

L2 MI - MINI PROJET

Challenge “Solve Xporters traffic volume problem” - Groupe Taxi

Team composé de 6 membres :

- Fanoa RAZAFIMBELO <fanoa.razafimbelo@u-psud.fr>
- Antonin PAOLI <antonin.paoli@u-psud.fr>
- Albanio DE SOUZA <albanio.desouza@u-psud.fr>
- Taïssir MARCE <taissir.marce@u-psud.fr>
- Hilmi CELAYIR <hilmi.celayir@u-psud.fr>
- Mohammad AKHLAGHI <mohammad.akhlaghi@u-psud.fr>



URL du challenge : <https://codalab.lri.fr/competitions/652>

Dossier Github du projet : https://github.com/Fanoa/Taxi/tree/master/starting_kit

URL de la vidéo : <https://www.youtube.com/watch?v=CoVWpsoXSOk>

Score Codalab : 0.9424156531

Description du défi et des données

Nous sommes un groupe de 6 personnes et nous avons décidé de travailler sur le projet Xporters. L'objectif du projet en tant que responsable d'un stand de limonade situé à côté d'une autoroute est de prédire le nombre de voitures qui passeront à une date, une heure, et des conditions météorologiques données ! L'ensemble de données contient 58 caractéristiques et la solution est le nombre de voitures (de 0 à 7280) en une heure. Ce projet est un problème de régression et pour évaluer les données nous utilisons la métrique R^2 [1]. Cette dernière mesure la qualité de la prédiction d'une régression linéaire, autrement dit elle évalue si la prédiction des données se rapprochent le plus possible de la réalité. Un résultat de 1 montre que chaque prédiction est similaire à la réalité et sinon plus elle se recule de la réalité et plus le résultat sera faible. (Il peut descendre au-dessous de 0)

Pour cela, nous avons divisé le projet en 3 parties :

- **Pré-processing** : retravailler les données brutes dans différents fichiers pour qu'elles soient exploitables par le modèle.
- **Modélisation** : trouver le meilleur modèle de régression pour l'exploitation des données ainsi que ses hyper-paramètres.
- **Visualisation** : afficher les résultats à l'aide de tableaux et graphes pour une meilleure compréhension du problème et des résultats.

Description des algorithmes étudiés

Le pré-processing et son code

La partie preprocessing consiste à préparer aux mieux nos données afin d'optimiser les performances de l'algorithme [2][15]. Nous avons dans un premier temps essayé de détecter les anomalies dans le jeu de données. Pour cela, nous avons utilisé l'algorithme Isolation Forest proposé par scikit-learn qui va faire un partitionnement récursif des données afin d'isoler les anomalies[14]. Pour cela chaque données du jeu est isolé, l'algorithme va calculer un score d'anomalie en la comparant aux autres afin de savoir à quel point elle est atypique, ainsi chaque donnée est isolé et évalué afin de savoir si il s'agit d'un outlier ou non.

Ensuite, nous avons réalisé une feature selection en enlevant du jeu de données les données qui ont une faible variance en définissant au préalable un seuil. Le seuil est défini par la formule $\text{Var}[X] = p(1 - p)$ où p représente la probabilité de la colonne X de contenir des zéro, la colonne est supprimé si la variance calculé est inférieur au seuil défini (voir fig. 1). Enfin, grâce au StandardScaler de la bibliothèque scikit learn[3], nous avons mis à l'échelle les données afin de les normaliser et les centrer en 0.

<pre>class preprocess : def __init__(self): self.reg = IsolationForest(max_samples=100) def fit_transform (self, X) : #Outliers Detection self.reg.fit(X) self.reg.predict(X) #Feature selection sel = VarianceThreshold(threshold=(.7 * (1 - .7))) X_train= sel.fit_transform(X) #scaling scaler = StandardScaler() scaled_X = scaler.fit_transform(X_train) return scaled_X</pre>	<p>Détection des outliers en utilisant Isolation Forest</p> <p>Feature selection sur les données en fonction de la variance(avec pour seuil 0.7)</p> <p>Centrage des données en 0 en fonction de la moyenne calculée</p>
---	---

Classe pré-processing du projet

La modélisation et son code

Un modèle de régression ayant été entraîné en amont permet de prédire le résultat d'une donnée. C'est pour cette raison que nous avons séparé les données en 2 ensembles : l'ensemble d'entraînement pour entraîner le modèle et l'ensemble d'évaluation pour tester les différents modèles et les comparer.

	Nombre d'exemples	Nombre de caractéristiques	Données manquantes ?
Ensemble d'entraînement	19281	58	Non
Ensemble de validation	19282	58	Non

Tableau sur les 2 ensembles de données

Pour rechercher le meilleur modèle nous avons testé chaque modèle sur les différents ensembles, puis nous avons reporté les différents scores dans le tableau suivant.

	CVScore	(+/-)	ScoreTraining	ScoreValidation	Overfitted	Underfitted
decisionTree	0.895753	0.0110091	1	0.893379	False	False
KNeighbors	0.714077	0.0169543	0.824117	0.738121	False	True
Linear	0.154124	0.0397991	0.159084	-0.86189	False	True
RandomForest	0.939665	0.013485	0.989563	0.939579	False	False
ExtraTrees	0.93402	0.013007	1	0.93184	False	False
Bagging	0.940094	0.0132516	0.989845	0.940764	False	False
GradientBoosting	0.923824	0.0139217	0.921651	0.916327	False	False
AdaBoost	0.826138	0.0217314	0.820622	0.820696	False	True
GaussianProcess	-2.36806	0.121442	1	-2.31089	True	False

Tableau de comparaison des différents modèles

Pour choisir le modèle, nous allons regarder le score sur l'ensemble de validation. Nous pouvons voir que le modèle Bagging() est le meilleur dans cette catégorie avec un CVScore élevé. Nous avons donc choisi d'utiliser ce modèle dans notre projet.

Pour information, le modèle Bagging() s'aide d'un autre modèle, appelé "BaseEstimator" pour faire la prédiction. Nous avons donc continué à faire quelques tests supplémentaires. Premièrement on s'est demandé ce qui était plus efficace entre le modèle RandomForest() et le modèle Bagging() ayant DecisionTree() comme BaseEstimator. Ces 2 choix proposent un fonctionnement presque identique pour l'entraînement et la prédiction des données. Puis deuxièmement, on s'est demandé si en remplaçant le BaseEstimator par un RandomForest(), nous arrivions à obtenir un score plus élevé.

Pour cela nous avons entraîné 10 fois, les différents modèles sur l'ensemble d'entraînement. Puis nous avons fait la moyenne des scores obtenus pour chaque ensemble de données et pour la Cross Validation.

Pour information lors de la soumission sur codalab, nous avons obtenu les résultats suivants, en cohérence avec les scores sur l'ensemble de validation :

- BaggingRegressor avec un DecisionTree comme BaseEstimator -> 0.9347583324
- BaggingRegressor avec un RandomForest comme BaseEstimator -> 0.9383864062
- RandomForest simple -> 0.9368342539

	CVScore	(+/-)	ScoreTraining	ScoreValidation
BaggingTree	0.93641	0.00830656	0.989072	0.940383
BaggingForest	0.942984	0.00960373	0.978857	0.946686
RandomForest	0.937124	0.0056703	0.988988	0.93964

Tableau de comparaison des 3 modèles

On peut constater que le bagging() avec un RandomForest() comme BaseEstimator est la solution la plus performante.

Enfin nous avons fini par utiliser la classe RandomizedSearchCV()[10] pour chercher les hyperparamètres du modèle retenu pour optimiser les performances. Pour information, cette classe permet de trouver les valeurs des différents hyperparamètres du modèle pour obtenir un score le plus élevé possible.

Cette dernière partie a été très longue pour nous ! Il faut savoir que la recherche prenait en compte les hyperparamètre du modèle, mais également ceux du BaseEstimator. Cela multiplie le temps de recherche car pour une certaine combinaison du BaseEstimator, il fallait tester toutes celles du Bagging(). Nous avons donc dû couper la recherche en deux. Le première avec le nombre d'arbres d'estimation puis une deuxième avec tous les autres. En sachant que même cette séparation à du demander plusieurs jours pour trouver la combinaison la plus performante. Cependant ce choix de séparation n'est pas complètement judicieux, car il ne permet pas d'optimiser au maximum la recherche.

<pre> model_dict = dict(decisionTree=DecisionTreeRegressor(), KNeighbors=KNeighborsRegressor(), Linear=LinearRegression(), RandomForest=RandomForestRegressor(), ExtraTrees=ExtraTreesRegressor(), Bagging=BaggingRegressor(), GradientBoosting=GradientBoostingRegressor(), AdaBoost=AdaBoostRegressor(), GaussianProcess=GaussianProcessRegressor()) def analyze_model_experiments(tabResult): tebad = tabResult.ScoreValidation < tabResult.ScoreValidation.median() trbad = tabResult.ScoreTraining < tabResult.ScoreTraining.median() overfitted = tebad & ~trbad underfitted = tebad & trbad tabResult['Overfitted'] = overfitted tabResult['Underfitted'] = underfitted return tabResult.style.apply(highlight_above_median) </pre>	<p>Dictionnaire qui rassemble les différents modèles de régression que l'on veut comparer</p> <p>classe pour détecter l'overfitting et l'underfitting</p> <p>Si le score de validation est au dessous de la médiane et que le score d'entraînement est au dessus de la médiane alors on est dans un cas d'overfitting</p> <p>Si le score de validation est au dessous de la médiane et idem pour le score d'entraînement alors on est dans un cas d'underfitting</p>
---	--

<pre> def highlight_above_median(s): """Highlight values in a series above their median. """ medval = s.median() return ['background-color: yellow' if v>medval else " for v in s] def performance(X_train, Y_train, X_valid, Y_valid, model_dict): """Run cross-validation on a bunch of models and collect the results.""" tabResult = pd.DataFrame(columns=["CVScore", "(/-)", "ScoreTraining", "ScoreValidation"]) for modelName, model in model_dict.items(): CVScore = 0 trainScore = 0 validScore = 0 i = 0 while i < 10: model.fit(X_train, Y_train) CVScore += cross_val_score(model, X_train, Y_train, cv=5, scoring=make_scorer(scoring_function)) Y_hat_train = model.predict(X_train) Y_hat_valid = model.predict(X_valid) trainScore += scoring_function(Y_train, Y_hat_train) validScore += scoring_function(Y_valid, Y_hat_valid) i = i + 1 tabResult.loc[modelName] = np.array([CVScore.mean(), CVScore.std() * 2, trainScore, validScore]) return tabResult compar_results = performance(X_train, Y_train, X_valid, Y_valid, model_dict) compar_results.round(6).style.background_gradient(cmap='Bl ues') best_model = compar_results.ScoreValidation.idxmax() print("Le meilleur modèle est : {}".format(best_model)) analyze_model_experiments(compar_results) bagging = BaggingRegressor() n_samples = len(X_train) distributions = dict(n_estimators=[300, 400], bootstrap=[True, False], bootstrap_features=[True, False], random_state=[50, 150], n_jobs=[5, 10]) reg = RandomizedSearchCV(bagging, distributions) search = reg.fit(X_train, Y_train) print("Score final : ", round(search.score(X_train, Y_train) *100, 4), " %") print("Meilleurs parametres : ", search.best_params_) print("Meilleure configuration : ", search.best_estimator_) </pre>	<p>Cette fonction permet de surligner en jaune les choses importantes du tableau, autrement dit ce qui est au dessus de la médiane, que ce soit des chiffres ou des booléens</p> <p>Cette fonction va calculer les différents scores.</p> <p>On crée un tableau avec la librairie panda Pour chaque modèle énoncé dans le dictionnaire : - on initialise les différentes variables</p> <p>- on boucle cette partie 10 fois - l'entraîner avec l'ensemble train - calculer le score de validation croisée - prédire avec l'ensemble train - prédire avec l'ensemble valid - calculer le score sur train - calculer le score sur valid</p> <p>- On incrémente le compteur de 1 - rajouter les infos dans le tableau</p> <p>On appelle la fonction performance</p> <p>On arrondis à 6 chiffres après la virgule dans le tableau (le style d'affichage ne sera pas gardé) On affiche le modèle avec le meilleur score sur l'ensemble de validation On affiche le tableau avec l'analyse et le surlignement jaune</p> <p>Une fois le modèle choisi, On crée un autre dictionnaire avec les différents hyperparamètres et pour chacun d'eux, une liste de valeurs possibles.</p> <p>On appelle la fonction de recherche On entraîne le modèle avec les hyperparamètres trouvés On affiche le score en pourcentage On affiche les valeurs des hyperparamètres les plus efficace On affiche la configuration complète du modèle</p>
--	---

Code recherche du meilleur modèle

La visualisation et ses figures

Dans cette partie nous avons créé des figures pour la visualisation des clusters dans nos données[11]. Les figures sont choisies en fonction des données que l'on affiche de sorte à les rendre le plus expressif et compréhensif possible. La plupart des graphiques ont d'abord été testés sur des données d'essais de petite taille afin qu'ils soient en accord avec nos attentes avant d'être utilisés dans le fichier README.

<pre>import numpy as np #Figure 1 x1 = np.linspace(1,Y_train.size,Y_train.size) y1 = np.array(Y_train) plt.figure(figsize = (15,8)) plt.scatter(x1,y1,s = 2,c='green',marker = '+') plt.title('Y_train') plt.ylabel('number of cars in 1 hour') #Figure 2 x2 = np.linspace(1,X_train[1].size,X_train[1].size) y2 = np.array(X_train) plt.figure(figsize = (15,9)) for i in range (y2[1].size): plt.scatter(x2,y2[:,i],s = 70,c='blue',marker = '+') plt.yscale('log') plt.ylim(1,500) plt.xlim(0,58) plt.title('X_train') plt.xticks(range(data.shape[1]), data.columns, fontsize=14, rotation=90) plt.ylabel('value') plt.xlabel('features') #Figure 3 x3 = np.linspace(1,X_valid[1].size,X_valid[1].size) y3 = np.array(X_valid) plt.figure(figsize = (15,9)) for i in range (y3[1].size): plt.scatter(x3,y3[:,i],s = 70,c='coral',marker = '+') plt.title('X_valid') plt.yscale('log') plt.ylim(1,400) plt.xlim(0,30) plt.xticks(range(data.shape[1]), data.columns, fontsize=14, rotation=90) plt.ylabel('value') plt.xlabel('features')</pre>	<p>Importation de la bibliothèque numpy en x, on aura simplement une entendu des différents résultats en y, on aura le nombre de voiture en 1 heure On trace la figure Paramètres de la figure On ajoute des titres et des labels sur la figure</p> <p>En x, on aura les 58 features En y, on aura les différentes valeurs de ces features On trace la figure</p> <p>Paramètres de la figure On ajoute une échelle logarithmique Ainsi que les limites de la figure</p> <p>On ajoute les différents labels et titres</p> <p>En x, on aura les 58 features En y, on aura les différentes valeurs de ces features On trace la figure</p> <p>Paramètres de la figure On ajoute une échelle logarithmique Ainsi que les limites de la figure</p> <p>On ajoute les différents labels et titres</p>
---	---

Code création des figures des données

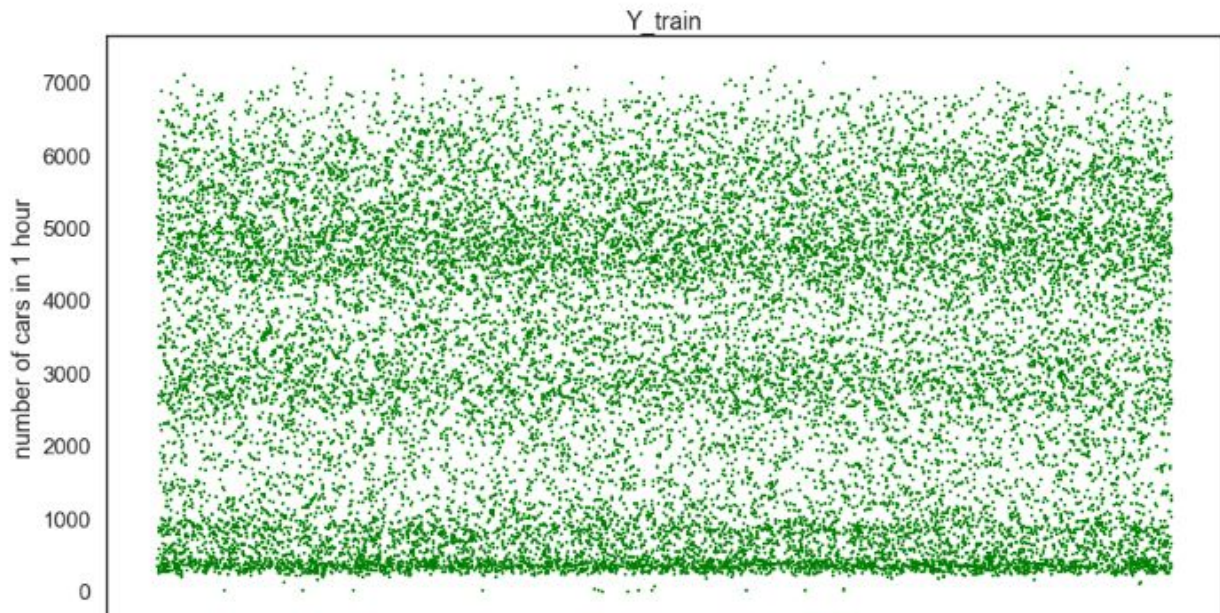
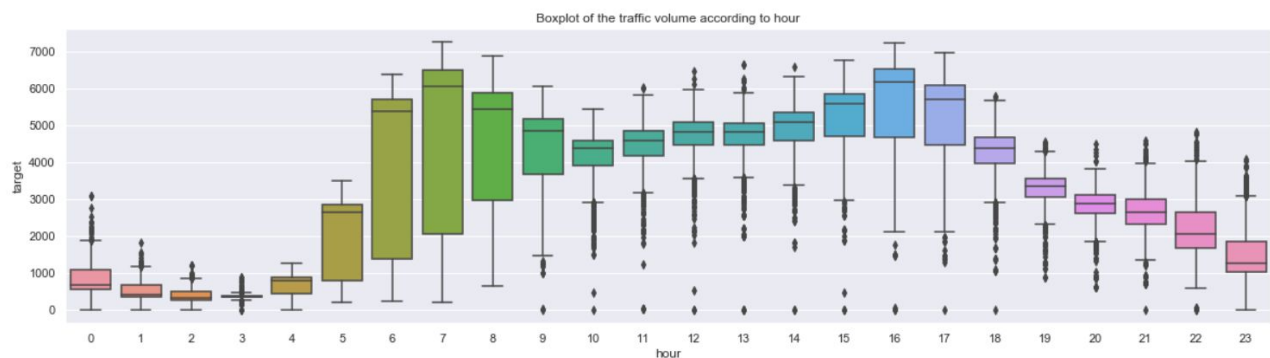


Figure 1 : Nombre de voitures en 1 heure

Dans cette figure, l'axe X représente simplement les 10281 données de l'ensemble d'entraînement. Autrement dit, pour chaque donnée, une croix verte représente le nombre de voitures présent sur l'autoroute en 1 heure. Plus la zone des croix vertes est dense, plus on peut en déduire que l'intervalle des nombres de voitures correspondante aura une probabilité beaucoup plus élevée d'être le résultat d'une donnée. Quelque soit les valeurs des caractéristiques, on constate que le résultat à plus de chance de se trouver vers 400, 3000 ou 4800. Il est complémentaire à la figure suivante.



Le volume du trafic en fonction de l'heure

Pour chacune des caractéristiques, la figure nous montre les différentes valeurs des données de l'ensemble d'entraînement. Ainsi nous pouvons déduire des plages de valeurs plus précise.

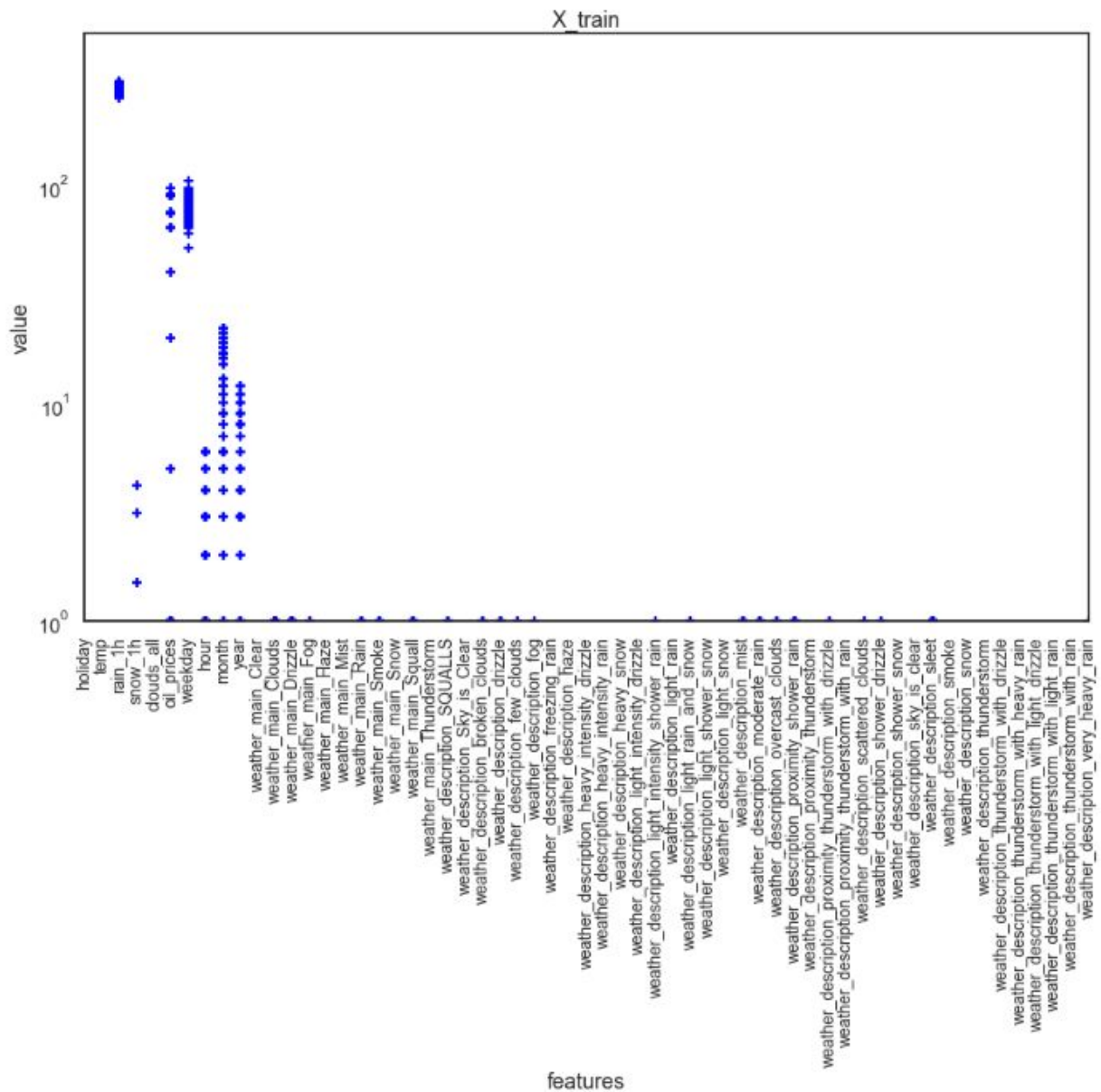


Figure 2 : les valeurs des différentes caractéristiques sur l'ensemble d'entraînement

Pour chacune des caractéristiques, la figure nous montre les différentes valeurs des données de l'ensemble de validation. Ainsi nous pouvons déduire des plages de valeurs plus précise.

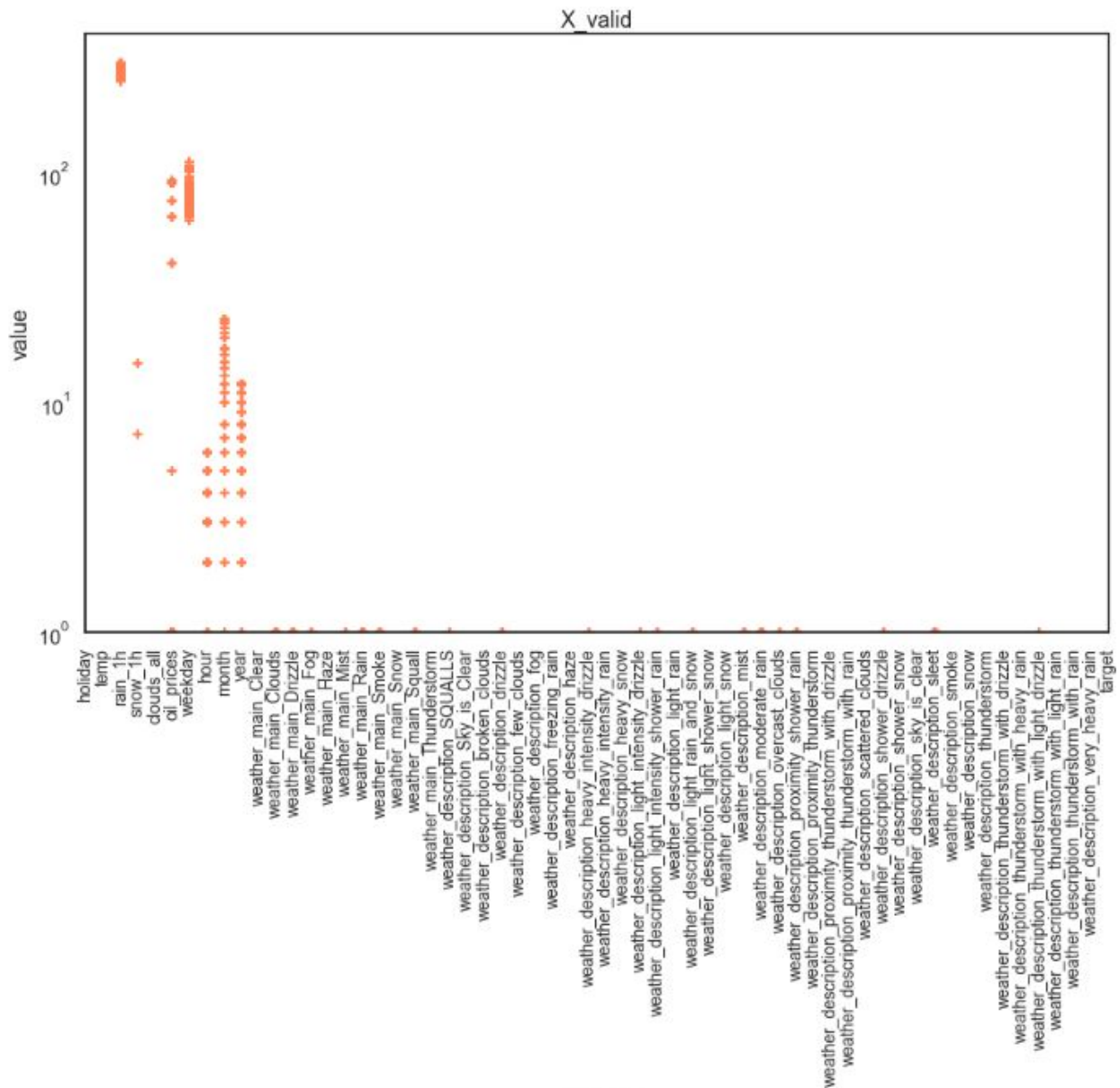


Figure 2 : les valeurs des différentes caractéristiques sur l'ensemble de validation

Tests Unitaires et Problèmes rencontrés

<pre>def testFeatureS (X): sel = VarianceThreshold(threshold=(.7 * (1 - .7))) return sel.fit_transform(X) T = [[0, 0, 1], [0, 1, 0], [1, 0, 0], [0, 1, 1]] testFeatureS(T) def testPrep (X): print("Avant preprocessing : ", X) N_X = fit_transform2(X) print("Après preprocessing : ", N_X)</pre>	<p>On teste le fonctionnement de la feature selection à partir de l'exemple de la fig.2 On définit un tableau T représentant les valeurs de base, puis en appliquant la fonction sur ce tableau on retrouve bien le résultat trouvé dans la fig. 2 : >> [[0, 1], [1, 0], [0, 0], [1, 1]]</p> <p>Test permettant de comparer les données de bases avant et après le preprocessing</p>
--	--

Test unitaire pour le preprocessing

<pre>def test() : mod = modelRegressor(DecisionTreeRegressor(criterion='friedman_m se', max_depth=100, random_state=150, splitter='random')) X_random = np.random.rand(1,59) Y_random = np.array([np.random.randint(200, 850)]) mod.fit(X_random, Y_random) Y_random_predict = mod.predict(X_random) if Y_random_predict == Y_random : print("Test Réussi") else : print("Test Echoué")</pre>	<ul style="list-style-type: none">- On load le modèle- On crée un tableau à 2 dimensions avec des valeurs entre 0 et 1 pour représenter une donnée (les valeurs importent peu, c'est le principe qui doit fonctionner)- On crée un tableau à 2 dimensions avec la réponse à cette donnée- On entraîne le modèle- On établit une prédiction- Si le résultat réel est identique au résultat prédit alors le test est réussi- Sinon le test est échoué
---	---

Test unitaire pour le modèle

Pour le pré-processing, on a vu qu'en appliquant l'algorithme du PCA sur le jeu de données, le score devenait faible voir négatif même en changeant le paramètre de `n_components` de 6 à 11. Nous avons donc à la place utilisé le Standard scaler de la bibliothèque scikit-learn[3] afin de mettre à l'échelle les données et les centrer en zéro.

Pour le modèle, notre plus grande difficulté a été de trouver les bon hyper paramètres pour le modèle. Nous avons utiliser comme dit précédemment des fonctions mais il fallait aussi bien réfléchir aux méthodes d'apprentissage qu'utilisait notre modèle, autrement dit, fallait-il utiliser le bootstrap* ou non ? Fallait-il limiter le nombre de features, alors que le preprocessing en éliminait quelques uns ? Et bien d'autres. (*le bootstrap est un technique qui permet de créer des sous-ensemble de manière aléatoire avec remise dans l'ensemble initial.)

Si on devait faire passer un message aux futurs étudiants ; pour la partie modèle en tout cas ; Il faut bien comprendre les modèles que vous utiliserez et ne pas hésiter à faire des recherches la dessus, car c'est comme ça que vous arriverez à trouver une solution sur son amélioration. Les explications de notre chargé de groupe nous a beaucoup aidé à comprendre le modèle que nous utilisons pour pouvoir l'améliorer.

Références

- [1]Wikipedia Coefficient de determination https://en.wikipedia.org/wiki/Coefficient_of_determination
- [2]Scikit-learn 6.3 Preprocessing Data [6.3. Preprocessing data — scikit-learn 0.22.2 documentation](#)
- [3]Scikit-learn StandardScaler [sklearn.preprocessing.StandardScaler — scikit-learn 0.22.2 documentation](#)
- [4]Sklearn.tree.DecisionTreeRegressor [DecisionTreeRegressor](#)
- [5]Sklearn.neighbors.KNeighborsRegressor [KNeighborsRegressor](#)
- [6]Sklearn.ensemble.RandomForestRegressor [RandomForestRegressor](#)
- [7]Sklearn.ensemble.ExtraTreesRegressor [ExtraTreesRegressor](#)
- [8]Sklearn.ensemble.BaggingRegressor [BaggingRegressor](#)
- [9]Sklearn.ensemble.GradientBoostingRegressor [GradientBoostingRegressor](#)
- [10]sklearn.model_selection.RandomizedSearchCV [RandomizedSearchCV](#)
- [11]Scikit-learn 2.3 Clustering [2.3. Clustering — scikit-learn 0.22.2 documentation](#)
- [12]Scikit-learn 1.13. Feature selection [1.13. Feature selection — scikit-learn 0.22.2 documentation](#)
- [13]Overfitting et Underfitting [overfitting-and-underfitting-with-machine-learning-algorithms](#)
- [14]Scikit-learn 2.7 Novelty and outlier detection [2.7. Novelty and Outlier Detection — scikit-learn 0.22.2 documentation](#)
- [15]H. Yang, *Data Mining: Concepts and Techniques* [Chapter 3 : Data Preprocessing](#)