MOUSSA AIAD

DATA CHALLENGE

PREDICT THE CRUDE OIL PRODUCTION'S TREND BASE ON THE PREVIOUS YEAR CRUDE OIL DATA.

MASTER 2 STATISTIQUE A L'UPMC

Plan

INTRODUCTION

Ce rapport rentre dans le cadre de mon Master 2 mathématique spécialité statistique à l'UPMC pour le cours de modèle linéaire. Le défi majeur de ce concours « data challenge » proposait par la société générale que j'ai participé, consiste à prédire la probabilité d'augmentation ou diminution de pétrole à partir d'une base de données. Cette base de données est composée de trois fichiers : fichier test, fichier train et le fichier Targuet présent dans la plateforme. On comprend bien que la séparation a été faite dans ce travail. Pour mener ce travail, j'ai utilisé PYTHON comme langage de programmation. La problématique de ce défi rentre dans le cadre l'apprentissage supervisé. Ainsi plusieurs méthodes ont été utilisées que nous allons voir plus en détail dans ce rapport, afin d'améliorer la précision de notre modèle. Donc la manière dont le traitement de données a été effectué sera abordée ainsi que les différentes méthodes utilisées pour améliorer la précision du modèle.

1-Traitement des données

-Étude univariée

Après l'extraction des données, une data visualisation a été directement effectuée pour comprendre les comportements de nos variables ainsi que leurs natures (variable catégorielle ou continue).

```
d=pd.read_csv('challenge_fichier_de_sortie_dentrainement_predire_la_tendance_de_la_production_de_petrole_brut.csv',sep=";")
dat_train=pd.read_csv('Train.csv',sep=";")
dat_test=pd.read_csv('Test.csv',sep=";")
y=d['Target']
train=dat_train.copy()
train=train.join(y)
ID=dat_test['ID']
```

Ainsi en utilisant la fonction **info()** pour voir la nature des variables, nous avons :

```
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10159 entries, 0 to 10158
Data columns (total 9 columns):
1_diffClosing stocks(kmt)          9898 non-null float64
1_diffExports(kmt)                 10159 non-null float64
1_diffImports(kmt)                 10008 non-null float64
1_diffRefinery intake(kmt)         10159 non-null float64
1_diffWTI                          10159 non-null float64
1_diffSumClosing stocks(kmt)       10159 non-null float64
1_diffSumExports(kmt)              10159 non-null float64
1_diffSumImports(kmt)              10159 non-null float64
1_diffSumRefinery intake(kmt)      10159 non-null float64
dtypes: float64(9)
memory usage: 714.4 KB
```

la base de données est composée des variables de types **int()** et **floats()**. Les variables de type **int()** sont catégorielles et ce n'est pas parce que les variables **month** et **country** sont de type **int()** qu'ils ne sont pas obligatoirement catégorielles. Les autres variables sont continues de types **floats()**.

L'étude univariée des variables permet de comprendre le comportement des variables dans le temps, ce qui nous donnera une idée sur l'augmentation ou diminution de la production de pétrole.

Ce flux d'augmentation ou de diminution peut influencer les prix de pétrole. Deux scénarios peuvent être envisageables :

- Baisse de la production entraîne une augmentation du prix par les compagnies vendeurs
- Baisse de la production entraîne une diminution du prix par les compagnies vendeurs

Ces scenarios ainsi envisagés nous amène à utiliser la loi de probabilité afin de suivre ces variables.

Concernant l'imputation des valeurs manquantes, l'étude univariée nous permettra aussi de voir laquelle de ces différentes méthodes sera la plus adaptée à notre problème.

-Étude bivariée

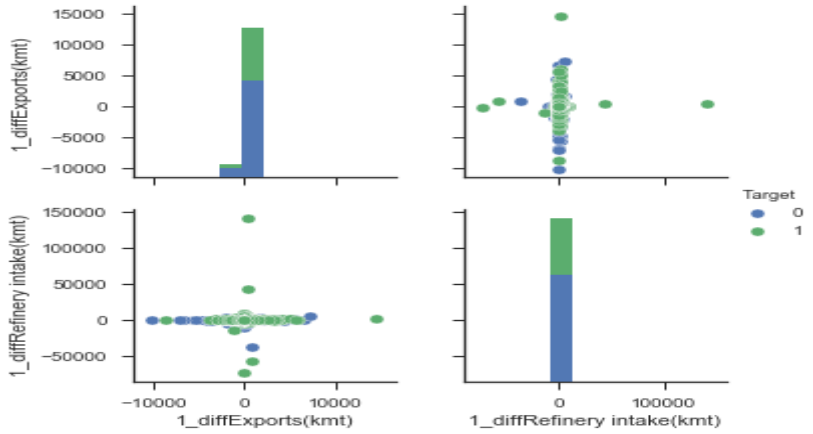
Pour mieux comprendre les relations existantes entre les différentes variables, l'étude bivariée était indispensable. Pour cette étape, j'ai commencé par utiliser la fonction **heatmap()** de la bibliothèque **Seaborn** pour représenter la matrice de corrélation de nos variables. Mais comme la taille est trop grande ; c'était difficile de distinguer les couleurs simples des couleurs vives. Une case dont la couleur est vive, nous permet d'affirmer que les variables sont très corrélées et dans ce cas on se

pose la question à savoir que ferons-nous des variables très corrélées ? Et à partir de quelle valeur on parlera d'une forte corrélation ?

Comme les variables corrélées ramènent une double information dans notre jeux de données ayant une taille très grande, on a décidé de supprimer les variables très corrélées (coefficient de corrélation >=0.8) afin d'éviter le phénomène de sur-apprentissage. On donnera plus de détail dans la partie « sélection de variables ». La fonction **heatmap()** nous a permis d'avoir une idée de la relation qui lie les variables entre elles. Après avoir utilisé cette fonction **heatmap()**, j'ai utilisé la fonction **sns.pairplot()** de la bibliothèque **seaborn.** Cette fonction ressemble à celle de **pairs()** du **langage R**. En fin on s'est servi de la fonction **sns.jointplot()** de la bibliothèque **seaborn** pour afficher une variable une contre l'autre. Après je me suis concentré sur la relation qui peut exister entre nos différentes variables. Voici un extrait de sortie de la fonction **sns.pairplot()** et de la fonction **sns.pairplot().**

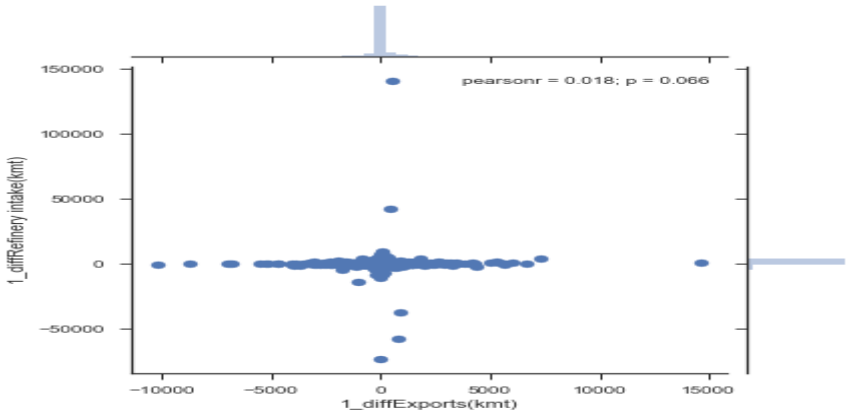
```
sns.set(style="ticks")
sns.pairplot(data = train, vars = ['1_diffExports(kmt)'
    , '1_diffRefinery intake(kmt)'],hue="Target")
```

```
<seaborn.axisgrid.PairGrid at 0x18b148d9c18>
```



La figure ci-dessus nous montre que les deux variables sont faiblement corrélées entre elles parce que l'ensemble des observations ne sont pas alignées dans la première bissectrice. Pour connaître leur coefficient de corrélation, nous allons utiliser la fonction **sns,jointplot()** de la bibliothèque **seaborn.** Sur la figure ci-dessous, le coefficient de corrélation et la p-valeur permettant de pouvoir répondre à la question selon laquelle les variables sont ou (ne sont pas) fortement corrélées. On voit que le coefficient de corrélation vaut 0,018 avec un p-valeur 0,066 ce qui confirme bien notre hypothèse selon laquelle les deux variables sont faiblement corrélées.

```
sns.jointplot(x="1_diffExports(kmt)",
              y="1_diffRefinery intake(kmt)",
              data=train)
<seaborn.axisgrid.JointGrid at 0x2778b2ddb00>
```



2- Traitement de données manquantes :

C'est une étape très importantes pour les problèmes de data science. Dans plusieurs situations on se retrouve avec une manque de plus ou moins de données dans la base données. Ce manque de données affaiblit la capacité à bien répondre à la problématique posée. Par exemple dans notre cas, cela causera une mauvaise prédiction de la probabilité d'augmentation ou de diminution du prix du pétrole. Pour contourner ce problème plusieurs méthodes existent dont quelques-unes utilisées pour ce travail.

- La méthode d'imputation par la moyenne

Cette méthode consiste à remplacer les valeurs manquantes par la moyenne de la variable en question comme la moyenne (estimateur de position) en espérant que cette moyenne arrive à s'approcher des vraies variables manquantes. Cette méthode n'est pas bonne parce que la moyenne n'est pas un estimateur robuste ce qui fait que lorsqu'on change nos données, la moyenne change complètement.

- La méthode d'imputation par la médiane

La médiane et la moyenne sont des estimateurs de position. La différence existante entre les deux est que la médiane est un estimateur robuste ce qui fait qu'elle est plus utilisée que la moyenne pour l'imputation des valeurs manquantes. L'idée reste pourtant la même, on remplace les valeurs manquantes pas la médiane de la variable en question.

- imputation par la méthode forward et backward

Le principe consiste à remplacer la valeur manquante par la valeur qui lui précède (respectivement par la valeur qui lui succède). Cette méthode est plus utilisée lorsque la variable présente plusieurs minima locaux.

Dans la suite de notre travail, on a rempli les valeurs manquantes par la méthode **forward** disponible dans la bibliothèque **pandas**.

3- Construction d'un premier modèle pour se lancer dans la bataille :

Après avoir fait le traitement de données, on a fait une sélection simple de variable pour se lancer dans le sujet.

```
def parse_model_1(X,Y):
  class_dummies = pd.get_dummies(X['month'], prefix='split_')
  class_dummies = pd.get_dummies(X['country'], prefix='split_')
  X = X[['7_diffSumProduction(kmt)','3_diffSumProduction(kmt)','11_diffSumProduction(kmt)']]
  Y = Y[['7_diffSumProduction(kmt)','3_diffSumProduction(kmt)','11_diffSumProduction(kmt)' ]]
  X = X.join(class_dummies)
  Y = Y.join(class_dummies)
  return X, Y
```

```
X1_train,X1_test=parse_model_1(dat_train.copy(),dat_test.copy())
```

On choisit les variables complètes et continues pour voir le score que l'on aura obtenu. Voir ci-dessous le code utilisé. A ce stade on a réalisé une sélection naïve des variables juste pour un de début et de voir les résultats qu'on aura avec nos trois variables choisis d'une façon pas très inintéressante. On a construit un modèle logistique qu'on verra un peu plus tard et qui nous a permis de calculer notre score avec une fonction qu'on a nommé **comput_scor().** Pour ces trois variables notre score était de 0,666.voir la fonction ci-après.

```
def compute_score(clf, X, y):
    xval = cross_val_score(clf, X, y, cv = 5)
    return(xval,'le score vaut' ,sum((xval)/len(xval)))
```

Après la soumission de mon travail sur le site mon score était à 0,4 au lieu de 0,66. Cette baisse de score est due au faite que je n'ai pas utilisé la même fonction que celle utilisée pour le data challenge.

4-Sélection des variables discriminante et choix de la méthode :

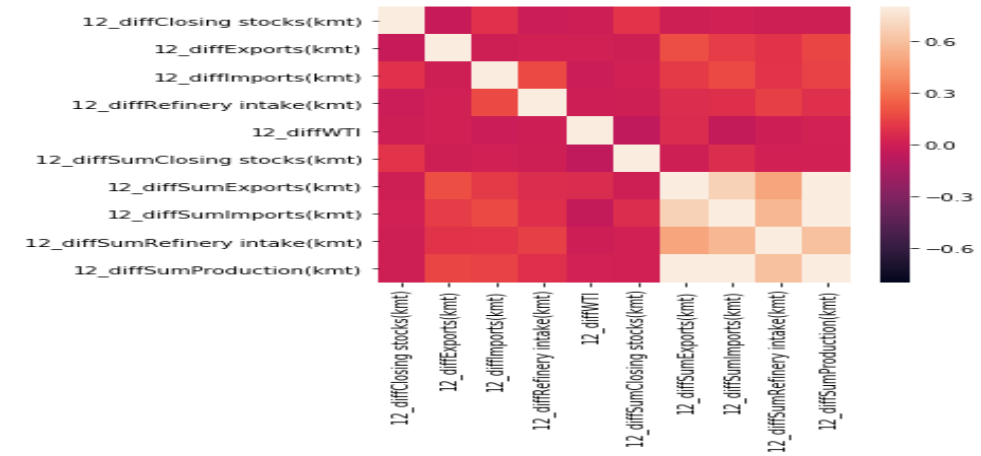
Dans le cas de notre travail, notre Target est un vecteur binaire ce qui nous amène à choisir un modèle logistique pour notre premier modèle sérieux. Ensuite, on a utilisé le **Randomforrest()**, **Gradientboosting()** et enfin le modèle **xgboost()**.

Modèle Logistique

Ce modèle est le plus facile qu'on peut penser de façon naturel. A ce stade avant d'utiliser le modèle logistique, on a commencé par la sélection de variable parce que lors du traitement des données on s'est rendu compte qu'on avait une forte corrélation entre plusieurs variables. On a donc un sur-ajustement avec le modèle complet. Pour la sélection de variables on a utilisé la fonction **heatmap()** de la bibliothèque **seaborn** pour repérer les variables qui sembleront être fortement corrélées**.** Toutes les variables ayant un coefficient de corrélation $\geq 0,8$, ont été considérées comme étant fortement corrélées et dans le cas contraire, les variables sont gardées pour la suite de mon travail (notre modèle sélectionné). Voici une figure ci-dessous représentant une sortie de la fonction **heatmap().**

```
plt.figure(figsize=(5,5))
correlation = train.corr()
sns.heatmap(correlation,vmin=-0.8, vmax=0.8)
```

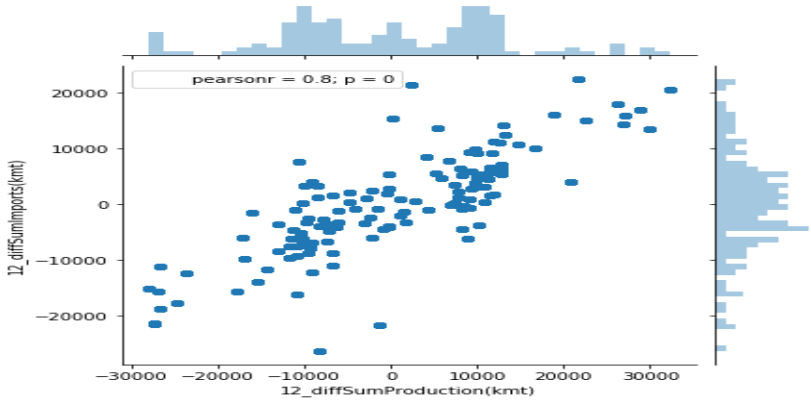
```
<matplotlib.axes._subplots.AxesSubplot at 0x2455b67c860>
```



En regardant la figure ci-dessus on peut voir que la variable **12_diffSumProduction(kmt)** semble être fortement corrélée avec **12_diffSumImport(kmt)** et **12_diffSumExport(kmt)** dans le sens qu'on s'est fixé. Pour valider ce choix, on a utilisé la fonction **sns.joinplot()** pour avoir plus de précision (voir figure ci-après).

```
sns.jointplot(x="12_diffSumProduction(kmt)",
              y="12_diffSumImports(kmt)",
              data=train)
```

```
<seaborn.axisgrid.JointGrid at 0x2455b905278>
```



On voit bien que l'ensemble des observations s'alignent sur la première bissectrice et avec coefficient de corrélation qui vaut 0,8. On peut donc affirmer que les variables sont fortement corrélées. J'ai donc supprimé les variables **12_diffSumImport** et **12_diffSumExport** et gardé la variable **12_diffSumProduction(kmt)** parmi mes variables sélectionnées. En utilisant cette approche j'ai sélectionné les variables que j'ai utilisés pour mon modèle logistique. Enfin, après avoir j'ai construit mon modèle et calculer mon score qui valait 0,3. On constate que mon score a donc diminué de 1 point. Passons maintenant au **GradientboostingClassifier()**.

Gradientboosting

Le **GradientboostingClassifier()** est un modèle de classification supervisée et non supervisée, ce qui nous a permis de l'utiliser dans notre travail. Le gros travail consiste à bien paramétrer le modèle. Si on arrive à bien paramétrer le modèle cela nous permettra d'avoir un meilleur modèle.

On parlera de comment on a sélectionné notre modèle et paramétrer le modèle en utilisant la fonction **gsearch1 = GridSearchCV()** de la bibliothèque **sklearn()**.

La sélection des variables a était réaliser sur les variables sélectionner après la suppression des variables fortement corrélées à l'aide de la méthode **SelectKBest()** de **sklearn()**.

```
import sklearn.feature_selection
for k in range(50,70):
    # k<= nombre de variable
    select = sklearn.feature_selection.SelectKBest(k=k)
    selected_features = select.fit(X_train,y_train)
    indices_selected = selected_features.get_support(indices = True)
    colname_selected = [X.columns[i] for i in indices_selected]
    X_train_selected = X_train[colname_selected]
    X_test_selected = X_test[colname_selected]
    adaboost= GradientBoostingClassifier(learning_rate=0.1, n_estimators=320,max_depth=5,
                                         min_samples_split=200, min_samples_leaf=30,
                                         subsample=0.93,random_state=15,max_features=k)
    adaboost.fit(X_train_selected, y_train)
    ada_pred=adaboost.predict_proba(X_test_selected)
    pred_boost_d,pred_boost_g=prob_pred(ada_pred)
    print("pour k=",k," ","1er_score",score_function(y_test,ada_pred[:,0]),"","2er_score",
          score_function(y_test,ada_pred[:,1]))
    pour k= 57   1er_score 0.797376490497  2er_score 0.202623509503
```

La valeur de k doit être inférieure aux nombre de variable. Le k optimal est celui qui réalise le plus grand score, dans notre cas k=57. La figure ci-dessous montre l'application de la méthode **SelectKBest()**. Les paramètres utilisés pour le modèle du **GradientboostingClassifier()** sont pris de façon aléatoire, ce qu'on va commencer à paramétrer.

Paramétrage du modèle GradientboostingClassifier()

Commençons par la valeur optimale du paramètre **n_estimateur** avec un intervalle de 300 à 600 avec un pas de 10. Cette valeur représente le nombre maximal d'arbre. La valeur optimale vaut 300 pour classer les variables.

```
param_dist = {'n_estimators':[k for k in range(300,600,10)]}
ran = RandomizedSearchCV(GradientBoostingClassifier(learning_rate=0.1,
min_samples_split=3403,min_samples_leaf=50,max_depth=8,max_features=52,
subsample=0.8,random_state=15), param_dist,cv=5, scoring='accuracy',n_iter=10)
ran.fit(X_train_selected[predictors],y_train)
ran.grid_scores_, ran.best_params_, ran.best_score_
{'n_estimators': 300},0.7516896855715545)
```

Recommençons maintenant avec le même intervalle de 300 à 600 avec un pas de 20 pour trouver la valeur optimale de **n_estimateur est** la nouvelle valeur trouvée est de 320.

```
predictors = [x for x in X_train_selected.columns if x not in ['Target']]
param_test1 = {'n_estimators':[k for k in range(300,600,20)]}
gsearch1 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1,
min_samples_split=3403,min_samples_leaf=50,max_depth=8,max_features=57,
                                         subsample=0.8,random_state=10),
param_grid = param_test1, scoring='roc_auc',n_jobs=4,iid=False, cv=10)
gsearch1.fit(X_train_selected[predictors],y_train)
gsearch1.grid_scores_, gsearch1.best_params_, gsearch1.best_score_
{'n_estimators': 320},0.7675167369407294)
```

-max_depth et **min_samples_split**

J'ai réalisé une recherche de grille robuste dans cette section, ce qui a pris 15 à 30 minutes, voire plus. Pour commencer, je vais tester les valeurs **max_depth** de 5 à 16 par pas de 1 et **min_samples_split** de 200 à 1500 par pas de 200. Ces valeurs sont basées sur mon intuition.

```
param_test_2 = {'max_depth':[k for k in range(5,16,1)], 'min_samples_split':[k for k in range(200,1500,200)]}
gsearch_2 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1,max_depth=4, n_estimators=51,
                                                max_features='sqrt', subsample=0.8, random_state=123),
param_grid = param_test_2, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch_2.fit(X_train,y_train)
gsearch_2.grid_scores_, gsearch_2.best_params_, gsearch_2.best_score_
{'max depth': 5, 'min samples split': 400},0.7747854792884274)
```

-Ici, nous prenons 5 comme valeur optimale de **max_depth** et essayé de valeurs différentes pour **min_samples_split** moins élevé. On obtient 432 comme valeur optimale comme pour **min_samples_split** voir ci-dessous.

```
predictors = [x for x in X_train_selected.columns if x not in ['Target']]
param_test_2 = { 'min_samples_split':[k for k in range(420,451,1)]}
gsearch_2 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1,
max_depth=5, n_estimators=441, max_features=57, subsample=0.8, random_state=123),
param_grid = param_test_2, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch_2.fit(X_train_selected[predictors],y_train)
gsearch_2.grid_scores_, gsearch_2.best_params_, gsearch_2.best_score_

mean: 0.79054, std: 0.01043, params: {'min_samples_split': 432}
```

-min_sample_leaf

Ici, nous obtenons 30 comme valeur optimale pour **min_samples_leaf**.

```
predictors = [x for x in X_train_selected.columns if x not in ['Target']]
param_test3 = {'min_samples_leaf':[k for k in range(27,40,1)]}
gsearch3 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1,
max_depth=4, n_estimators=395,min_samples_split=432, max_features=57, subsample=0.8,
                                                        random_state=123),
param_grid = param_test3, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch3.fit(X_train_selected[predictors],y_train)
gsearch3.grid_scores_, gsearch3.best_params_, gsearch3.best_score_
{'min_samples_leaf': 30},0.7890979665854098)
```

-Subsample

Réglage du sous-échantillon et création de modèles avec un taux d'apprentissage supérieur.

L'étape suivante consisterait à essayer les différentes valeurs du sous-échantillon. Prenons les valeurs 0.89,0.9,0.91,0.92,0.93,0.931,0.932,0.935,0.937,0.94,0.95,0.96 .

```
param_test5 = {'subsample':[0.6,0.7,0.75,0.8,0.85,0.9,0.93]}
gsearch5 = GridSearchCV(estimator = GradientBoostingClassifier(learning_rate=0.1,
    subsample=0.8, n_estimators=75,max_depth=5,min_samples_split=432,random_state=123
                min_samples_leaf=30, random_state=10,max_features=57,),
param_grid = param_test5, scoring='roc_auc',n_jobs=4,iid=False, cv=5)
gsearch5.fit(X_train_selected,y_train)
gsearch5.grid_scores_, gsearch5.best_params_, gsearch5.best_score_
{'subsample': 0.93}, 0.7880452125309028)
```

Calcul du score final

Après avoir paramétrer le modèle **Gradientboosting()**, on va le tester avec notre jeu de données train et le tester sur notre jeu données test pour voir en fin la précision de notre modèle. Vous verrez ci-après le code python utilisé pour tester le modèle ainsi que le score obtenu.

```
select = sklearn.feature_selection.SelectKBest(k=57)
selected_features = select.fit(X_train,y_train)
indices_selected = selected_features.get_support(indices = True)
colname_selected = [X.columns[i] for i in indices_selected]
X_train_selected = X_train[colname_selected]
X_test_selected = X_test[colname_selected]
adaboost= GradientBoostingClassifier(learning_rate=0.1, n_estimators=320,
                max_depth=5,min_samples_split=200, min_samples_leaf=30,
                    subsample=0.93, random_state=10,max_features=57)

adaboost.fit(X_train_selected, y_train)
ada_pred=adaboost.predict_proba(X_test_selected)
#pred_boost_d,pred_boost_g=prob_pred(ada_pred)
print("1er_score",score_function(y_test,ada_pred[:,0]))
print("2er_score",score_function(y_test,ada_pred[:,1]))
#1er_score 0.801058512664
#2er_score 0.198941487336
```

Le modèle **Gradientboosting()** améliore de très loin mon score. Notre score a augmenté de 4 points. Ce score restera le mieux que j'ai pu réaliser pour ce data challenge. Après ma soumission mon score final a pris la valeur de 0,81.

CONCLUSION

le modèle **Gradientboosting()** nous a permis d'améliorer la prédiction de la probabilité de la hausse ou la baisse de prix du pétrole. Le **Gradientboosting()** combine un ensemble d'apprenants faibles et offre une meilleure précision de la prédiction. À tout instant t0, les résultats du modèle sont pondérés en fonction des résultats de l'instant précédent t0-1. Les résultats classés correctement reçoivent un poids inférieur et ceux qui ont été classés par erreur sont pondérés plus haut. Cette technique est suivie pour un problème de classification alors qu'une technique similaire est utilisée pour la régression.

Les méthodes de **Gradientboosting()** correspondent à une forme de régression (et de classification) non linéaire extrêmement performante. Le ***Gradientboosting()*** est une méthode ensembliste comme le **R*andomforest()***, qui se base sur des arbres de décision comme méthode sous-jacente à l'ensemble, même si l'ensemble formé est très différent. Alors que le **R*andomforest()*** construit plusieurs arbres en parallèle, le **Gradientboosting()** construit lui aussi k arbres, mais il le fait simultanément (en série). L'arbre $k + 1$ aura accès à son prédécesseur ou plus précisément à l'erreur de son prédécesseur, ce qui permet de corriger l'erreur commise dans l'étape précédente.

Pour un problème de classification, la prédiction est une somme pondérée de chacun des algorithmes faibles et non un vote à la majorité comme le fait le **R*andomforest()***. Pour améliorer cette prédiction, on peut essayer d'améliorer la sélection de variable et de travailler encore plus nos variables catégorielles avant de les faire rentrer dans le modèle ; ce que je n'ai pas fait. Pour la sélection on peut utiliser une sélection de **Ridge** (**Lasso** ou **Elastic net**).

J'ai utilisé le modèle **xgboost()** pour améliorer la précision du **Gradientboosting()** et la même méthode de paramétrisation que le modèle **Gradientboosting()** pour paramétrer le modèle **xgboost()** . Par la suite, je me suis rendu compte à quel point c'est difficile d'améliorer le modèle **xgboost()** .

Vous trouverez en Annexe le code pour le modèle **RandomForest()** et le modèle **xgboost()** .

ANNEXE

Dans cette partie nous mettons les modèles finaux après avoir fait la sélection des variables et paramétré les modèles avec la même procédure que celui du modèle **Gradientboosting()** .

Voici le code le modèle **Randomforest()**

```
rf123 = RandomForestClassifier(n_estimators=320,criterion='entropy',
max_depth=15, min_samples_split=1000,min_samples_leaf=71,
                               random_state=10, max_features=47)
model123=rf123.fit(X_app, y_app)
#importances1 = rf123.feature_importances_
pred123=rf123.predict_proba(X_test)
#prediction de probabilté
pred1234=rf123.predict_proba(dat_test)
print("train",X_test.shape,"test",dat_test.shape)
print("1er_score",score_function(y_test,pred123[:,0]))
print("2er_score",score_function(y_test,pred123[:,1]))
```

```
train (3353, 122) test (2000, 122)
1er_score 0.249986611716
2er_score 0.750013388284
```

```
'1er_score 0.251151807941\n   2er_score 0.748848192059'
```

Voici le code pour le modèle **xgboost()**

```
xgb1 = XGBClassifier(learning_rate =0.1,n_estimators=55,max_depth=5, min_child_weight=1,gamma=0,subsample=0.8,
                     colsample_bytree=0.8,objective= 'binary:logistic', nthread=4, reg_alpha=1e-06,
                      scale_pos_weight=1,seed=27)
xgb1.fit(X_train_selected, y_train)
ada_pred=xgb1.predict_proba(X_test_selected)
print("pour k=","1er_score",score_function(y_test,ada_pred[:,0]),"","2er_score",score_function(y_test,ada_pred[:,1]))
#pour k= 55 1er_score 0.207374645167  2er_score 0.792625354833
```

```
pour k= 1er_score 0.210018571086  2er_score 0.789981428914
```