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

Contexte et description du problème :

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 météorologique données ! L'ensemble de données contient 58 caractéristiques et la solution est le nombre de voitures (de 0 à 7280) en une heure.

Ce choix de projet provient de raisons bien différentes : pour certains c’est le contexte du projet qui les attiraient, pour d’autres c’est la façon dont sont organisés les données. En effet, pour ceux de la première catégorie le sujet se rapprochait d’un domaine particulier dans quoi ils avaient une passion ou bien parce que la présentation du projet les a convaincu. Pour ceux de la seconde catégorie c’est simplement accepter le défi de travailler sur la régression, chose que nous n’avons pas eu le temps d'aborder au semestre 3.

Comme dit précédemment, 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 dire 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 reculera de la réalité et plus le résultat sera faible. (Il peut descendre au dessus de 0)

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

- **Pré-processing :** retravailler les données brutes données dans différents fichiers pour qu’ils soient exploitable 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.

Définition des différentes parties :

Partie 1 : Pre-processing (Fanoa et Taïssir)

La partie preprocessing consiste à préparer aux mieux nos données afin d'optimiser les performances de l'algorithme.[2]
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]. Puis, nous avons réalisé une feature selection en enlevant du jeu de données les données qui ont une faible variance.Enfin, grâce au StandardScaler de la bibliothèque scikit learn[3], nous avons mis à l'échelle les données afin de les centrer en 0.

Plus de détails sur le code peut être retrouvé dans le fichier README_preprocessing.ipynb dans notre Github Taxi.

Partie 2 : Modèle (Hilmi et Antonin)

Pour commencer, 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. Ensuite nous avons créé une liste de 9 modèles de régression. Explications des plus importants :

- DecisionTreeRegressor() : c'est une méthode d'apprentissage supervisé. L'objectif est de créer un modèle qui prédit la valeur d'une variable cible en apprenant des règles de décision simples déduites des caractéristiques des données. [4]
- KNeighborsRegressor() : La régression est basée sur les voisins où l'étiquette attribuée à un point d'interrogation est calculée sur la base de la moyenne des étiquettes de ses voisins les plus proches. [5]
- RandomForestRegressor() : il ajuste un certain nombre d'arbres de décision sur divers sous-échantillons de l'ensemble de données et utilise la moyenne pour améliorer la précision prédictive et contrôler le surajustement. [6]
- ExtraTreesRegressor() : il met en œuvre un méta estimateur qui ajuste un certain nombre d'arbres de décision sur divers sous-échantillons de l'ensemble de données et utilise la moyenne pour améliorer la précision prédictive et contrôler l'ajustement excessif. [7]
- BaggingRegressor() : c'est un méta-estimateur d'ensemble qui ajuste les régresseurs de base chacun sur des sous-ensembles aléatoires de l'ensemble de données d'origine, puis agrège leurs prédictions individuelles pour former une prédiction finale. [8]
- GradientBoostingRegressor() : il permet l'optimisation de fonctions de perte arbitrairement différenciables. À chaque étape, un arbre de régression est ajusté sur le gradient négatif de la fonction de perte donnée. [9]

Pour chacun d'entre eux, nous allons les entraîner sur l'ensemble d'entraînement puis nous calculons le score sur l'ensemble de validation. Avec ces résultats nous prendrons le modèle qui a obtenu le meilleur score. Ensuite grâce à la classe RandomizedSearchCV()[10] nous allons 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.

Plus de détails sur le code peut être retrouvé dans le fichier README_Model.ipynb dans notre Github Taxi.

Partie 3 : Visualisation (Albanio et Mohammad)

Dans cette partie nous avons créé des figures pour la visualisation des clusters dans nos données[11]. Puis nous avons créé des graphes pour étudier l'erreur de prédiction sur le régresseur trouvé par l'équipe modèle. Et enfin nous avons tracé les performances du modèle d'apprentissage sur le jeu de données.

Plus de détails sur le code peut être retrouvé dans le fichier README_vizualisation.ipynb dans notre Github Taxi.

Résultats et Figures :

Figure 1 : Principe du feature selection sur un jeu de données

0	0	1
0	1	0
1	0	0
0	1	1

Pour effectuer une feature selection de nos données, nous calculons la variance de chaque colonne du tableau du jeu données à l'aide de la formule Var[X] = p(1 - p) avec p la probabilité de la colonne X de contenir des 0. Si la probabilité p est supérieur aux seuil défini : la colonne est supprimée. Nous avons pris arbitrairement comme seuil p = 0.7.
Dans le cas ci-dessus, par exemple, la première colonne a une probabilité ¾ de contenir des 0, ce qui est au dessus du seuil choisi. Notre algorithme va donc supprimé la colonne en question.[12](voir test 1 plus bas pour l'illustration en code python)

Figure 2 : Statistiques sur les différents ensembles de données

	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

A l'aide de la fonction split de python, nous avons séparé les données du projet en 2 ensembles : ensemble d'entraînement et ensemble de validation. Cela nous servira pour tester les différents modèles.

Explication des 59 différentes caractéristiques :

- holiday (1) : valeur booléenne indiquant si l'heure se situe dans une période de vacances
- temp (2) : Température moyenne en kelvin
- rain_1h (3) : Quantité en mm de pluie tombée dans l'heure
- snow_1h (4) : Quantité en mm de neige tombée dans l'heure
- clouds_all (5) : Pourcentage de la couverture nuageuse
- oil_prices (6) : prix du baril de pétrole en dollars
- weekday, hour, month, year (7-10) : date
- weather_main_* (11-21) : Description globale de la météo
- weather_descrption_* (22-59) : Description détaillée de la météo

	holiday	temp	rain_1h	snow_1h	clouds_all	oil_prices	weekday	hour	month	year
count	38563.000000	38563.000000	38563.000000	38563.000000	38563.000000	38563.000000	38563.000000	38563.000000	38563.000000	38563.000000
mean	0.001245	281.197804	0.379081	0.000203	49.350284	80.079942	2.984311	11.408578	6.518009	2015.510645
std	0.035259	13.239935	50.073028	0.007602	39.029958	9.992938	2.003339	6.947282	3.405988	1.892133
min	0.000000	0.000000	0.000000	0.000000	0.000000	38.724760	0.000000	0.000000	1.000000	2012.000000
25%	0.000000	272.160000	0.000000	0.000000	1.000000	73.343967	1.000000	5.000000	4.000000	2014.000000
50%	0.000000	282.341000	0.000000	0.000000	64.000000	80.134711	3.000000	11.000000	7.000000	2016.000000
75%	0.000000	291.790000	0.000000	0.000000	90.000000	86.771668	5.000000	17.000000	9.000000	2017.000000
max	1.000000	310.070000	9831.300000	0.510000	100.000000	128.465356	6.000000	23.000000	12.000000	2018.000000

Tableau de statistiques sur les 10 premières caractéristiques

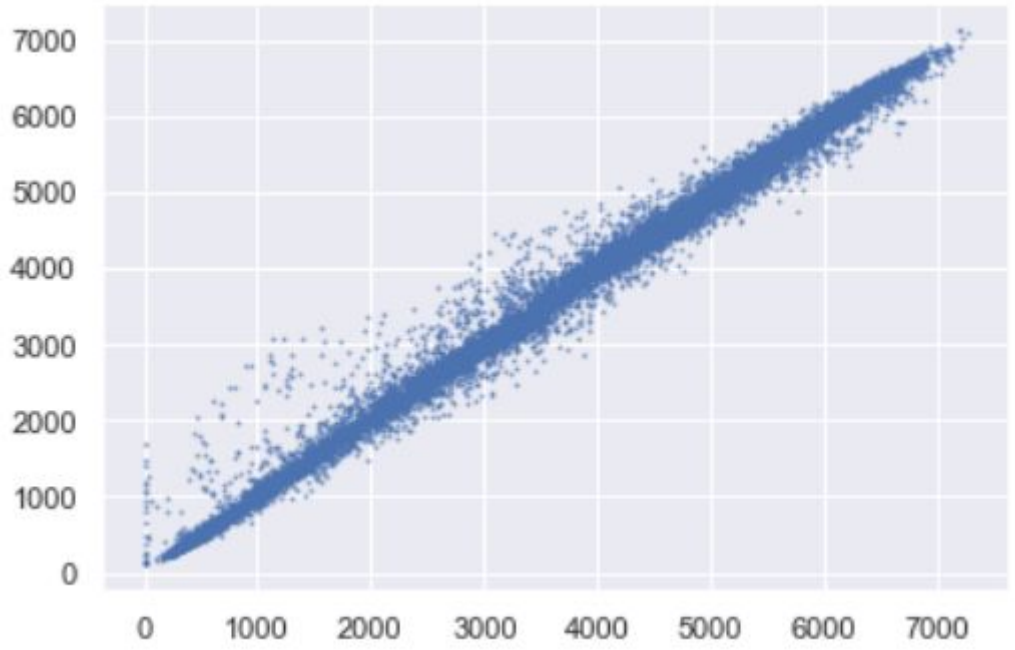
Figure 3 : Statistiques des différents modèles

	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, de plus il a également un CVScore élevé ce qui nous a permis de le choisir. Pour information, les modèles qui utilisent des arbres de décisions ont des score sur l'ensemble d'entraînement à 1, ce qui est excellent, mais on voit par la suite que le score sur l'ensemble de validation est un peu plus faible. Ca se comprend par le fait que les arbres de décision ont une prédiction excellente quand ils ont déjà eu à traiter une certaine donnée mais on plus de mal quand les données sont entièrement nouvelles. Enfin Les deux dernières colonnes permettent de déterminer les modèles qui sont dans le sur-apprentissage ou le sous-apprentissage.

Figure 4 : Nombre de voitures sur l'autoroute pour l'ensemble des données



Graphique (résultats réels * résultats prédits)

Ce graphique montre la précision de notre modèle. Plus les points bleus forment une diagonale parfaite, plus le modèle est performant et son score se rapprochera de 1 et inversement. Pour informations, pour chaque données de l'ensemble d'entraînement, un point va se placer en fonction de son résultat réel (X) et de son résultat prédit (Y).

Code et Tests Unitaire :

Code 1 : Le preprocessing des données

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
	Détection des outliers en utilisant Isolation Forest
	
	Feature selection sur les données en fonction de la variance(avec pour seuil 0.7)
	
	Centrage des données en 0 en fonction de la moyenne calculée

Code 2 : trouver le meilleur modèle

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)
	Dictionnaire qui rassemble les différents modèles de régression que l'on veut comparer
	
	classe pour détecter l'overfit et l'underfit
	
	Si le score de validation est au dessous de la moyenne et que le score d'entraînement est au dessus de la moyenne alors on est dans un cas d'overfitting
	
	Si le score de validation est au dessous de la moyenne et idem pour le score d'entraînement alors on est dans un cas d'underfitting

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]	Cette fonction permet de surligner en jaune les choses importantes du tableau, autrement dit ce qui est au dessus de la moyenne, que ce soit des chiffres ou des booléens

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():
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)
tabResult.loc[modelName] = np.array([CVScore.mean(), CVScore.std() * 2, trainScore, validScore])
return tabResult
	Cette fonction va calculer les différents scores.
	
	On crée un tableau avec la librairie panda
	Pour chaque modèle énoncé dans le dictionnaire :
	- 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
	- rajouter les infos dans le tableau

compar_results = performance(X_train, Y_train, X_valid, Y_valid, model_dict) compar_results.round(6).style.background_gradient(cmap='Blues') best_model = compar_results.ScoreValidation.idxmax() print("Le meilleur modèle est : {}".format(best_model)) analyze_model_experiments(compar_results)	On appelle la fonction performance 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
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_)	Une fois le modèle choisi, On crée un autre dictionnaire avec les différents hyperparamètres et pour chacun d'eux, une liste de valeurs possibles. 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

Test Unitaire :

Test 1 : test unitaire pour le preprocessing

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)
	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]]
	
	Test permettant de comparer les données de bases avant et après le preprocessing

Test 2 : test unitaire pour le modèle

def test() :
  mod = modelRegressor(DecisionTreeRegressor(criterion='friedman_mse', 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")
	- 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

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