Rapport projet Xporters

Nom de l'équipe : TRUCK
Membres de l'équipe : Tristan Jeromin, Melahi Jugurtha, Ziqian Peng, Damien Ouzillou,Virgile Bertrand, Elsa Metivier.
Lien du github : https://github.com/javaax/truck

Introduction :

Nous avons décidé de travailler sur ce projet car nous étions intéressés par la régression linéaire. Nous voulions essayer quelque chose de nouveau puisque nous avons eu un cours d'apprentissage automatique au semestre dernier où nous avons déjà vu une grande partie du contenu de tous les autres projets (en Java par contre, pas en Python). Cependant, nous n'avions pas abordé la régression linéaire en profondeur. Nous avions seulement vu l'aspect théorique en cours sans jamais l'implémenter en quelque langage que ce soit. Ce projet nous permettra donc d'approfondir nos connaissances en résolvant de manière conviviale le projet Xporters.

Description du problème :

Le projet Xporters est un challenge proposé sur Codalab. Il s'agit d'un problème de régression. Le but de ce projet est de trouver quels sont les facteurs qui influent le plus sur la fréquence de passage des voitures à un même endroit tout au long d'une année. Ainsi, on peut prédire en fonction d'un jour donné, de son heure et de ses informations météorologiques, le nombre approximatif de voitures qui passent. Pour cela, nous avons eu accès à un très grand jeu de données composé de 58 paramètres différents pour pouvoir au mieux résoudre le problème (nombreux facteurs météorologiques, données sur de nombreuses dates, …).

Approche choisie :

- Pour la partie **preprocessing**, nous avons d'abord supprimé les données aberrantes. Certaines données extrêmes et/ou incohérentes sont dites aberrantes. Notre but est de les détecter et de nous en débarrasser afin d'avoir un dataset plus cohérent et donc d'améliorer les performances de nos algorithmes. Pour cela, nous avons importé et utilisé le module LocalOutlierFactor de la bibliothèque sklearn.neighbors.
  Ensuite, nous avons effectué une réduction de dimension avec un PCA (Principal Component Analysis) puis nous avons regardé pour quel nombre de dimensions (ici une dimension correspond à un paramètre, donc pour combien de paramètres) le score était optimal. Le PCA consiste à identifier les directions de plus grandes variations du nuage de points : les composantes principales, puis effectuer une projection orthogonale sur l'hyperplan engendré par ces composantes principales. Cela se repose essentiellement sur de l'algèbre linéaire.
  Enfin, nous avons fait une sélection des paramètres les plus importants et nous avons enlevé ceux qui avaient le moins d'impact sur le score final (voir qui n'en avaient pas du tout). Nous avons également cherché les paramètres qui avaient le plus d'impact sur le passage de voitures.

- Pour la partie **modèle**, nous avons placé dans un tableau une liste de modèles à tester sur les données. Les données ont été divisées en deux groupes: le groupe de test et le groupe de validation. Chaque modèle est testé avec la mesure r2 pour les données tests. Ils sont ensuites testés en cross-validation pour les données de validations. A partir de ces résultats, nous avons cherché les meilleurs hyper-paramètres pour les modèles les plus performants. Nous avons utilisé pour cela la méthode RandomizedSearchCV afin de pouvoir obtenir des résultats plus rapidement qu'avec GridSearchCV.

- Pour la partie **visualisation**, nous avons commencé par un affichage en clusters des données. Nous avons donc dû effectuer un PCA (Principal Component Analysis) sur nos données afin de se réduire à seulement 2

paramètres (moyenne des meilleurs paramètres et élimination des plus mauvais qui n'indiquent rien), qui sont les plus importants. Le principe du PCA a été très bien expliqué par le groupe preprocessing qui devait également en faire un pour la réduction de la dimension de leurs données. Dans ce but nous avons voulu ensuite réaliser une régression linéaire des nos données obtenues. Et finalement nous avions également pour objectif d'afficher la performance des modèles qu'on utilise et de trouver lequel est le meilleur à appliquer. Nous avons donc demandé au binôme chargé de la partie modèle de nous transmettre leurs résultats pour qu'on puisse les afficher et donc aboutir à une sélection du meilleur modèle pour continuer notre projet.

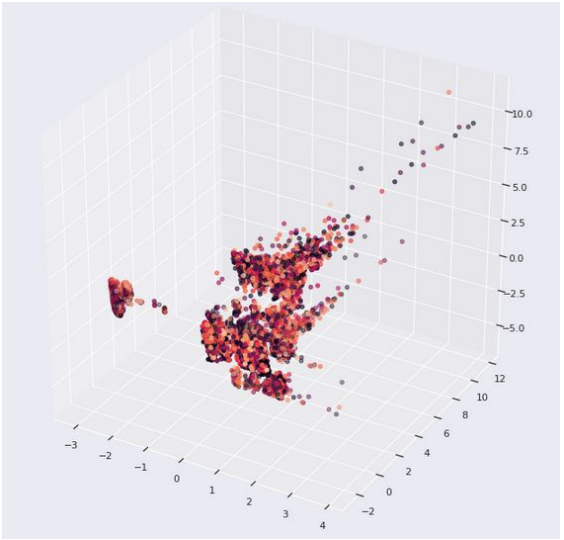
Description des travaux des binômes et résultats obtenus :

- Damien et Virgile ont fait la partie *preprocessing*, disponible dans le notebook README_{preprocessing}. Avant tout, pour pouvoir observer l'amélioration de notre jeu de données, nous avons choisi 3 modèles :
    - Nearest Neighbors
    - Random Forest
    - Gradient Boosting

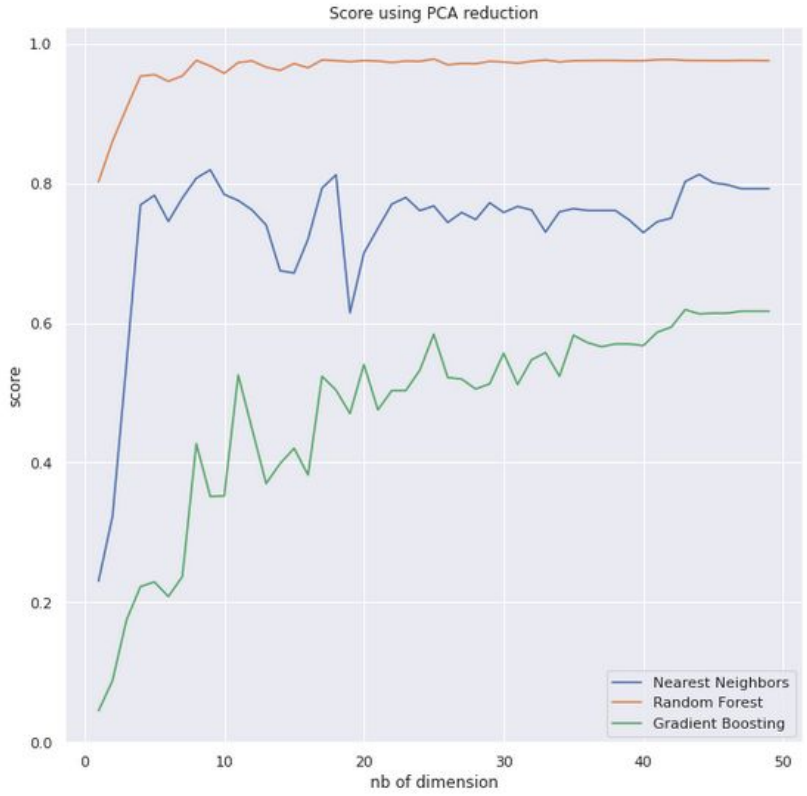
  Ainsi chaque fois que nous modifions notre set de données initiales (par exemple lorsqu'on l'ampute des données "aberrantes") nous pouvons voir l'impact que nos modifications ont sur l'apprentissage. N'étant pas chargés du choix du modèle et des meilleurs hyper-parametres, nous nous sommes contentés de prendre les paramètres par défaut fournis avec ces modèles.
  Pour réaliser la détection des outliers nous avons utilisé comme il a déjà été dit, le module LocalOutlierFactor.
  Nous avons commencé par initialiser le nombre de voisins à 10, mais nous avons remarqué que certaines données pour nous DEVAIENT être enlevées après cette étape (par exemple des données où la température était de 0 degré kelvin, un peu frais…). Nous avons donc augmenté le nombre de voisins à 50 sans que cela nous apporte entière satisfaction et sommes finalement arrivés à 200 voisins. Nous avons alors été incapables de trouver des données qui "n'auraient pas dû" être là. Cependant nous avons prévu de pousser l'analyse un peu plus loin en faisant varier le nombre de voisins et en traçant le score avec nos modèles suivant le nombre de voisins. Cela afin de connaître le nombre de voisin optimum, mais nous ne l'avons pas encore réalisé.

  Pour réaliser la réduction de dimension, nous avons utilisé deux méthodes : PCA et SVD.
  Ces deux méthodes nous ont semblé avoir des résultats très similaires, que ce soit sur le score ou même directement sur les clusters que l'on obtient (que l'on a pu observer seulement en 2 et 3 dimensions). Nous avons commencé par normaliser nos données (maintenant sans les outliers) pour pouvoir avoir une réduction de dimension qui fasse sens.

  Voici les cluster qui apparaissent pour la réduction sur 3 dimensions en utilisant la SVD (la couleur correspond au trafic pour cette donnée)

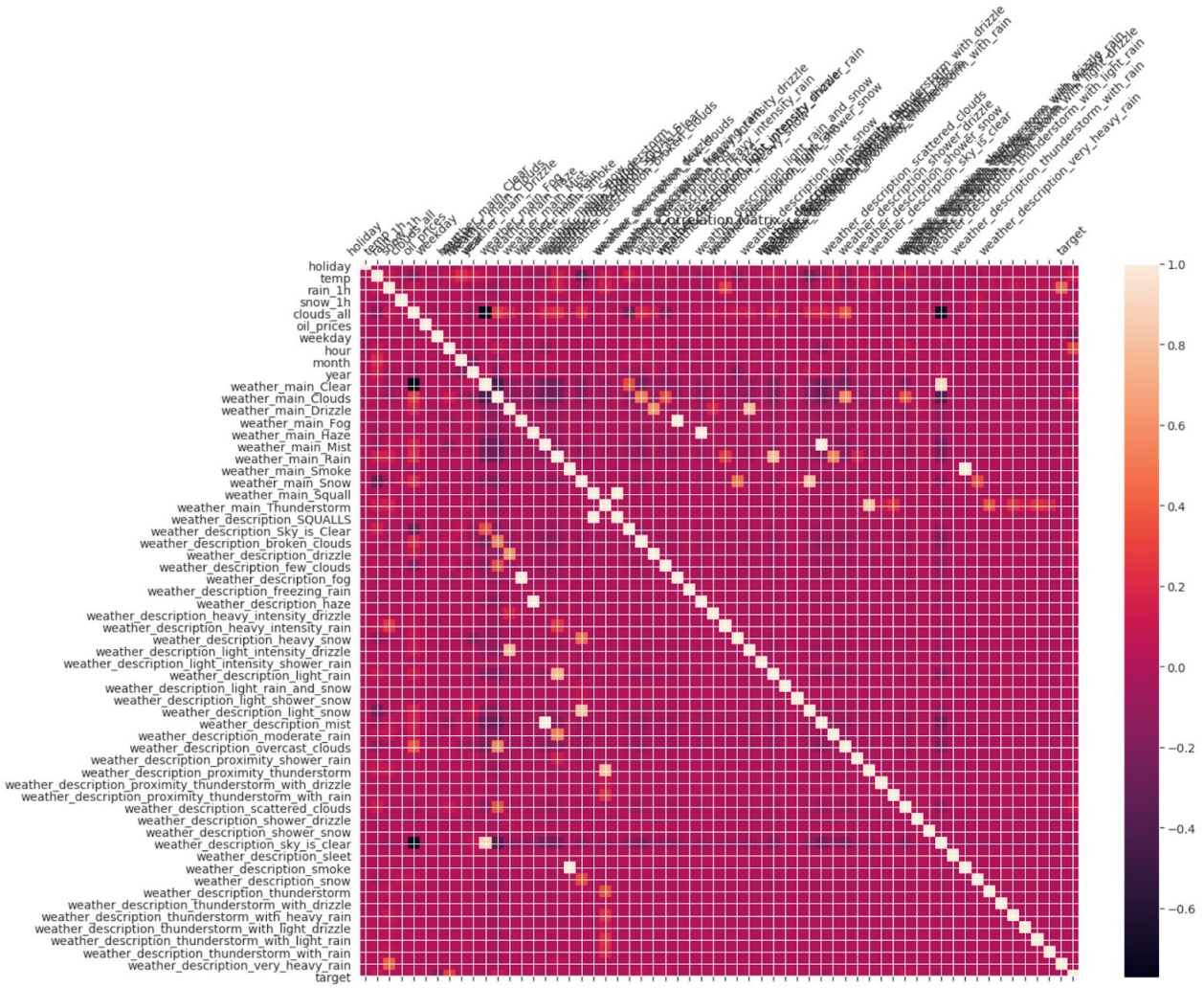


Pour déterminer le nombre de dimension qu'il nous faut choisir, nous avons mesuré le score de nos modèles chaque fois que nous réalisions une réduction de dimension et avons ensuite affiche le score en fonction du nombre de dimension.



On choisira entre 8 et 10 dimensions au final, on remarque que le score a en effet déjà atteint son maximum sur ce nombre de dimensions.

Enfin, concernant la sélection des paramètres, nous avons une matrice de corrélation extrêmement utile. En effet, elle nous permet de voir la corrélation entre chaque paramètre. Ce qui nous intéresse c'est d'avoir des paramètres corrélé au maximum avec le target (que l'on peut lire sur la dernière colonne/ligne de la matrice) et au maximum dé-corrélés entre eux (afin de ne pas avoir "plusieurs fois" la même information).



Le mode de fonctionnement de cette matrice est simple : pour chaque paramètre, elle indique la corrélation entre ce paramètre et tous les autres, ce qui explique que toute la diagonale est au maximum de corrélation possible (un paramètre est forcément en corrélation avec lui même). On peut voir à quel point un paramètre est en lien avec un autre à l'aide de la couleur des cases, plus elles sont claires plus les paramètres sont corrélés entre eux. Pour attribuer un score à chaque paramètre, il suffit simplement de regarder la corrélation entre ce paramètre et le target, à savoir le trafic de voitures. Cette matrice nous sert donc à savoir quels sont les paramètres les plus importants. Nous pouvons choisir le nombre de paramètres les plus impactants que nous voulons afficher, par ordre décroissant comme vous pouvez le voir ci-dessous dans la dernière ligne de code avec le "[:20]" (on a décidé d'afficher les 20 paramètres les plus impactants, donc) .

```
#Affiche les données qui sont les plus importantes de maniere decroissante

print('Most important features according to the correlation with target')
most_important_features = outliers.corr()['target'].sort_values(ascending=False)[:20]
print (outliers.corr()['target'].sort_values(ascending=False)[:20], '\n')
```

```
Most important features according to the correlation with target
target                                          1.000000
hour                                            0.351034
temp                                            0.135739
weather_main_Clouds                             0.118518
weather_description_scattered_clouds            0.084405
weather_description_broken_clouds               0.064993
clouds_all                                      0.063926
weather_description_few_clouds                  0.043727
weather_description_proximity_shower_rain       0.034116
weather_description_haze                        0.018787
weather_main_Haze                               0.018787
weather_description_Sky_is_Clear                0.018285
weather_description_overcast_clouds             0.017495
weather_description_light_intensity_drizzle     0.015465
weather_description_light_rain                  0.013835
weather_main_Rain                               0.010323
weather_description_light_shower_snow           0.008574
weather_description_light_intensity_shower_rain 0.007103
weather_main_Drizzle                            0.006786
weather_description_shower_snow                 0.006185
Name: target, dtype: float64
```

Nous voyons que seuls les 15 premiers paramètres ont un impact plus ou moins conséquent (c'est-à-dire 0.01 ou plus de corrélation avec target). Nous avons donc décidé de créer un tableau de données mais avec au plus 15 paramètres et non plus 58. Nous gagnerons beaucoup de temps de calcul sans pour autant perdre en efficacité de prédiction. Pour se faire, nous avons utilisé les modules VarianceThreshold, SelectKBest et SelectFromModel (en utilisant le model *LassoCV*) de la bibliothèque sklearn.feature_selection.
Nous avons également pensé à combiner plusieurs paramètres en un seul (en particulier ceux portant sur la météo, qui nous pensons, ont un impact assez similaire sur le target.

- Tristan et Jugurtha ont fait la partie *modèle* qui est disponible dans le répertoire *README_{model}*. Après avoir testé les différents modèles avec la mesure r2 et en cross-validation, il semble que le modèle RandomForestRegressor soit le plus performant. Les mesures pour chaque modèle sont indiquées dans le tableau ci-dessous:

Method	Nearest Neighbors	ElasticNet	Decision tree	Random Forest	Gradient Boosting
Training	0.8343	0.1597	0.9671	0.9751	0.7352
CV	0.64	0.16	0.90	0.94	0.74
Valid	0.748569295	0.1625867749	0.9067954143	0.9413530963	0.7341450885

Le modèle RandomForestRegressor fonctionne de la manière suivante:

1) On divise les données en de nombreux sous-groupes.
2) Pour chaque sous-groupe, un arbre de décision est créé.

3) Les variables de segmentation sont choisies aléatoirement, et chaque arbre est divisé selon la meilleure segmentation.
4) Chaque nouvelles données présentées au modèle sont évaluées en fonction de tous les arbres.

Nous avons ensuite cherché les meilleurs hyper-paramètres de RandomForestRegressor avec RandomizedSearchCV qui prend pour fonction de mesure r2. On trouve les paramètres suivants:

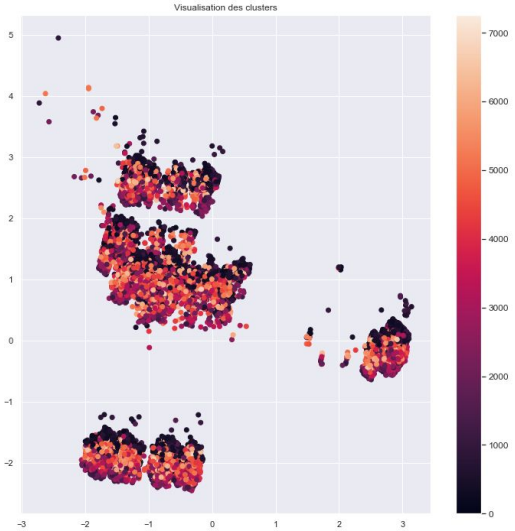
random_state	5
n estimator	140
max_features	auto
max_depth	50
criterion	friedman mse

En appliquant ces nouveaux hyper-paramètres à RandomForestRegressor, on obtient les scores suivants:

Training	0.9784
CV	0.95
Valid	0.9473205799

Nous remarquons que le modèle fonctionne légèrement mieux avec ces nouveaux paramètres.

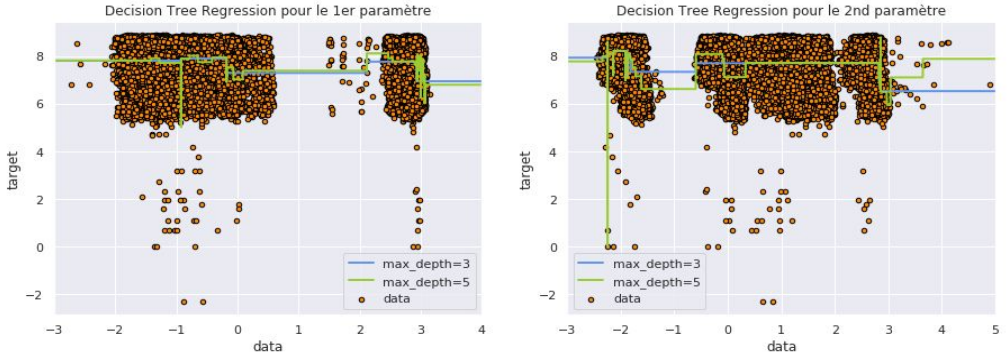
- Elsa et Ziqian ont fait la partie *visualisation* des données et des résultats. Le fichier de cette partie est dans le répertoire ***README_{visualisation}*** sur le lien de notre github. Pour la visualisation de *cluster,* nous avons décidé d'appliquer le *PCA* de *sklearn.decomposition.*



Ici on voit très clairement trois clusters bien séparés. On a affiché la couleur des données en fonction du nombre de voitures qui passent (on voit d'après la barre à droite que les couleurs vers le beige représentent les données où il y a

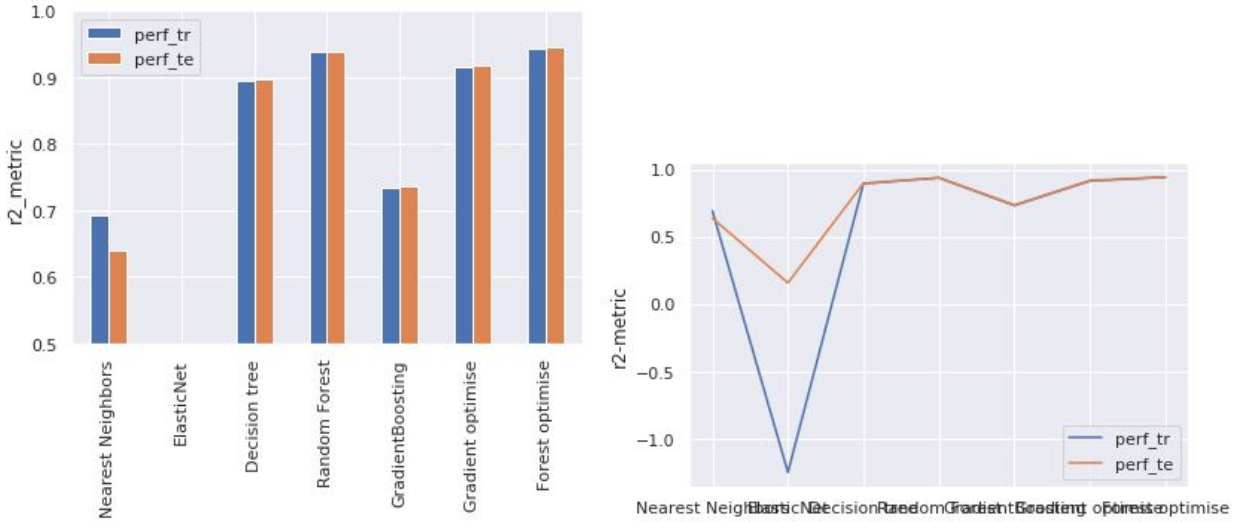
beaucoup de voitures qui passent, jusqu'à 7000, alors que plus on se rapproche du violet, plus le nombre de voitures est beaucoup plus faible, parfois inférieur à 1000).

Pour la régression des données, on a conservé le PCA qu'on avait obtenu et on a utilisé le module python du *DecisionTreeRegressor* sur nos 2 paramètres obtenus grâce au PCA. On obtient deux figures affichées que l'on a mis dans un espace logarithmique :



Nous avons pris le logarithme de notre Y_train (nombre de voitures) pour pouvoir affiner la représentation et donc mieux extraire des conclusions de ces figures.
On peut voir en effet qu'il y a des données situées à l'écart des autres (situées bien en dessous) qui représentent les voitures qui passent peu. On voit que notre modèle a du mal à prédire les jours/ paramètres lorsque le nombre de voitures est faible. On peut également dire que nos 2 paramètres obtenus par le PCA sont donc des paramètres optimaux. N'ayant eu les résultats des modèles que tardivement, nous avons juste affiché leur performance. Bien sûr, maintenant nous allons faire fonctionner notre régression avec plusieurs modèles et pouvoir obtenir de nouveaux résultats par la suite.

Ensuite, on a utilisé le module *pandas* pour afficher les performances des modèles utilisés. Nous avons repris la base du TP2 et ainsi on obtient les graphiques suivants :



Pour obtenir les scores, nous avons repris ceux obtenus par ceux qui avaient fait le fichier README_{model}.ipynb donc du binôme chargé de faire la partie modèle. Ensuite, ayant les scores à disposition, nous avons profité des méthodes de la librairie *pandas.DataFrame.plot* pour les représenter.
Nous avons constaté qu'il n'y pas de overfitting et les modèles *Forest optimise* et *Random Forest* ont les plus hauts scores.
Par ailleurs, le modèle *ElasticNet* a un score très bas pour les données de validation et un score négatif pour les données d'apprentissage. En effet, sur le graphique par barres, nous avons mis les valeurs entre 0,5 et 1 pour mieux comparer et

bien sûr nous ne voyons pas ceux du modèle Elastic Net puisqu'ils sont bien trop bas. Donc ce modèle ne peut pas du tout faire la régression de notre projet. On doit donc privilégier sans aucun doute les modèles Forest Optimise et Random Forest pour la suite de notre projet, sauf si le binôme modèle nous apporte d'autres informations et modèles plus performants lors de notre prochaine séance.

Conclusion :

Nous pensons avoir bien apprivoisé le mini projet Xporters. Nous avons, en binômes, rassemblé nos résultats sur ce rapport pour avoir un rendu global de notre avancement. Nous voyons que nous avons maintenant un meilleur score que celui que nous avions initialement, ce qui est de bon augure pour la suite. Nous espérons avoir été à la hauteur et souhaitons continuer dans cette voie, tout en sachant que nous avons des idées pour encore faire mieux comme vous avez pu le lire dans ce rapport.

Références :

[1] README.ipynb fourni avec le starting kit.
[2] Scikit-learn documentation : Pipeline and FeatureUnion, Preprocessing data, Model evaluation, Feature selection : http://scikit-learn.org
[3] Scikit-learn documentation feature selection : http://machinelearningmastery.com/feature-selection-in-python-with-scikit-learn/
[4] Scikit-learn documentation outlier detection : https://scikit-learn.org/stable/modules/outlier_detection.html
[5] Scikit-learn documentation PCA : https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html