Stock Return Prediction

by QRT

Préparé par : Salomé AMAR

Pour le : 1 mars 2020

SOMMAIRE

CONTEXTE DU CHALLENGE

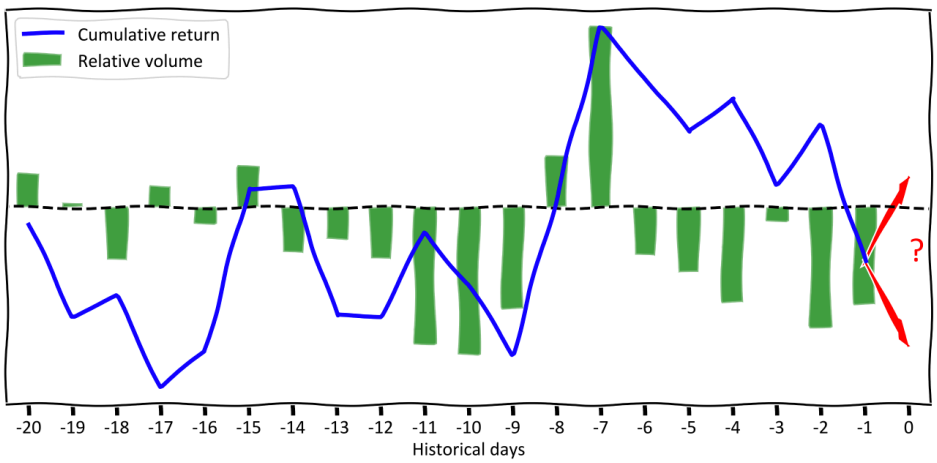
QRT

Le challenge est proposé par la société Qube Research & Technologies Limited, gestionnaire d'investissement quantitatif et systématique. QRT cherche à mettre en oeuvre une approche scientifique de l'investissement financier, en combinant données, recherche, technologie et expertise commerciale.

L'un des objectifs clés de QRT est de creuser de légères informations parmi l'énorme quantité de données disponibles sur le marché. Pour ce faire, des techniques d'apprentissage automatique sont utilisées pour prendre de meilleures décisions commerciales grâce à une analyse approfondie de milliers de sources de données différentes.

Enjeu

Le défi proposé est un problème de classification binaire visant à prédire si le retour d'un stock sur le marché américain est dans le top 50% des rendements boursiers les plus élevés (**1**) ou non (**0**), à l'aide de données historiques sur une période de 20 jours.



Dans un monde financier en mouvement constant, il est extrêmement difficile de détecter les tendances qui font monter ou descendre un titre. Les stratégies d'investissement quantitatives nécessitent l'analyse de données historiques pour prédire la tendance d'un titre dans un avenir proche. Cependant, le niveau extrêmement faible de signal et de bruit en fait un problème très difficile.

DESCRIPTION DES DONNÉES

Les ensembles de données d'entrée comprennent 47 colonnes :

• **ID** : des identifiants de ligne uniques,

• **DATE** : un index de la date (les dates sont aléatoires et anonymisées donc il n'y a pas de continuité ou de lien entre les dates),

• **STOCK** : un indice du stock,

• **INDUSTRY** : un indice du domaine de l'industrie boursière (par exemple, aéronautique, informatique, compagnie pétrolière),

• **INDUSTRY_GROUP** : un indice de l'industrie du groupe,

• **SUB_INDUSTRY** : un indice de niveau inférieur de l'industrie,

• **SECTOR** : un indice du secteur du travail,

• **RET_1 à RET_20** : les rendements résiduels historiques des 20 derniers jours (RET_1 est le rendement de la veille et ainsi de suite),

• **VOLUME_1 à VOLUME_20** : le volume relatif historique échangé au cours des 20 derniers jours (VOLUME_1 est le volume relatif de la veille et ainsi de suite).

Les jeux de données en sortie contiennent deux colonnes :

• **ID** : correspond aux identifiants d'entrée,

• **RET** : le signe du retour du stock résiduel au moment présent (cible binaire).

418 595 observations sont disponibles pour les ensembles de données d'apprentissage et 198 429 observations pour les ensembles de données test.

Le retour d'un stock *j* au jour *t* au prix P_j^t est donné par : $$R_j^t = \frac{P_j^t}{P_j^{t-1}} - 1$$

Le volume relatif d'un stock *j* parmi les n stocks au jour *t* est donné par :

$$\bar{V}_j^t = \frac{V^t}{\mathrm{median}(\{V_j^{t-1}, \ldots, V_j^{t-20}\})}$$

$$\mathcal{V}_j^t = \bar{V}_j^t - \frac{1}{n}\sum_{i=1}^{n} \bar{V}_{je}^t$$

Ainsi, le volume d'un stock est relatif à son passé et au volume moyen des autres stocks. **RET_i** est la performance d'une action par rapport au marché.

La principale difficulté de ce défi sera donc de gérer un jeu de données bruyant et des séries temporelles à la limite de la non stationnarité.

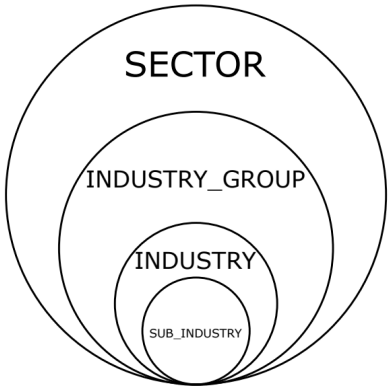
ANALYSE EXPLORATOIRE DES DONNÉES

ID	DATE	STOCK	INDUSTRY	INDUSTRY_GROUP	SECTOR	SUB_INDUSTRY	RET_1	VOLUME_1	RET_2	VOLUME_2	...	VOLUME_16	RET_17
0	0	2	18	5	3	44	-0.015748	0.147931	-0.015504	0.179183	...	0.630899	0.003254
1	0	3	43	15	6	104	0.003984	NaN	-0.090580	NaN	...	NaN	0.003774
2	0	4	57	20	8	142	0.000440	-0.096282	-0.058896	0.084771	...	-0.010336	-0.017612
3	0	8	1	1	1	2	0.031298	-0.429540	0.007756	-0.089919	...	0.012105	0.033824
4	0	14	36	12	5	92	0.027273	-0.847155	-0.039302	-0.943033	...	-0.277083	-0.012659
...	...	...	...	...	...	...	...	...	...	...	...	...	...
418590	223	5703	32	10	4	77	0.021843	-0.217823	-0.021703	-0.125333	...	-0.161543	0.007785
418591	223	5705	35	12	5	91	-0.006920	-0.375251	0.000000	-0.029437	...	-0.955492	-0.016221
418592	223	5709	2	1	1	5	0.021869	-0.978856	-0.005929	-1.026267	...	-0.476550	0.029714
418593	223	5710	33	10	4	83	0.012248	-0.627169	0.010925	-0.842108	...	-0.210079	0.023729
418594	223	5713	26	7	4	60	0.076162	-1.325986	-0.000988	0.198856	...	0.277896	-0.037037

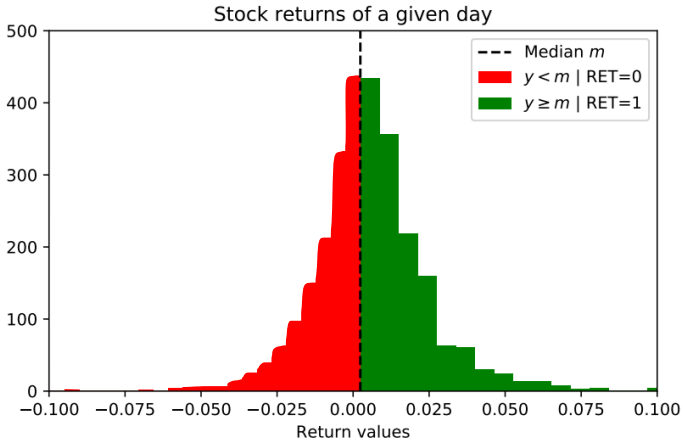
418595 rows × 47 columns

Nous avons chargé les données et affiché les principales caractéristiques du jeu de données. Les corrélations linéaires entre les variables ne semblent pas significatives.

Les features '**DATE**', '**STOCK**', '**INDUSTRY**', '**INDUSTRY_GROUP**', '**SECTOR**', '**SUB_INDUSTRY**' représentent un indice, donc ont un nombre relativement faible de valeurs uniques (entre 12 et 156). On a environ 200 jours non ordonnés. Une première idée serait de transformer ces indices en catégories d'intervalle, afin de garder la notion d'ordre tout en diminuant la complexité de la feature. Nous préférerons créer des features conditionnelles basées sur des statistiques et sur ces features catégoriques.



On rappelle que pour un jour donné, **RET** est 1 si le rendement surpasse la médiane de tous les rendements ce jour. Par construction, le jeu de donnée est équilibré et la médiane de la cible est à 0.



Nous avons beaucoup de valeurs manquantes dans les features **RET_i** et **VOLUME_i** (entre 2000 et 75 000 par colonne). Nous traiterons les valeurs manquantes par la suite, de deux manières différentes.

APPLICATION DES MODÈLES

Features engineering et features selection

Le principal inconvénient du défi étant de gérer le bruit, nous allons créer des features qui agrège d'autres features avec des statistiques, et ensuite nous sélectionnerons les features à garder dans nos modèles.

Pour ceci, nous créons des classes héritant de BaseEstimator et TransformerMixin, et on les utilisera dans une Pipeline, qui chaînera les différentes opérations.

La classe *Features_engi* calcule des statistiques sur une cible donnée conditionnellement à certaines features. Nous voulons générer des features décrivant la médiane de **RET_1** , **RET_5**, **RET_11**, **RET_15**, **RET_20** conditionnellement au **SECTEUR** et à la **DATE**, et la médiane de **VOLUME_1**, **VOLUME_5**, **VOLUME_11**, **VOLUME_15**, **VOLUME_20** conditionnellement à **STOCK** et **INDUSTRY**.

La classe *Features_select* sélectionne les features à garder dans le modèle. Si l'on ne précise rien, toutes les features sont gardées (y compris les nouvelles features). On peut donner en argument une liste précise de features ou choisir la liste de features déjà pré-sélectionnée qui comprend toutes les nouvelles features et les features **RET_1** à **RET_20**, **VOLUME_1** à **VOLUME_20**.

Pre processing

Nous remplirons les valeurs manquantes de deux manières différentes : en remplaçant par 0 ou par la moyenne. Pour ceci, nous utilisons la classe SimpleImputer de Scikit-learn, que nous mettrons dans notre Pipeline. Afin de normaliser les valeurs, nous ajouterons la classe MinMaxScale de Scikit-learn à notre Pipeline.

Création des Pipelines

Nous avons choisi d'appliquer des méthodes de boosting et de bagging. Notre mesure d'évaluation de performance des modèles sera l'*Accuracy,* métrique utilisée dans le benchmark disponible du challenge. Pour information, le score public généré par ce benchmark est de 51.31%.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

$$\begin{aligned}
\text{TP} &= \sum_{i=1}^{n} \mathbf{1}_{y_i=1,\hat{y}_i=1} \\
\text{TN} &= \sum_{i=1}^{n} \mathbf{1}_{y_i=-1,\hat{y}_i=-1} \\
\text{FN} &= \sum_{i=1}^{n} \mathbf{1}_{y_i=1,\hat{y}_i=-1} \\
\text{FP} &= \sum_{i=1}^{n} \mathbf{1}_{y_i=-1,\hat{y}_i=1}
\end{aligned}$$

Nous allons créer 5 modèles différents :

1. création de features, sélection des features, remplissage des valeurs manquantes par 0, normalisation des valeurs, random forest,

```
clf_zero_rf = Pipeline([("engin1",Features_engi()),
                        ('engin2', Features_engi(target_feature = ['VOLUME'], gb_features = ['STOCK','INDUSTRY'])),
                        ('selection', Features_select('pre-defini')),
                        ("zero", SimpleImputer(missing_values=np.nan,strategy='constant', fill_value=0)),
                        ("scale", MinMaxScaler()),
                        ("rf", RandomForestClassifier(n_estimators= 200, max_depth= 5))])

clf_zero_rf.fit(X_train, y)
clf_zero_rf.predict(X_train)
score_zero_rf = cross_val_score(clf_zero_rf, X_train, y, cv=st_cv, scoring = 'accuracy')
score_zero_rf.mean()
```

2. création de features, sélection des features, remplissage des valeurs manquantes par la moyenne, normalisation des valeurs, random forest,

```
clf_mean_rf = Pipeline([("engin1", Features_engi()),
                        ('engin2', Features_engi(target_feature = ['VOLUME'], gb_features = ['STOCK','INDUSTRY'])),
                        ('selection', Features_select('pre-defini')),
                        ("mean", SimpleImputer(missing_values=np.nan,strategy='mean')),
                        ("scale", MinMaxScaler()),
                        ("rf", RandomForestClassifier(n_estimators= 200, max_depth= 5))])

clf_mean_rf.fit(X_train, y)
score_mean_rf = cross_val_score(clf_mean_rf, X_train, y, cv=st_cv, scoring = 'accuracy')
score_mean_rf.mean()
```

3. création de features, sélection des features, remplissage des valeurs manquantes par 0, normalisation des valeurs, gradient boosting,

```
clf_zero_gb = Pipeline([('engin1', Features_engi()),
                        ('engin2', Features_engi(target_feature = ['VOLUME'], gb_features = ['STOCK','INDUSTRY'])),
                        ('selection', Features_select('pre-defini')),
                        ("zero", SimpleImputer(missing_values=np.nan,strategy='constant', fill_value=0)),
                        ("scale", MinMaxScaler()),
                        ("gb", GradientBoostingClassifier())])

clf_zero_gb.fit(X_train, y)
score_zero_gb = cross_val_score(clf_zero_gb, X_train, y, cv=st_cv, scoring = 'accuracy')
score_zero_gb.mean()
```

4. création de features, sélection des features, remplissage des valeurs manquantes par la moyenne, normalisation des valeurs, gradient boosting,

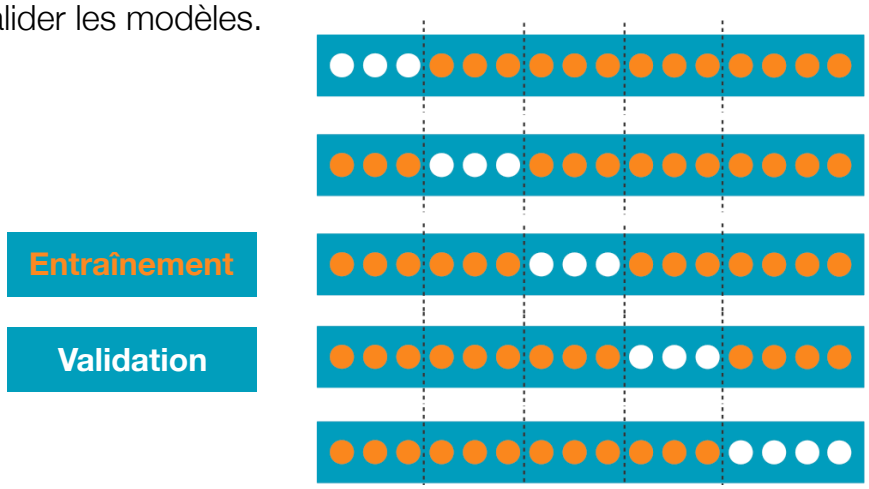
```
clf_mean_gb = Pipeline([('engin1', Features_engi()),
                        ('engin2', Features_engi(target_feature = ['VOLUME'], gb_features = ['STOCK','INDUSTRY'])),
                        ('selection', Features_select('pre-defini')),
                        ("mean", SimpleImputer(missing_values=np.nan,strategy='mean')),
                        ("scale", MinMaxScaler()),
                        ("gb", GradientBoostingClassifier())])

clf_mean_gb.fit(X_train, y)
score_mean_gb = cross_val_score(clf_mean_gb, X_train, y, cv=st_cv, scoring = 'accuracy')
score_mean_gb.mean()
```

5. création de features (**RET_1** à **RET_5**, **VOLUME_1** à **VOLUME_5** conditionnellement à **SECTOR**, **INDUSTRY**, **STOCK**), sélection des features (nouvelles features et **RET_1** à **RET_5**, **VOLUME_1** à **VOLUME_5**), remplissage des valeurs manquantes par la moyenne, normalisation des valeurs, random forest.

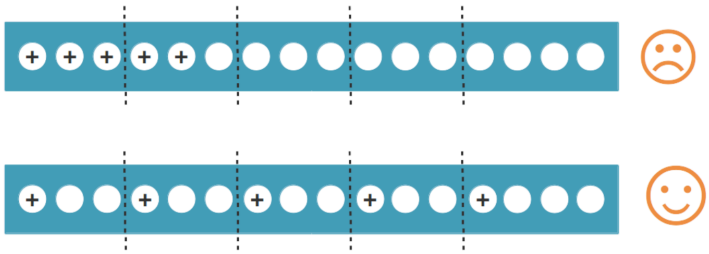
```
lst = ['RET_%d' % (i+1) for i in range(5)]
lst += ['VOLUME_%d' % (i+1) for i in range(5)]
lst += ['VOLUME_%d_SECTOR_INDUSTRY_STOCK_median' % (i+1) for i in range(5)]
lst += ['RET_%d_SECTOR_INDUSTRY_STOCK_median' % (i+1) for i in range(5)]


clf_mean_rf2 = Pipeline([("engin1", Features_engi(shifts = [1,2,3,4,5], statistics = ['median'], target_feature = ['
                        ('selection', Features_select(lst)),
                        ("mean", SimpleImputer(missing_values=np.nan,strategy='mean')),
                        ("scale", MinMaxScaler()),
                        ("rf", RandomForestClassifier(n_estimators= 200, max_depth= 5))])

clf_mean_rf2.fit(X_train, y)
score_mean_rf2 = cross_val_score(clf_mean_rf2, X_train, y, cv=st_cv, scoring = 'accuracy')
score_mean_rf2.mean()
```

Pour les deux Random Rorest, nous avons choisi comme paramètres 200 arbres de profondeur maximale 5 noeuds. Pour les deux Gradient Boosting, nous avons gardé les paramètres par défaut de Scikit-learn.

Nous exécutons également une validation croisée stratifiée à K = 5 fold afin de valider les modèles.



On stratifie afin de ne pas avoir des proportions différentes de 1 et de 0 dans les jeux d'entraînement et de validation, ce qui pourrait biaiser les résultats.



Choix du modèle et génération des prédictions

Voici les scores obtenus par nos modèles :

1	2	3	4	5
0.51433002	0.51525693	0.51211549	0.51266255	0.51232332

Le modèle ayant l'Accuracy la plus élevée est le modèle 2. Random Forest où les valeurs manquantes ont été remplies par la moyenne. Nous choisissons alors ce modèle pour faire nos prédictions.

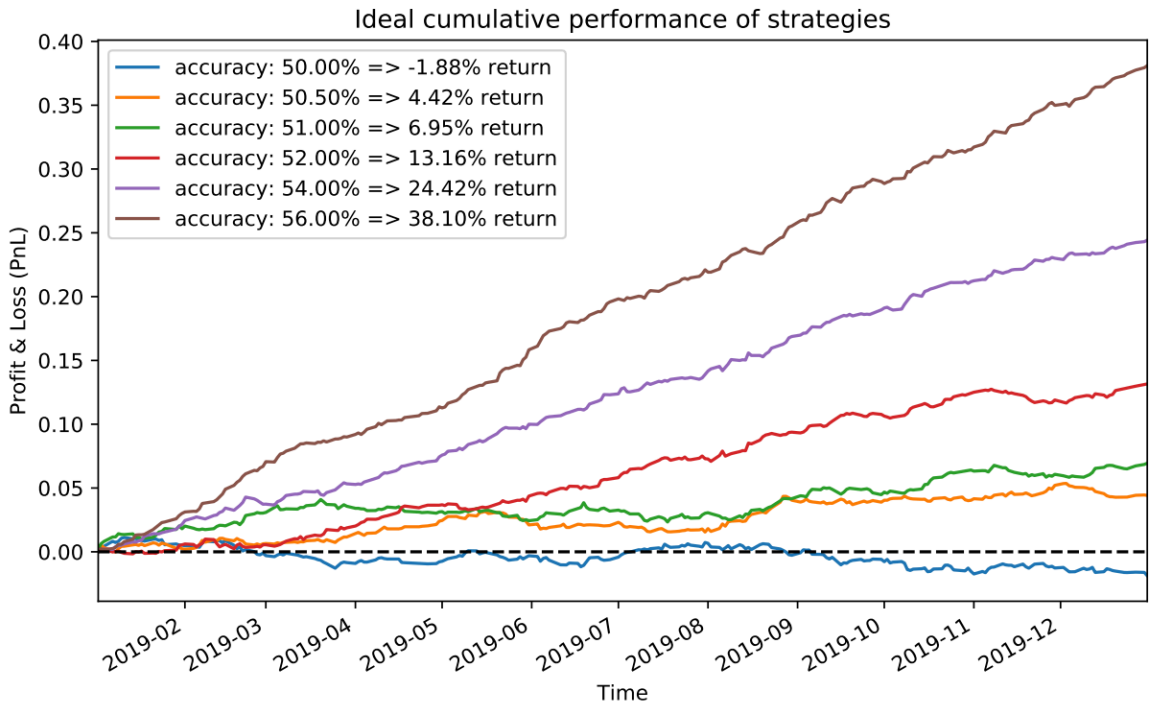
Afin d'affiner encore notre modèle, nous tentons d'effectuer une validation croisée afin de trouver de meilleurs paramètres pour la Random Forest.

```
param_grid = [
{'rf__max_depth': [6, 8, 10], 'rf__max_features': [10, 20], 'rf__n_estimators': [100, 200]}
]

grid_search_forest = GridSearchCV(clf_mean_rf, param_grid, cv=st_cv)
grid_search_forest.fit(X_train, y)
```

Les paramètres optimaux sont :

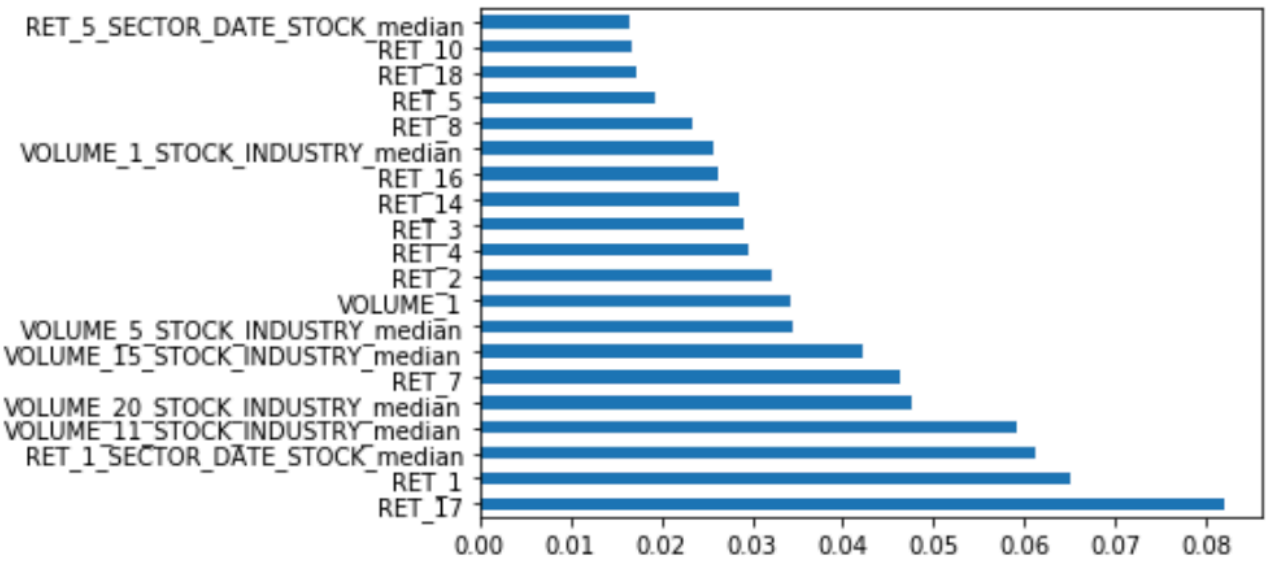
```
{'rf__max_depth': 8, 'rf__max_features': 20, 'rf__n_estimators': 200}
```

Avec ce modèle, notre *Accuracy* est désormais de 51.52%. On a donc entre 6.95% et 13.16% de rendement, soit 3 à 6 fois plus qu'avec l'aléatoire.



Features importance

Afin d'expliquer notre modèle, nous regardons quelles sont les features les plus utilisées. Certaines des nouvelles features apparaissent, ce qui est bon signe.



Pour aller plus loin

Afin d'améliorer notre approche, nous aurions pu créer d'autres features, en changeant les paramètres de la classe ***Features_engi*** ou encore par d'autres méthodes, sélectionner d'autres features dans le modèle, tester d'autres métriques d'évaluation (par exemple l'aire sous la courbe ROC), analyser les données de manières plus approfondie, expliquer les prédictions avec des techniques telles que SHAP Values…