L2-Mini Projet
Xporters - groupe Caravan
Membres : Ouassim Meftah, Jeyanthan Markandu, Julien Rolland, Mohamed Oumezzaouche, Pierre Cornemillot, Martin Vitani
URL du challenge: https://codalab.lri.fr/competitions/652
Repo GitHub du projet : https://github.com/pierre6398/caravan

XPorters

Contexte et description du problème :

Le challenge qui nous a été proposé est une prédiction du trafic sur une autoroute, c'est-à-dire le nombre de voitures sur cette autoroute à une date et une heure donnée, en fonction d'informations sur la météo qu'il faudra juger utiles ou non. Pour cela, nous avons à notre disposition un ensemble de données qui permettront d'effectuer une régression. Notre groupe divisera son travail en trois tâches principales (preprocessing, modélisation et visualisation) qui seront détaillées plus bas.

Description des classes :

Preprocessing : Mohamed OUMEZZAOUCHE, Ouassim MEFTAH

L'objectif de notre groupe est de faire du tri parmi nos données, garder les plus utiles afin d'améliorer l'efficacité du modèle. Pour cela nous avons commencé par chercher comment enlever de notre set de données les colonnes qui sont composées à au moins 95% de valeurs 0 (on utilise la fonction percentile et la formule de la variance). Ce sont les variables inutiles pour le modèle. Par la suite nous avons fait en sorte de trouver et supprimer les valeurs extrêmes/incohérentes avec data.drop. Nous avons ainsi écrit deux fonctions distinctes pour détecter les outliers . Ensuite le nouveau set de données a été utilisé par la team modèle afin de voir si une amélioration est visible sur le score. Enfin, nous avons essayé de redimensionner les données afin qu'elle paraisse moins lourdes et plus homogènes pour un preprocessing plus efficace.

Model : Julien ROLLAND, Jeyanthan MARKANDU

L'objectif de notre groupe est de choisir le meilleur modèle ainsi que les meilleurs hyper-paramètres. pour cela nous avons commencé par essayer différents modèles, 7 en tous. Parmis eux nous avons choisis celui qui donnait le meilleur score : GradientBoostingRegressor. Ensuite nous avons utilisé les méthodes GridSearchCV et RandomizedSearchCV du module sklearn qui tentent de trouver les meilleurs paramètres respectivement en testant avec toutes les valeurs de manière exhaustive et en testant seulement certaines valeurs choisies aléatoirement.

Visualisation : Pierre CORNEMILLOT, Martin VITANI

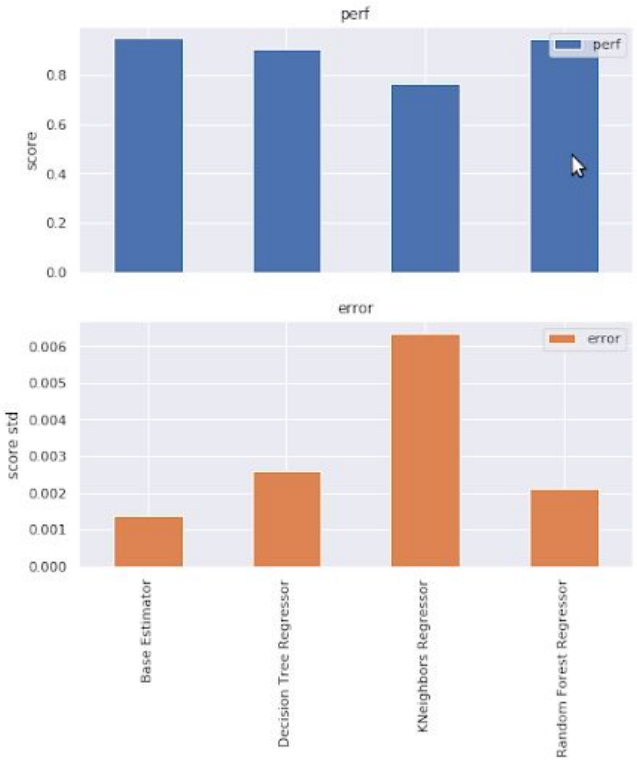
L'objectif de notre groupe est d'afficher les données de la meilleure manière possible, afin de pouvoir les analyser et les comprendre. Par exemple, le problème abordé étant une régression, il peut être utile de représenter les performances d'un modèle de prédiction sur les données d'entraînement avec une droite de prédiction, ainsi que l'erreur résiduelle par rapport à celle-ci afin de connaître l'acuité de cette prédiction. Pour tester la validité de ces visualisations de performances, nous avons utilisé différentes méthodes de régression de Sklearn. De plus, une visualisation de clusters concernant certains features permettra d'étudier l'utilité de ceux-ci et d'établir des connections entre l'évolution du score et les modifications en preprocessing. Seront présentés plus bas des morceaux de code et des exemples des visualisations que nous avons rajoutés.

Résultats préliminaires :

Le notebook README.ipynb original permet d'obtenir un score de 0,9467. Les résultats étant donc déjà très corrects dès le départ (plus que pour une classification), les différents modèles de prédiction ne seront pas facile à différencier, et trouver un meilleur modèle encore moins (comme le montre la figure 1 page suivante). Il est néanmoins possible d'améliorer ce score via le preprocessing et quelques modifications sur les hyper-paramètres de chaque modèle étudié.

Visualisation :

Figure 1 : Comparaison des performances

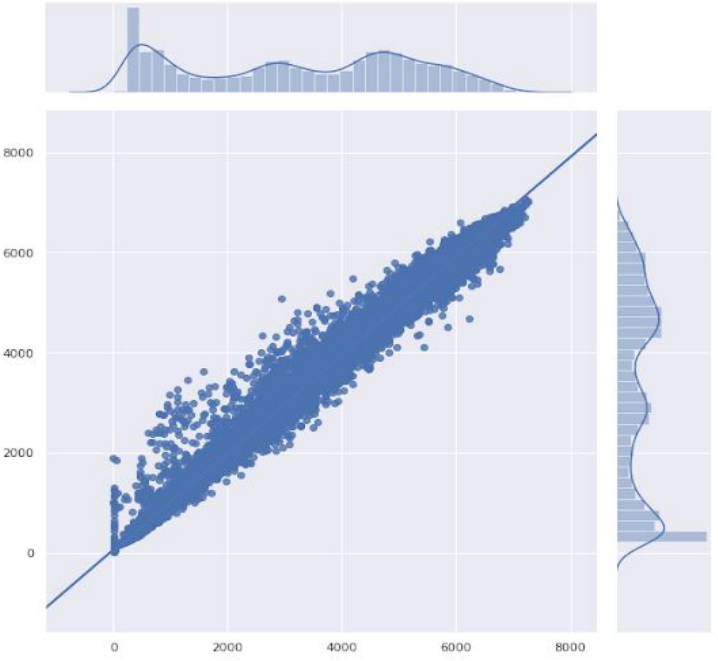


Il nous paraissait intéressant d'afficher le score calculé pour différents modèles afin d'aider à déterminer le plus efficace, ainsi que la variance correspondante, car un modèle performant mais très imprécis n'est pas forcément le meilleur choix.

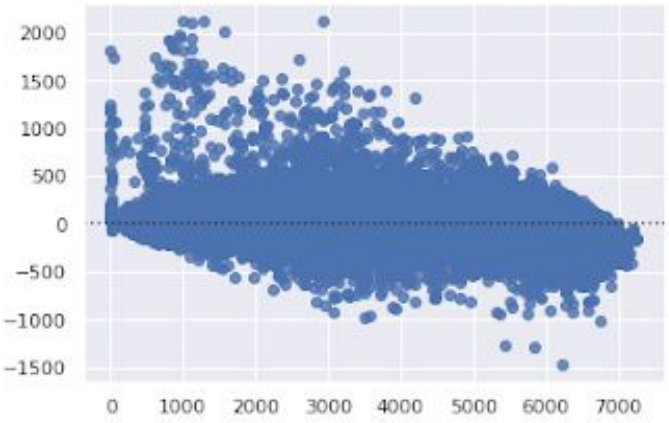
On remarque que le modèle de base (implémenté dans le programme squelette d'origine), a un des meilleurs scores et une variance très faible. Le modèle des KNeighbors semble quant à lui peu recommandé pour le remplacer .

Figure 2 : Fit et erreur résiduelle

Pour étudier la performance d'un modèle donné, nous avons affiché les données d'entraînement en fonction de la prédiction. La perfection serait la droite x=y représentée ci-contre.



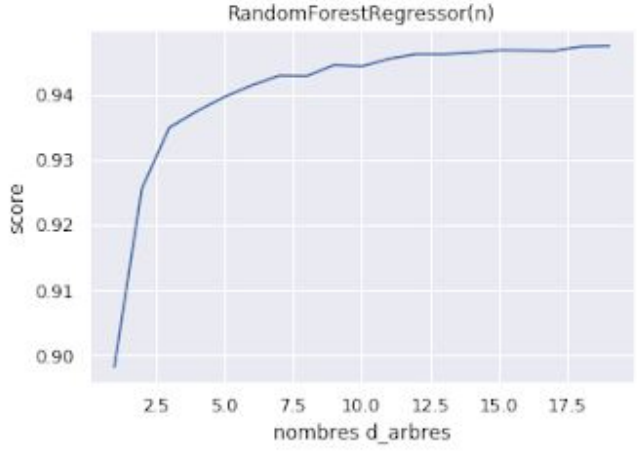
Voici le résultat obtenu pour le modèle de base. L'allure globale du schéma montre que la prédiction est cohérente, bien qu'elle ne soit pas parfaite.
On remarque que l'on peut également avoir une bonne visualisation de la densité de présence des données aux différentes positions, ce qui peut aider à comprendre lesquelles sont mal-interprétées.



Pour étudier d'un peu plus près cette performance, nous avons affiché l'erreur résiduelle du modèle (erreur par rapport à la droite).

Voici la position de chaque donnée dans un graphique (Ytrain,Ypredict) par rapport à la droite de prédiction. Cette visualisation sera bien utile pour juger de la qualité de chaque modèle (elle donne également une idée de la densité du positionnement des données)

Figure 3 : Score en fonction des hyper-paramètres :



Les hyper-paramètres allant probablement jouer un grand rôle dans le choix du modèle et l'amélioration du score, nous avons voulu être capable de représenter un score en fonction de l'évolution d'un hyper-paramètre.

Ici on remarque pour une forêt que le score augmente avec le nombre d'arbres, mais de façon logarithmique.

Code visualisation :

```
###affichage des performances###
from sklearn.tree import DecisionTreeRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import RandomForestRegressor

model_name = ["Base Estimator","Decision Tree Regressor","KNeighbors Regressor","Random Forest Regressor"]
model_list = [M,DecisionTreeRegressor(),KNeighborsRegressor(),RandomForestRegressor()]
scores_perf = []
scores_err = []

for i in range(len(model_list)) :
    sc=cross_val_score(model_list[i], X_train, Y_train, cv=5, scoring=make_scorer(scoring_function))
    scores_perf.append(sc.mean())
    scores_err.append(sc.std())

df = pd.DataFrame({'perf': scores_perf,'error': scores_err}, index=model_name)
ax=df.plot.bar(subplots=True,figsize=(8,8))
ax[0].set_ylabel("score")
ax[1].set_ylabel("score std")
```

```
###result visualization###
import seaborn as sns
import numpy as np
sns.set(style='darkgrid')
sns.jointplot(Y_train, Y_hat_train,kind="reg",height=9)
```

```
###residual error###
g = sns.residplot(Y_train,Y_hat_train)
```

```
###performance de RandomForestRegressor en fonction du nombre d'arbres###
S=[]
N=[]
s=make_scorer(scoring_function)
for n in range (1,20):
    sc=cross_val_score(RandomForestRegressor(n_estimators=n), X_train, Y_train, cv=5,scoring=s)
    S.append(sc.mean())
    N.append(n)
S=np.array(S)
N=np.array(N)
plt.plot(N,S)
plt.xlabel('nombres d_arbres')
plt.ylabel('score')
plt.title('RandomForestRegressor(n)')
```

Model :

Nous avons commencé par essayer plusieurs model. Pour cela nous avons modifié les class model et avons fait des copie. Nous les avons entraînées et avons calculé le score pour chacun. Cela a été une tâche plutôt laborieuse car certain model prenais très longtemps à être entraîné et nous ne pouvions qu'attendre.
En les comparant nous nous somme aperçu que GradientBoostingRegressor donnais les meilleur résultats.

```
# CV score (95 perc. CI): 0.94 (+/- 0.01)
class model7 (BaseEstimator):
    def __init__(self):
        '''
        This constructor is supposed to initialize data members.
        Use triple quotes for function documentation.
        '''
        self.num_train_samples=0
        self.num_feat=1
        self.num_labels=1
        self.is_trained=False
        self.mod = GradientBoostingRegressor(max_depth=9,random_state=0, n_estimators=100)# Initalizing the model
```

Nous avons ensuite cherché les meilleurs hyper-paramètres pour ce modèle avec les méthodes GridSearchCV et RandomizedSearchCV du module sklearn. Celles ci nous ont permis d'améliorer nos résultats en faisant passer le score de 92 à 94. Cependant encore une fois il était difficile de faire beaucoup d'essai étant donné le temps de calcul de ces méthodes.

```
from sklearn.model_selection import GridSearchCV
#impossible de comprendre le fonctionement des methodes
def grid(model,data_name= 'xporters',data_dir='./input_data/'):
    temps_a=time.time()
    D = DataManager(data_name, data_dir , replace_missing=True)
    M=model()
    X_train = D.data['X_train']
    Y_train = D.data['Y_train']
    #if not(M.is_trained) : M.fit(X_train, Y_train)
    #param = M.mod.get_params()
    #for key in param:
    #    param[key] = np.empty(0)
    #print(param)
    GSCV = GridSearchCV(M.mod, {"max_depth":[1,2,3,4,5,6,7,8,9],"random_state":[0], "n_estimators":[100]})
    GSCV.fit(X_train,Y_train)
    print(GSCV.best_params_)

def random(model,data_name= 'xporters',data_dir='./input_data/'):
    temps_a=time.time()
    D = DataManager(data_name, data_dir , replace_missing=True)
    M=model()
    X_train = D.data['X_train']
    Y_train = D.data['Y_train']
    #if not(M.is_trained) : M.fit(X_train, Y_train)
    param = M.mod.get_params()
    res = RandomizedSearchCV(M.mod, {"max_depth":[1,2,3,4,5,6,7,8,9],"random_state":[0], "n_estimators":[100]})
    res.fit(X_train,Y_train)
    print(res.best_params_)
```

Preprocessing :

Pour commencer nous avons rédigé une fonction qui supprimait les colonnes manuellement, le problème étant que sur un fichier comportant plus de 38000 lignes et 60 colonnes, c'était une fonction peu efficace donc nous nous sommes penchés plus longuement sur le sujet pour créer une fonction qui pourrait parcourir tout le fichier et identifier les features à enlever. Pour créer notre fonction nous avons importé la bibliothèque sklearn.feature_selection et utilise le plafond de variance. Par la suite ce nouveau tableau qui a déjà été modifié est utilisé en paramètre dans notre fonction. La fonction supprColonnes permet de parcourir les colonnes de data et si plus de 95% des valeurs de la colonne sont inférieures ou égales à 1, on supprime la colonne et on donne une copie de data en retour.

```
from sklearn.feature_selection import VarianceThreshold
selection = VarianceThreshold(threshold=(.8*(.2)))
Y = selection.fit_transform(data)

data.drop_duplicates()
data

def supprColonnes(Y,data):
    df = data
    to_drop = []
    for col in list(data):
        p = np.percentile(df[col],95)
        if p<=1:
            df = df.drop(columns=col)
    return df

S = supprColonnes(Y,data)
data = S
print(data)
```

Le résultat est le suivant, on a réduit le nombre de colonnes de 60 à 8.

```
[38563 rows x 8 columns]
```

Ensuite passons à la partie pour gérer les outliers. C'était la partie la plus compliquée à comprendre et à mettre en place. D'abord nous avons écrit la fonction find_extremes_outliers, qui regarde la feature f et qui identifie les valeurs extrêmes en utilisant des quartiles et des limites outliers, nous avons fait en sorte que toutes les valeurs qui sont dans les limites sont copiées dans indice_outliers et la fonction retourne l'ensemble des valeurs utilisables pour la fonction suivante qui supprimera les outliers identifiés sur les lignes.

```
def find_extreme_outliers(f):    # feature f
    q1 = np.percentile(f,25)
    q3 = np.percentile(f,75)
    ei = q3-q1
    Lmin = q1 - 1.5*ei
    Lmax = q3 + 1.5*ei
    indice_outliers = list(f.index[(f<Lmin)|(f>Lmax)])
    val_outliers = list(f[indice_outliers])
    return indice_outliers, val_outliers
```

La fonction suivante est la fonction suppr_outliers qui supprime les lignes en prenant le tableau data en argument et en retournant une liste update avec les valeurs correctes.

```
#suppression des outliers
def suppr_outliers(data):

    all_outliers_rows = set()

    for column in list(data.columns):
        print(column)

        outliers_for_feature_f, _ = find_extreme_outliers(data[column])
        all_outliers_rows.update(set(outliers_for_feature_f))

    indice_outliers = list(all_outliers_rows)

    print(data.shape)
    data_dropped = data.drop(index = indice_outliers) #inplace=True remplace directment datadrop dans data
    print(data_dropped.shape)
```

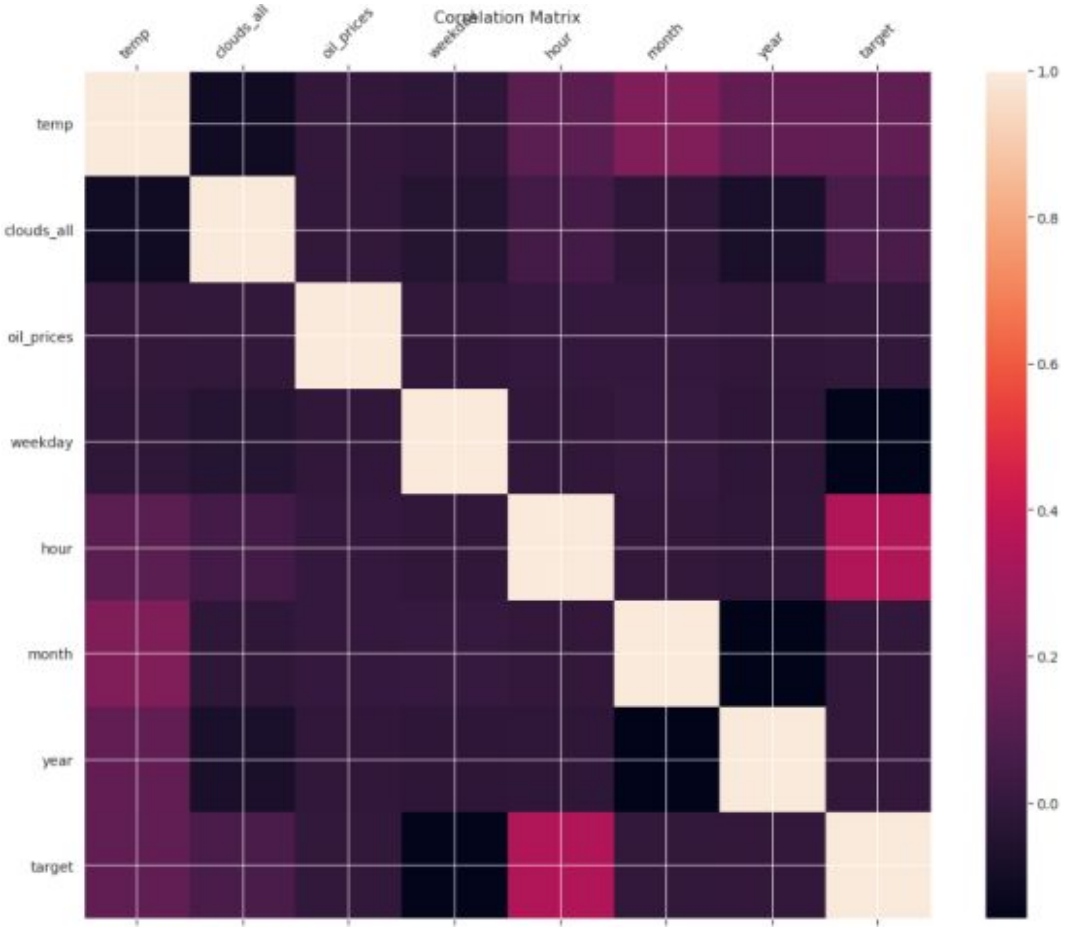
Par la suite, nous appelons la fonction et on affiche avant la taille du tableau data de base et data_dropped qui est la copie modifiée de data. Ainsi nous retirons 292 lignes de data.

```
suppr_outliers(data)
```

```
temp
clouds_all
oil_prices
weekday
hour
month
year
target
(38563, 8)
(38271, 8)
```

On obtient alors une matrice de corrélation beaucoup plus épurée avec les quelques données qui nous sont vraiment utile pour un preprocessing de meilleure qualité. Le but sera d'entraîner les modèles sur ses données afin d'obtenir le meilleur résultat possible. Bien sûr, un affinement des données comme un redimensionnage de ces dernières peut être mis en

place pour maximiser les chances d'obtenir le meilleur résultat possible.



Matrice de corrélation après suppression des outliers et des colonnes inutiles

Nous obtenons aussi un nouveau jeu de données des features les plus importantes:

```
Most important features according to the correlation with target
target         1.000000
hour           0.350545
temp           0.131803
clouds_all     0.064201
year           0.002271
month         -0.000533
oil_prices    -0.000706
weekday       -0.150780
Name: target, dtype: float64
```