# Rapport projet Xporters

**Nom de l'équipe :** TRUCK
**Membres de l'équipe :** Tristan Jeromin, Melahi Jugurtha, Ziqian Peng, Damien Ouzillou,Virgile Bertrand, Elsa Metivier.
**URL du challenge:** https://codalab.lri.fr/competitions/652
**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 SVD et 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 (voir Figure 3). 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 (voir Figure 5).

- 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 (voir Figure

5). 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 (voir Figure 7).

- 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 (voir Figure 12). 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 :

- **Preprocessing**

  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 (voir Figure 0). 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.

```
clf = LocalOutlierFactor(n_neighbors= 200)
outliers = data.copy()
outliers['outliers'] = pd.Series(clf.fit_predict(outliers), index = outliers.index)
```

```
#Les valeurs de la colonne outliers sont 1 ou -1 suivant si elles ont été considérées comme des
outliers
# par notre modèle dans la cellule au dessus ou non
# On garde donc celles qui sont à 1, les autres étant les outliers

outliers = outliers[outliers['outliers'] == 1]
outliers = outliers.drop('outliers', 1)
```

Figure 0 : Code Python qui permet d'obtenir un tableau de données sans les données aberrantes

Voici le bout de code concerné. Comme nous l'avons déjà expliqué, le "n_neighbors = 200" correspond aux 200 voisins dont on parlait ci-dessus.
D'abord, on crée une copie de nos données qu'on nomme outliers, puis on fait en sorte que les valeurs de la colonne outliers du tableau de données outliers soient 1 ou -1 selon si elles sont considérées comme des outliers ou non, 1 correspondant aux données fiables et -1 aux outliers.
Les 2 dernières lignes permettent de ne conserver que les données dont la valeur est 1 dans la colonne outliers, donc les données qui sont jugées fiables. On se débarrasse des données dont la valeur dans la colonne outliers est -1 car elles sont considérées comme aberrantes.

```
data[data['target'] == 0] #exemple de valeurs aberrantes
```

| | holiday | temp | rain_1h | snow_1h | clouds_all | oil_prices | weekday | hour | month | year | ... | weather_description_s |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **19804** | 0 | 296.68 | 0.0 | 0.0 | 40 | 79.246760 | 5 | 18 | 7 | 2016 | ... | 0 |
| **28248** | 0 | 295.90 | 0.0 | 0.0 | 90 | 66.353169 | 5 | 23 | 7 | 2016 | ... | 0 |

2 rows × 60 columns

```
outliers[outliers['target'] == 0] #exemple de valeurs aberrantes
```

| | holiday | temp | rain_1h | snow_1h | clouds_all | oil_prices | weekday | hour | month | year | ... | weather_description_smoke | w |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

0 rows × 60 columns

Figure 1 : Code Python pour tester si les données aberrantes ont été supprimées

Voici un test que nous avons effectué pour voir si les données aberrantes étaient effectivement effacées dans le tableau outliers (voir Figure 1), nous avons trouvé qu'il y a 2 données dans le tableau initial data où le target vaut 0 ce qui est évidemment incohérent. On voit que lorsque l'on cherche dans le tableau de données outliers sans valeurs aberrantes (si tout s'est bien passé), il n'y a effectivement pas de donnée où le target vaut 0 donc elles ont été supprimées ce qui montre que notre objectif est atteint.

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. (voir Figure 2)

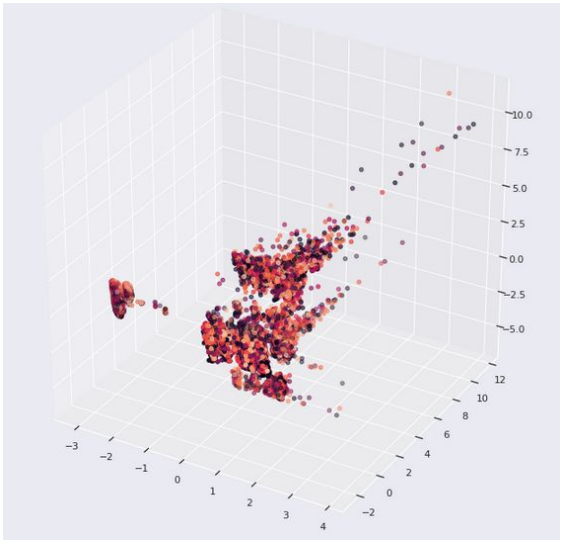


Figure 2 : Clusters avec la réduction sur 3 dimensions avec la SVD

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.

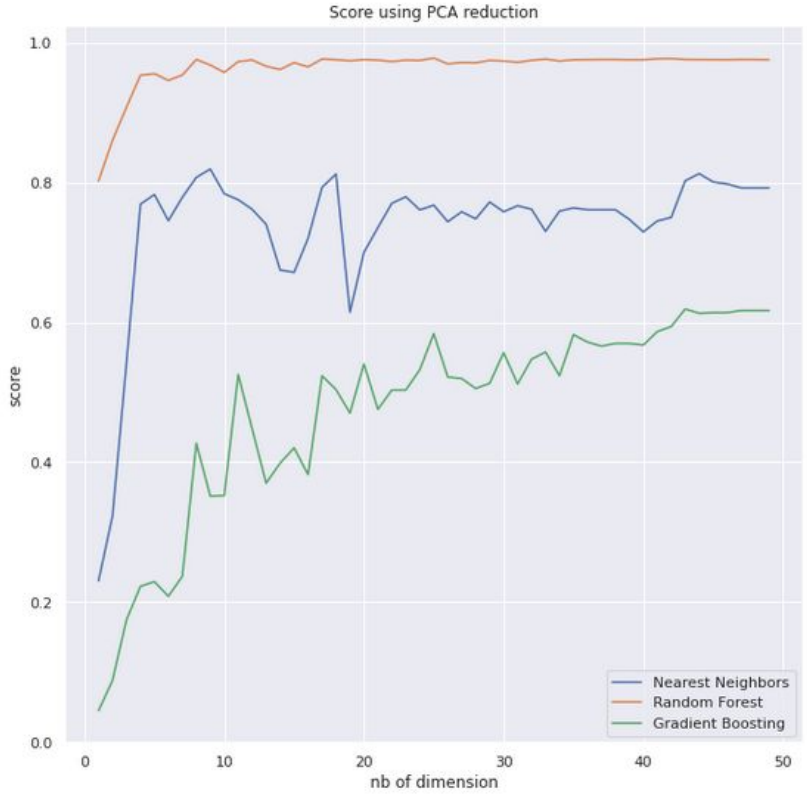


Figure 3 : Score de différents modèles en fonction du nombre de dimensions

On choisira entre 15 et 20 dimensions au final, on remarque que le score a en effet déjà atteint son maximum et s'est stabilisé sur ce nombre de dimensions (voir Figure 3).

Enfin, concernant la sélection des paramètres, nous avons une matrice de corrélation extrêmement utile (voir Figure 4). 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).

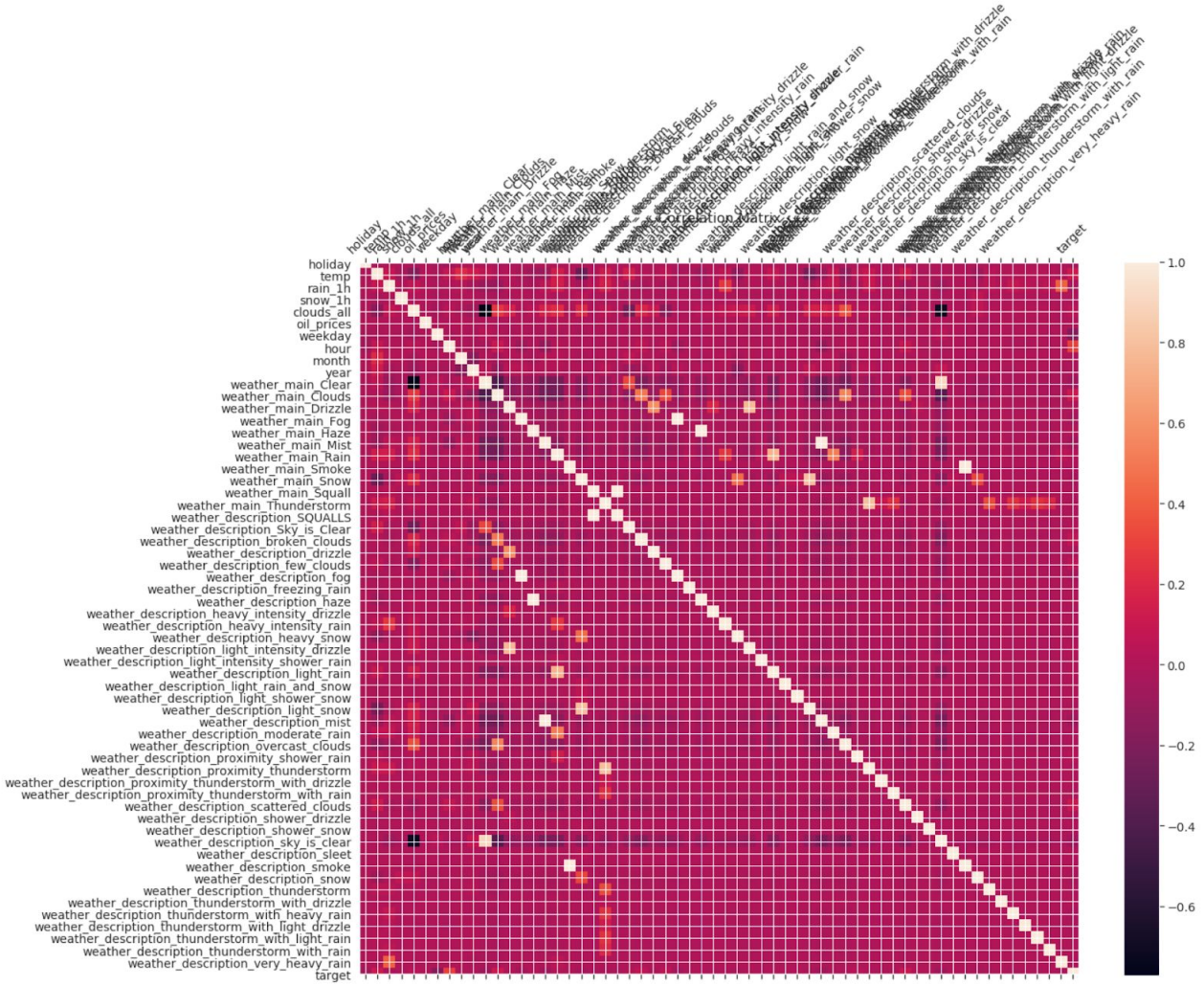


Figure 4 : Matrice de corrélation entre tous les paramètres de notre jeu de données

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 Figure 5 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
```

Figure 5 : Code Python qui permet d'afficher quels sont les paramètres les plus importants (i.e les plus corrélés avec target)

Nous voyons (voir Figure 5) 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).

- **Modèle**
  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 la Figure 6:

| 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 |

Figure 6 : Tableau des mesures pour chaque modèle

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 (voir Figure 7) :

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

Figure 7 : Meilleurs hyper-paramètres de RandomForestRegressor avec RandomizedSearchCV

En appliquant ces nouveaux hyper-paramètres à RandomForestRegressor, on obtient les scores suivants (voir Figure 8) :

| | |
|---|---|
| Training | 0.9784 |
| CV | 0.95 |
| Valid | 0.9473205799 |

Figure 8 : Nouveau score après avoir appliqué les hyper-paramètres de la Figure 7

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

| | Nombre d'exemples | Nombre de features | is sparse ? | Variables catégorielles? | Données manquantes? |
|---|---|---|---|---|---|
| Training | 38563 | 59 | Non | Non | Non |
| Valid | 4820 | 59 | Non | Non | Non |
| Test | 4820 | 59 | Non | Non | Non |

Figure 9 : Statistiques sur les données

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn import svm
from sklearn import linear_model
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import DotProduct, WhiteKernel
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import ElasticNet
from sklearn import linear_model
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor

model_name = [
    "Nearest Neighbors",
    "ElasticNet",
    "Decision tree",
    "Random Forest",
    "GradientBoosting",
    "Gradient optimise",
    "Forest optimise"]

model_list = [
    KNeighborsRegressor(2),
    ElasticNet(random_state=0),
    DecisionTreeRegressor(),
    RandomForestRegressor(n_estimators=10),
    GradientBoostingRegressor(random_state=1, n_estimators=10),

    GradientBoostingRegressor(random_state=2, n_estimators=100, max_features = 'auto',
                              loss = 'huber',
                              learning_rate = 0.1,
                              criterion =  'friedman_mse'),

    RandomForestRegressor(random_state = 1,
        n_estimators = 180,
 max_features = 'auto',
 max_depth = 50,
 criterion= 'mse')
]

trained_model_name = model_dir + data_name
```

Figure 10 :

Pour tester les différents modèles, nous avons commencé par les stocker dans un tableau nommé "model_ list" (voir figure 10). Les noms de ces modèles sont stockés dans le tableau "model_name" afin de pouvoir les identifier par la suite.

```
from sklearn import model_selection
from sklearn.model_selection import train_test_split

X_train = D.data['X_train']
Y_train = D.data['Y_train']

assert len (X_train) == len (Y_train)

X_entrainement,X_validation,Y_entrainement,Y_validation = train_test_split(X_train,Y_train,test_size=0.33,random_stat
```

```
#Affichage des performances pour chaque modele

from sklearn.metrics import make_scorer
from sklearn.model_selection import cross_val_score
from libscores import get_metric
metric_name, scoring_function = get_metric()

for i in range(7):
    M = model_list[i]
    M.fit(X_entrainement,Y_entrainement)

    Y_hat_train = M.predict(D.data['X_train'])
    Y_hat_valid = M.predict(D.data['X_valid'])
    Y_hat_test = M.predict(D.data['X_test'])

    print('\n',model_name[i])
    print(metric_name, '= %5.4f' % scoring_function(Y_train, Y_hat_train))
    scores = cross_val_score(M, X_validation, Y_validation, cv=5, scoring=make_scorer(scoring_function))
    print('Cross-validation: %0.2f (+/- %0.2f)' % (scores.mean(), scores.std() * 2))
```

Figure 11 :

Les données X_train et Y_train sont séparées en deux pour former les données de tests et de validations (Figure 11). On utilise pour cela la méthode “train_test_split”. Enfin, on parcourt chaque modèle de “model_list” a l’aide d’une boucle for et on les entraînent avec la méthode “fit” avant d’afficher pour chacun leurs scores pour la mesure r2 et pour la cross-validation (Figure 11).

- **Visualisation**

  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.*

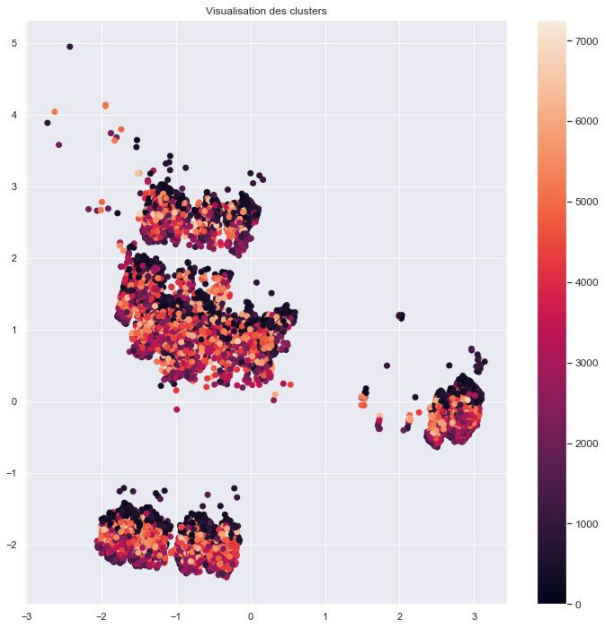


Figure 12 : PCA pour visualiser des clusters

Ici on voit très clairement trois clusters bien séparés (voir Figure 12). 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 :

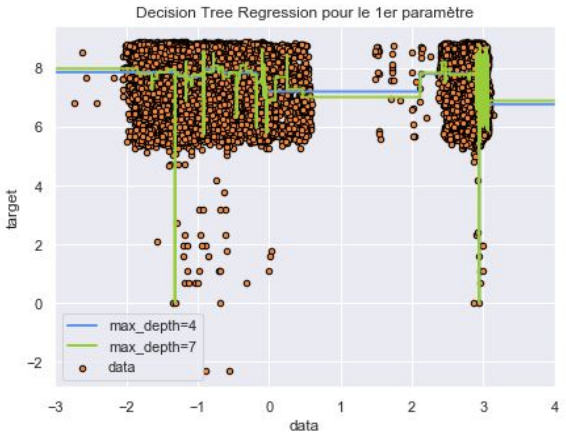


Figure 13 : Régression des données suivant le 1er paramètre de notre PCA obtenu à l'aide de Tree Regressor

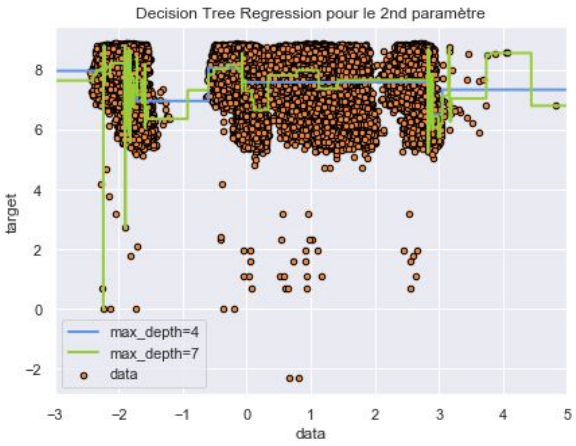


Figure 14 : Régression des données suivant le 2nd paramètre de notre PCA obtenu à l'aide de Tree Regressor

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 sur les figure 13 et 14) 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 (figure 13 et 14). 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 :

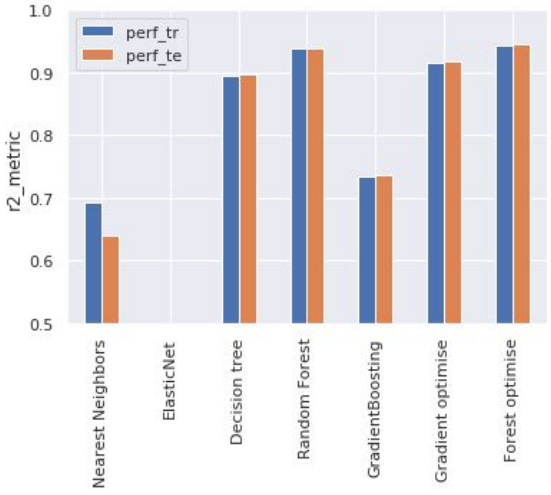


Figure 15 : Diagramme en bâton des performances de chaque modèle


Figure 16 : Performances des modèles en courbe

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 (Figure 15).

Par ailleurs, le modèle *ElasticNet* a un score très bas pour les données de validation et un score négatif pour les données d'apprentissage (Figure 16). 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 (Figure 15). 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.

Nous allons expliquer plus en détails certaines parties du codes :

```
def kmoy (x,Y):
    from sklearn.cluster import KMeans
    from sklearn.datasets import make_blobs


#exemple pour visualiser les clusters avec la méthode des K-moyennes
    plt.figure(figsize=(12, 12))

    random_state = 170

        #on fait pca sur les données
    scaler = StandardScaler()
    scaler.fit(x)
    scaled_data = scaler.transform(x)
    pca = PCA(n_components = 2)
    pca.fit(scaled_data)
    x_pca = pca.transform(scaled_data)
    if(x_pca.shape== (38563, 2)):
        y_pred = KMeans(n_clusters=2, random_state=random_state).fit_predict(x_pca)

        plt.scatter(x_pca[:, 0], x_pca[:,1], c=Y)
        plt.title("Visualisation des clusters")
            #on essaye d'afficher la légende c'est-à-dire la couleur des points correspond à une
certaine quantité
            #en l'occurence, Y est en fait target donc le nombre de voitures
            #qui passent donc on veut savoir à combien de voitures correspondent les couleurs
        cb = plt.colorbar()
        cb.ax.tick_params(labelsize=12)
        plt.show()
    return 0
```

Figure 17 : Code de la visualisation en clusters

Pour la **visualisation en clusters** (Figure 17), on crée une fonction kmoy qui prend en paramètres x et Y (on appliquera *['X_train']* et *['Y_train']*). Avec x, on crée ensuite un scaler pour standardiser nos features et on leur applique *fit* pour faire des moyennes des features etc… Ainsi nos données sont normalisées. On crée un PCA pour réduire nos données en 2 dimensions et on l'applique à nos données que l'on vient de modifier. Le *pca.transform* applique donc la réduction de dimension. Ensuite, une fois notre jeu de données normalisé et aux bonnes dimensions, on peut appliquer l'algorithme des K-moyennes. On affiche grâce à *scatter*, le premier paramètre obtenu par le pca en fonction du second et en appliquant la couleur correspondante à chacune des données, soit grâce à Y. On rajoute également la barre des couleurs qui indique le nombre de voitures qui

passent. Pour appliquer notre algorithme des K-moyennes, on fait un test pour bien avoir les bonnes dimensions désirées pour nos données de x_pca). Cette fonction dans le main ne met aucun mal à tourner.

```
def classifier (x, Y):
    fig, (ax1, ax2) = plt.subplots(1, 2,figsize=(15,5))

    Y[Y==0] = 0.1

    Y = np.log(Y)
    #on fait pca sur les données
    scaler = StandardScaler()
    scaler.fit(x)
    scaled_data = scaler.transform(x)
    pca = PCA(n_components = 2)
    pca.fit(scaled_data)
    x_pca = pca.transform(scaled_data)

    if(x_pca.shape== (38563, 2)):

        x1=x_pca[:,0][:, np.newaxis]
        x2=x_pca[:,1][:, np.newaxis]

        regr_1 = DecisionTreeRegressor(max_depth=4)
        regr_1b = DecisionTreeRegressor(max_depth=7)
        regr_2 = DecisionTreeRegressor(max_depth=4)
        regr_2b = DecisionTreeRegressor(max_depth=7)
        regr_1.fit(x1, Y)
        regr_1b.fit(x1, Y)
        regr_2.fit(x2, Y)
        regr_2b.fit(x2, Y)


        # Predict
        X_test1= np.linspace(-3.0, 4.0, num=x.size)[:, np.newaxis]
        X_test2= np.linspace(-3.0, 5.0, num=x.size)[:, np.newaxis]
        y_1 = regr_1.predict(X_test1)
        y_1b = regr_1b.predict(X_test1)
        y_2 = regr_2.predict(X_test2)
        y_2b = regr_2b.predict(X_test2)
        if(y_1.shape==y_1b.shape and y_1b.shape==y_2.shape and y_2.shape==y_2b.shape):

            # Plot the results
            #plt.figure()
            ax1.scatter(x1, Y, s=20, edgecolor="black",c="darkorange", label="data")
            ax1.plot(X_test1, y_1, color="cornflowerblue", label="max_depth=4", linewidth=2)
            ax1.plot(X_test1, y_1b, color="yellowgreen", label="max_depth=7", linewidth=2)
            ax1.set_xlabel("data")
            ax1.set_ylabel("target")
            ax1.set_xlim(-3,4)
            ax1.set_title("Decision Tree Regression pour le 1er paramètre")
            ax1.legend()

            ax2.scatter(x2, Y, s=20, edgecolor="black",c="darkorange", label="data")
            ax2.plot(X_test2, y_2, color="cornflowerblue", label="max_depth=4", linewidth=2)
            ax2.plot(X_test2, y_2b, color="yellowgreen", label="max_depth=7", linewidth=2)
            ax2.set_xlabel("data")
            ax2.set_ylabel("target")
            ax2.set_xlim(-3,5)
            ax2.set_title("Decision Tree Regression pour le 2nd paramètre")
            ax2.legend()
    return 0
```

Figure 18 : Code de la régression

Afin de visualiser les erreurs de la **régression**, on crée une fonction classifier qui prend en paramètre x et Y(qui seront D.data[X_train] et D.data[Y_train]) et on applique le PCA et le *DecisionTreeRegressor* (figure 18) :

Pour commencer, on prend nos données x et les étiquettes Y et on standardise x. On modifie Y pour pouvoir se placer dans un espace log donc on enlève possiblement les cas où une valeur de Y est égale à zéro. De plus, on réduit x en 2 dimensions grâce à la méthode avec le PCA (comme expliquée auparavant) et on met les données normalisées réduites dans la variable x_pca (cf les lignes 8-13 de Figure 18). Ensuite, on stocke dans x1 les données de la première dimension de x_pca (1er paramètre) et dans x2 celles de la seconde dimension(2nd paramètre). Pour chacun, on applique les *DecisionTreeRegressor* de différentes profondeurs pour la régression. Par exemple, pour x1:

- D'abord on crée un *DecisionTreeRegressor* avec *max_depth=4* et un autre avec *max_depth=7*.
- Ensuite, on leur fait connaître notre jeu de données x1 en appliquant *fit(x1, Y)*.
- Enfin, on peut utiliser le modèle pour la régression selon x1 et on applique *predict* pour prédire nos données sur la plage de données située entre -3 et 4 pour une meilleure visualisation.
  (cf les lignes 20-38 de Figure 18 )

On visualise le résultat grâce à *scatter* et *plot.* On créer une figure pour chaque dimension de données, c'est-à-dire, un pour x1 et un autre pour x2, donc pour chaque paramètre. Pour chacun, on dessine les données

réelles par *scatter* (cf la ligne 37 et 46 de Figure 18) et on dessine les données prédites par *DecisionTreeRegressor* en appliquant *plot* (cf les lignes 43,44,52,53 de Figure 18). De plus, on met des légendes et on modifie les valeurs des paramètres de *scatter* et de *plot* etc. pour obtenir un graphe plus clair et plus joli.

Pour le **test**, cette partie de code fonctionne bien et il reproduit deux figures pertinents à notre projet. On a fait des tests pour que les dimensions de nos arrays aient toutes la même taille sinon on s'arrête. Il fonctionne aussi si on modifie l'intervalle des valeurs sur lequel on prédire ou si on modifie la valeur de *max_depth.*

## 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

[6] Scikit-learn documentation Decision Tree Regressor :
https://scikit-learn.org/stable/auto_examples/tree/plot_tree_regression.html#sphx-glr-auto-examples-tree-plot-tree-regression-py

[7] Scikit-learn documentation Kmeans :
https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html