	...	...	...
1837074	12	6	20	72	10	36	60	11
1837075	72	36	12	20	6	10	60	9
1837076	20	12	72	60	6	36	10	12
1837077	36	20	72	6	60	12	10	11
1837078	60	20	10	36	72	12	6	11

1816260 rows × 8 columns

In [63]:
```
X_so
```

Out[63]:

	coin2	coin3	coin4	coin5	coin6	coin7	coin8
0	36	6	12	10	20	72	60
1	36	10	20	12	72	6	60
2	10	72	12	6	20	60	36
3	6	72	60	12	10	20	36
4	12	10	36	20	72	60	6
...	...	...	...	...	...	...	...
1837074	12	6	20	72	10	36	60
1837075	72	36	12	20	6	10	60
1837076	20	12	72	60	6	36	10
1837077	36	20	72	6	60	12	10
1837078	60	20	10	36	72	12	6

1837079 rows × 7 columns

Random Forest

In [64]:
```
X_train_ao, X_test_ao, y_train_ao, y_test_ao = train_test_split(X_ao.values, y['distance'].values, test_size=0.5,shuffle=True,random_state=1)
X_train_so, X_test_so, y_train_so, y_test_so = train_test_split(X_so.values, y['distance'].values, test_size=0.5,shuffle=True,random_state=1)
```

In [65]:
```
rdmf_ao = RandomForestClassifier(n_jobs=32,random_state=6)
rdmf_so = RandomForestClassifier(n_jobs=32,random_state=6)
```

In [66]:
```
rdmf_ao.fit(X_train_ao,y_train_ao)
rdmf_so.fit(X_train_so,y_train_so)
```

Out[66]:
```
RandomForestClassifier(n_jobs=32, random_state=6)
```

In [67]:
```
y_pred_train_ao=rdmf_ao.predict(X_train_ao)
y_pred_train_so=rdmf_so.predict(X_train_so)
```

In [68]:
```
print(mean_absolute_error(y_pred_train_ao,y_train_ao))
print(mean_absolute_error(y_pred_train_so,y_train_so))
```

```
2.1773708029816916e-06
0.777674110734547
```

In [69]:
```
y_pred_test_ao=rdmf_ao.predict(X_test_ao)
y_pred_test_so=rdmf_so.predict(X_test_so)
```

In [70]:
```
print(mean_absolute_error(y_pred_test_ao,y_test_ao))
print(mean_absolute_error(y_pred_test_so,y_test_so))
```

```
0.8607474905828816
0.7915714067977443
```

On remarque qu'avec les données qui prennent en compte l'orientation, on a un overfitting énorme et un moins bon score au test que sans orientation. On garde donc la prediction sans orientation qu'on va appliquer à nos données finales.

Appplication finale

Finalement, le meilleur modèle est celui qui correspond à la transformation sans orientation. On applique les mêmes transformations au data_test.

In [71]:
```
data_test=data_test.applymap(transformer_so)
data_test=data_test.drop(['coin1'],axis=1)
```

In [72]:
```
rdmf=RandomForestClassifier(n_jobs=32,random_state=6)
rdmf.fit(X_so,y)
```

Out[72]:
```
RandomForestClassifier(n_jobs=32, random_state=6)
```

In [73]:
```
y_pred_test=rdmf.predict(data_test)
```

In [74]:
```
# Voici le nombre de prédiction finale
pd.DataFrame(y_pred_test).value_counts()
```

Out[74]:
```
11    1753960
10      75413
```

```
12       7707
dtype: int64
```

In [75]:
```
# On télécharge nos données
finale = pd.DataFrame(y_pred_test)
ids = pd.DataFrame(np.arange(1837079,3674159))
ids['ID'] = ids
finale.index = ids['ID']
```

In [76]:
```
finale.to_csv("y_test.csv")
```