Rapport Projet 8

Compétition Kaggle

Ubiquant Market Prediction

Marie-France LAROCHE-BARTHET

Avril 2022

Table des matières

Contexte

La compétition

Ubiquant Investment (Beijing) Co., Ltd est un fond spéculatif quantitatif national de premier plan basé en Chine. Créés en 2012, ils s'appuient sur des talents internationaux en mathématiques et en informatique ainsi que sur une technologie de pointe pour stimuler les investissements quantitatifs sur les marchés financiers.

Dans ce concours, il est demandé aux candidats de construire un modèle qui prévoit le taux de retour d'un investissement. Le modèle est entrainé et testé sur les prix historiques. En cas de succès, cela améliorera la capacité des chercheurs quantitatifs à prévoir les rendements et permettra aux investisseurs de toute échelle de prendre de meilleures décisions.

Il s'agit d'une compétition de prévisions avec une phase de formation active et une deuxième période où les modèles seront exécutés par rapport aux données réelles du marché. La compétition se déroule du 18 janvier au 18 avril 2022.

Les données

Les données sont disponibles à cette adresse : https://www.kaggle.com/competitions/ubiquant-market-prediction/data. Cet ensemble de données contient des fonctionnalités dérivées de données historiques réelles provenant de milliers d'investissements.

L'évaluation

Les soumissions sont évaluées sur la moyenne du coefficient de corrélation de Pearson pour chaque ID de temps. Il est demandé d'utiliser l'API de séries chronologiques python fournie. Ainsi, dans le kernel Kaggle, il faut utiliser le code suivant :

```
import ubiquant
env = ubiquant.make_env()   # initialize the environment
iter_test = env.iter_test()    # an iterator which loops over the test set and
sample submission
for (test_df, sample_prediction_df) in iter_test:
    sample_prediction_df['target'] = 0  # make your predictions here
    env.predict(sample_prediction_df)   # register your predictions
```

Une erreur est obtenue en cas de soumission incluant des valeurs nulles ou infinies, et les soumissions qui n'incluent qu'une seule valeur de prédiction recevront un score de -1.

Aspects considérés de la compétition

Cette compétition a été l'occasion pour moi d'évaluer des modèles de machine learning de familles différentes : régression linéaire, ElasticNet, XGBoost et réseaux de neurones convolutionnels unidimensionnels. L'influence de la réduction du nombre de variables via l'analyse en composantes principales (notée PCA par la suite), ainsi que la mise à l'échelle des données ont également été étudiées.

Analyse des données sur dataset entier

Le dataset original « pèse » 17,2 Go, une version « allégée » disponible au format parquet a été utilisée (taille 3,6 Go)[1].

Description des données

Le dataset *train.csv* comprend 3 141 410 lignes et 304 colonnes dont voici les informations :

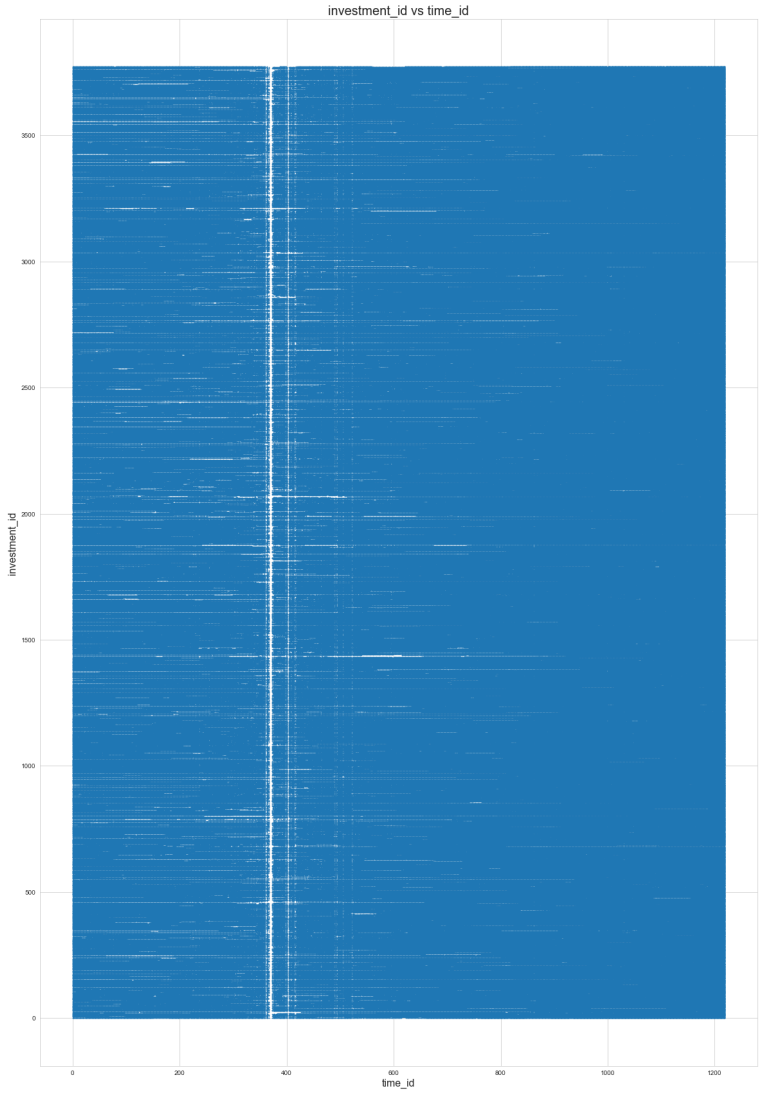
[1] https://www.kaggle.com/code/camilomx/parquet-format-quickstart

- **row_id** : identifiant unique de la ligne
- **time_id** : code ID pour le moment où les données ont été recueillies. Les time IDs sont dans l'ordre, mais le temps réel entre les time IDs n'est pas constant et sera probablement plus court dans le testset que dans l'ensemble du trainset
- **investment_id** : code ID pour un investissement. Les investissements ne sont pas tous présents à toutes les time IDs
- **target** : variable cible, appelée target dans la suite du rapport
- **[f_0:f_299]** : variables « anonymes » générées à partir de données de marchés. Nous les nommerons features dans la suite du rapport.

Le dataset ne contient pas de valeurs manquantes.

Investment_id et time_id

Investment_Id prend 3579 valeurs différentes et time_id 1211. Lorsque nous traçons investment_id en fonction de time_id[2], nous remarquons qu'il y a plus d'investment_id pour des time_id élevés.

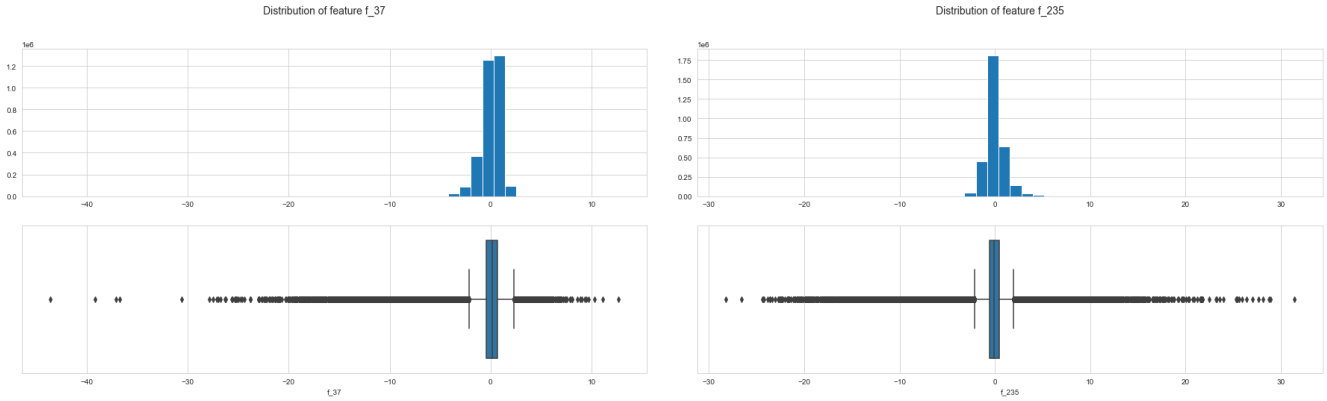


INVESTMENT_ID EN FONCTION DE TIME_ID

[2] inspiré de https://www.kaggle.com/code/jiahauc/ubiqunt-eda-linearregression

Features

Les features ont globalement une moyenne proche de zéro. Néanmoins leurs distributions ne sont pas forcément symétriques. Par ailleurs, nous notons la présence d'outliers dans certaines distributions.

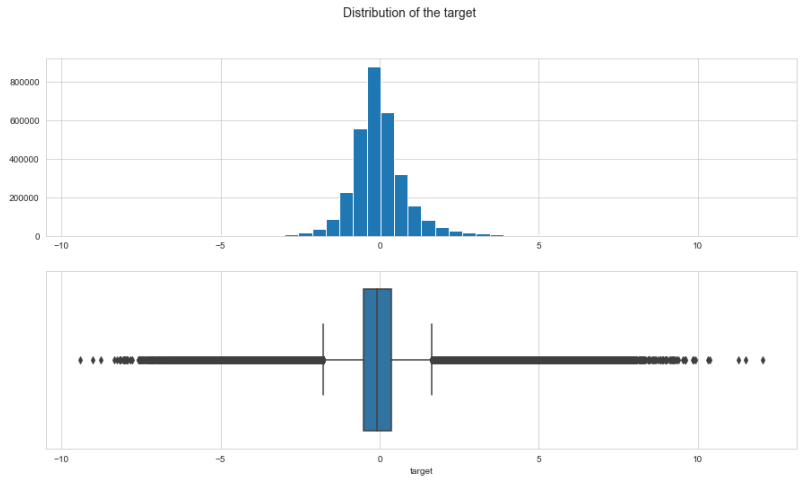


EXEMPLES DE DISTRIBUTION FEATURE F_i

Il pourrait donc être intéressant d'effectuer un RobustScaler pour mettre à l'échelle les données et diminuer l'influence des outliers dans les modélisations.

Target

Celle-ci est plutôt centrée en zéro et comprend aussi des outliers.



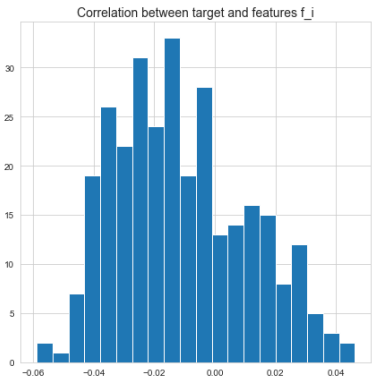
DISTRIBUTION DE LA TARGET

Analyse multivariée sur dataset réduit

Pour raisons de performance d'ordinateur nous réduisons la taille du dataset en vue de l'analyse multivariée ainsi que pour la modélisation. Le dataset réduit est constitué est avec 5% des données choisies de manière aléatoire, puis trié par time_id et investment_id. Ce dataset comporte 157 070 lignes.

Target vs features

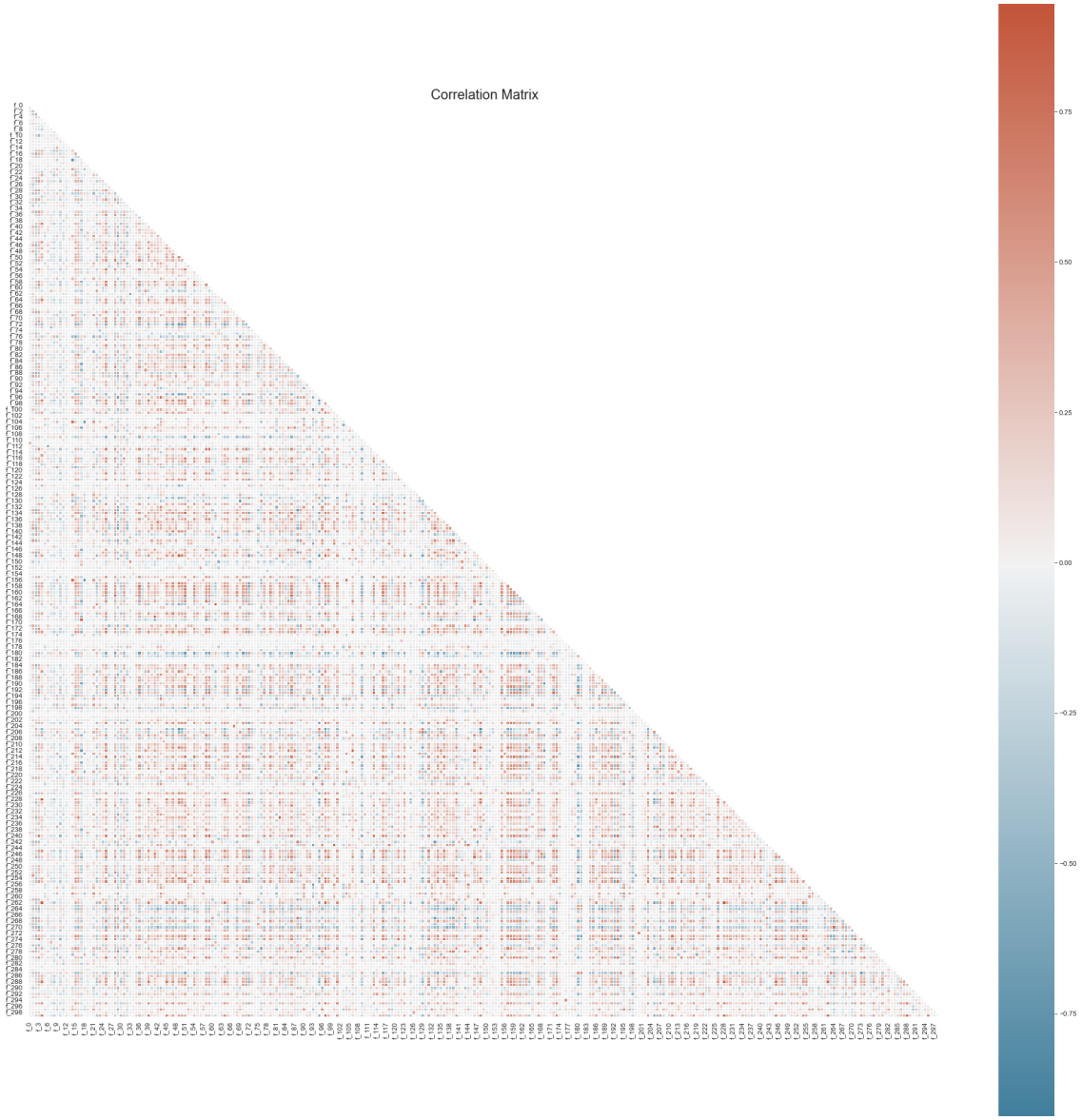
Comme le montre le graphe ci-dessous, la target est peu corrélée aux features :



DISTRIBUTION DES CORRÉLATIONS DE LA TARGET AVEC LES FEATURES F_i

Entre les features

Comme le montre la matrice des corrélations ci-dessous[3], les features sont globalement peu corrélées entre elles.



MATRICE DES CORRÉLATIONS ENTRE LES FEATURES F_i

[3] inspiré de https://www.kaggle.com/code/bastiendelaval/analyse-oc

Notons néanmoins la présence de quelques fortes corrélations (au-dessus de 0.80 en valeur absolue). Nous envisageons de générer des jeux de données en supprimant ces features fortement corrélées du dataset de deux manières : les f_i faibles (avec i les plus petits, dataset résultant suffixé _low) et les f_i élevées (avec i les plus grands, dataset résultant suffixé _up). Cette méthode permet d'identifier rapidement quelles sont les features concernées.

Preprocessing

Split du dataset réduit

Nous constituons notre jeu de données d'entrainement avec les 140 000 premières lignes du dataset réduit, les lignes restantes constituant le jeu de test (soit 10.9% des données). Nous isolons la target dans les deux sets (y_train et y_test). Nous ne conservons que les données des colonnes features.

Scaling

Pour l'influence du scaling (mise à l'échelle des données) sur la modélisation, nous utilisons un RobustScaler, qui a tendance à diminuer l'influence des outliers.

PCA

Pour l'influence de la réduction de dimension sur la modélisation, nous utilisons la PCA. Nous choisissons une réduction du nombre de features conservant 85% de la variance expliquée.

Récapitulatif des données générées

Le tableau ci-dessous récapitule les différentes données générées (sous forme de arrays). Nous testerons les différents modèles de machine learning avec les lignes surlignées en jaune (toutes ces données ne seront pas systématiquement utilisées pour chaque modèle) :

X_data	Nb features	% Réduction de dimension via PCA	Description
X_train	300	/	Données sans scaling ni PCA
X_scaled	300	/	Données avec scaling mais sans PCA
X_pca85	125	58 %	Données sans scaling mais avec PCA
X_scal_pca85	10	96.7%	Données avec scaling et PCA
X_train_up	280	/	Données sans scaling ni PCA et en supprimant les features f_i élevés fortement corrélées
X_up_scaled	280	/	Données avec scaling mais sans PCA et en supprimant les features f_i élevés fortement corrélées
X_up_pca85	126	55 %	Données sans scaling mais avec PCA et en supprimant les features f_i élevés fortement corrélées
X_up_scal_pca85	9	96.8%	Données avec scaling et PCA et en supprimant les features f_i élevés fortement corrélées
X_train_low	280	/	Données sans scaling ni PCA et en supprimant les features f_i faibles fortement corrélées
X_low_scaled	280	/	Données avec scaling mais sans PCA et en supprimant les features f_i faibles fortement corrélées
X_low_pca85	125	55.4%	Données sans scaling mais avec PCA et en supprimant les features f_i faibles fortement corrélées
X_low_scal_pca85	9	96.8%	Données avec scaling et PCA et en supprimant les features f_i faibles fortement corrélées

Modèles testés

Régression Linéaire

Il s'agit du modèle le plus basique pour résoudre un problème de régression. Sa notation matricielle est : $Y = X\beta + \epsilon$.

Nous l'utilisons ici en tant que modèle de base et le notons LinearRegression.

ElasticNet

Il s'agit d'une régression linéaire avec une combinaison des pénalités de régularisation L1 (norme L1 = somme des valeurs absolues des coefficients) et L2 (norme L2 = somme des carrés des coefficients). La fonction à minimiser est :

$$\min_{\beta_1,\cdots,\beta_p} \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{p} \beta_j z_{ij} \right)^2 + \lambda \left[\alpha \sum_{j=1}^{p} |\beta_j| + (1-\alpha) \sum_{j=1}^{p} \beta_j^2 \right]$$

avec λ coefficient de pénalité (ou régularisation) et α ratio L1.[4]

Ce modèle est un mélange de régression LASSO et de régression RIDGE.

XGBoost

Il s'agit d'une implémentation open source optimisée de l'algorithme d'arbres de boosting de gradient (eXtreme Gradient Boosting). Le Boosting de Gradient est un algorithme d'apprentissage supervisé dont le principe est de combiner les résultats d'un ensemble de modèles plus simple et plus faibles afin de fournir une meilleure prédiction.

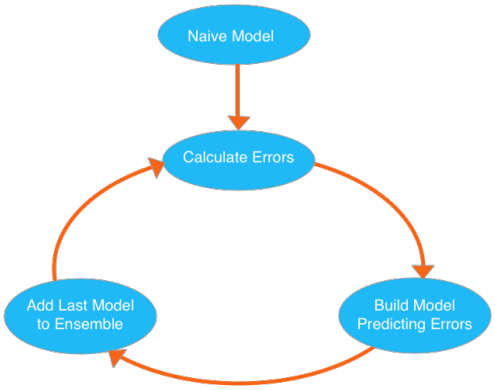


SCHÉMA DU PRINCIPE DE XGBOOST *(source : https://www.kaggle.com/code/dansbecker/xgboost/notebook)*

L'algorithme passe par des cycles qui construisent à plusieurs reprises de nouveaux modèles et les combinent dans un modèle d'ensemble. L'algorithme commence le cycle en calculant les erreurs pour chaque observation dans l'ensemble de données. Il construit ensuite un nouveau modèle pour les prédire. Il ajoute les prédictions de ce modèle de prédiction d'erreurs à « l'ensemble de modèles ».

Pour faire une prédiction, l'algorithme ajoute les prédictions de tous les modèles précédents. Il utilise ces prédictions pour calculer de nouvelles erreurs, construire le modèle suivant et l'ajouter à l'ensemble.

L'algorithme a besoin d'une prédiction de base pour démarrer le cycle. En pratique, les prédictions initiales peuvent être assez naïves. Même si ces prédictions sont extrêmement inexactes, des ajouts ultérieurs à l'ensemble corrigeront ces erreurs.

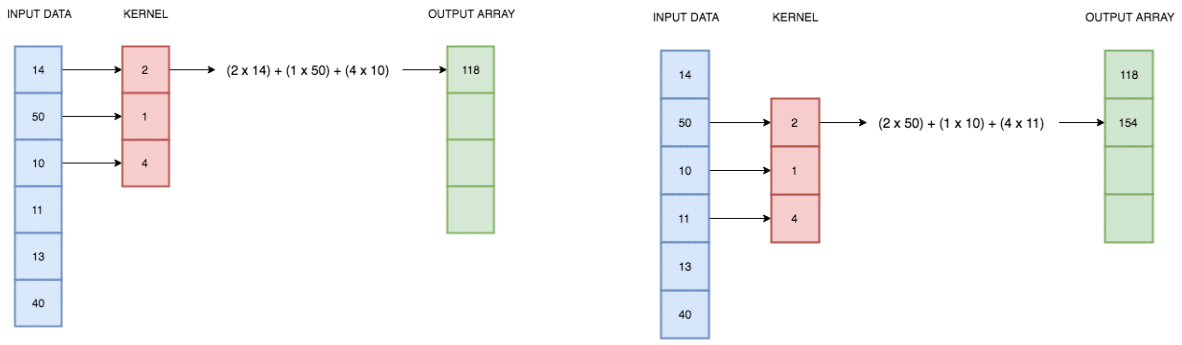
Dans le cadre de la compétition, nous utilisons XGBRegressor.

---

[4] image de la formule issue de https://eric.univ-lyon2.fr/~ricco/cours/slides/regularized_regression.pdf

Réseaux de neurones convolutionnels unidimensionnels (Conv1D)

Il s'agit d'une sous catégorie des réseaux de neurones utilisant l'opération mathématique de convolution via des filtres (ou kernels). Dans le cas de tels réseaux, le kernel glisse le long d'une seule dimension. Ce type de neurones est utilisé dans les séries temporelles ou dans le cadre de textes ou d'audios. Chaque ligne du dataset avec ses n colonnes est en fait transformée en un vecteur colonne à n lignes[5]. Voici un exemple de fonctionnement de convolution à 1D :



EXEMPLE DE CONVOLUTION À 1D *(source : https://blog.floydhub.com/reading-minds-with-deep-learning/*

Plusieurs couches de convolution sont empilées avec des fonctions d'activation en sortie. Des couches de Pooling peuvent être ajoutées en sortie des couches de convolution pour diminuer la taille des cartes de caractéristiques. Une couche de Maxpooling va retenir la valeur maximale d'une cellule dans une zone de la carte de caractéristiques.

Des couches Dropout, utilisées pour éviter le surajustement dans les réseaux de neurones, définissent de manière aléatoire un «taux» de fraction d'unités d'entrée à 0 à chaque mise à jour pendant le temps d'entraînement.

Une couche de Flatten convertit les données multidimensionnelles en un seul vecteur à traiter.

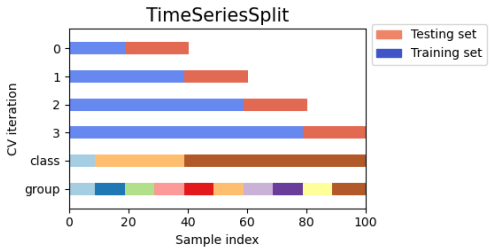
Des couches Dense (réseau neuronal standard) sont ajoutées en fin de réseau qui se termine par une couche Dense constituée d'une seule sortie pour retourner les prédictions.

Nous testons différents empilement de couches qui sont indiqués dans l'annexe.

Entrainement des modèles

Utilisation de TimeSeriesSplit

Pour les modèles classiques de machine learning (LinearRegression, ElasticNet et XGBRegressor), nous avons effectué une validation croisée sur le jeu de train en utilisant TimeSeriesSplit de la librairie scikit-learn. Cet outil permet de générer les folds selon le schéma ci-dessous :



VALIDATION CROISÉE UTILISANT UN TIME SERIES SPLIT AVEC K =4 *(source : https://scikit-learn.org/stable/modules/cross_validation.html#time-series-split)*

Dans notre cas, nous avons choisi 5 folds et une taille fixe du jeu de validation de 20 000 lignes.

[5] en suivant ce kernel https://www.kaggle.com/code/kmkarakaya/1-dimensional-convolution-conv1d-for-regression/notebook

Pour les modèles de Conv1D, le jeu de validation est généré pendant la phase d'entrainement : il représente les 20 derniers % du jeu d'entrainement initial.

Métriques utilisées

Les métriques principales utilisées sont :

- la RMSE (Root Mean Squared Error) : $RMSE = \sqrt{(\frac{1}{n}) \sum_{i=1}^{n} (actual_i - predict_i)^2}$
- le coefficient de corrélation de Pearson : $pearson = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2 \sum_{i=1}^{n} (y_i - \bar{y})^2}}$

Nous avons également regarder en complément les métriques suivantes :

- la MAE (Mean Absolute Error) : $MAE = (\frac{1}{n}) \sum_{i=1}^{n} \left| actual_i - predict_i \right|$
- la MSE (Mean Squared Error) : uniquement pour les modèles Conv1D et en tant que fonction coût $MSE = (\frac{1}{n}) \sum_{i=1}^{n} (actual_i - predict_i)^2$

La RMSE a été évaluée sur les jeux de validation et de test pour tous les modèles. Le coefficient de corrélation de Pearson n'a pas été évalué sur les jeux de validation des modèles Conv1D. La MAE a été évaluée sur les jeux de validation et de test pour LinearRegression et Conv1D.

Optimisation des modèles

L'entrainement des modèles ElasticNet et XGBRegressor s'est fait via une grille de recherches des meilleurs hyperparamètres. Pour ElasticNet, nous avons utilisé l'outil GridSearchCV de scikit-learn et pour XGBRegressor nous l'avons fait manuellement[6].

Pour Conv1D, nous avons entrainé les modèles selon les cas sur 30 ou 50 epochs maximum. Nous avons mis en place un callback d'earlystopping, paramètre pour arrêter l'entrainement du modèle après 5 epochs (10 dans certains cas) sans minimisation de la RMSE sur le jeu de validation. Le callback checkpoint permet alors d'enregistrer les poids des paramètres du modèle correspondant à la RMSE la plus basse sur le jeu de validation. Pour le meilleur modèle obtenu, nous avons testé un autre learning_rate. Les données sont envoyées dans le réseau par lot (batch) de taille 4096.

[6] tutoriel utilisé : https://towardsdatascience.com/beyond-grid-search-hypercharge-hyperparameter-tuning-for-xgboost-7c78f7a2929d

Récapitulatif des modèles testés

Les tableaux ci-dessous rassemblent les différentes combinaisons X_data/modèle qui ont été testées :

X_data	X_train	X_scaled	X_pca85	X_scal_pca85
Modèles	LinearRegression Conv1D_1 à 7 Conv1D_1_1	LinearRegression ElasticNet Conv1D_1 et 6 Conv1D_6_1	LinearRegression XGBRegressor Conv1D_1	LinearRegression ElasticNet XGBRegressor

X_data	X_up_pca85	X_up_scal_pca85	X_low_pca85	X_low_scal_pca85
Modèles	LinearRegression	LinearRegression ElasticNet	LinearRegression	LinearRegression ElasticNet

Résultats

Sur le jeu de validation

LinearRegression

Les résultats sont présentés dans le tableau ci-dessous :

X_data	RMSE	MAE	Pearson_coef
X_up_pca85	**0.906208**	**0.618814**	**0.105118**
X_low_pca85	0.906225	0.618842	0.10491
X_pca85	0.906316	0.618908	0.104369
X_low_scal_pca85	0.908127	0.620206	0.07334
X_up_scal_pca85	0.908131	0.620209	0.07328
X_scal_pca85	0.9082	0.620276	0.072287
X_scaled	0.908742	0.621333	0.09949
X_train	0.908742	0.621333	0.09949

Les trois métriques vont dans le même sens pour les trois premiers modèles et les meilleurs scores sont obtenus avec X_up_pca85. Une trop grande réduction du nombre de features n'est pas optimale pour la LinearRegression car cela génère les plus mauvais scores. Nous constatons également que les scores du coefficient de corrélation de Pearson sont assez faibles.

Nous retenons comme meilleur modèle de LinearRegression celui entrainé avec X_up_pca85.

ElasticNet

Les résultats sont présentés dans le tableau ci-dessous :

X_data	RMSE	Pearson_coef
X_scaled	**0.90662**	**0.10346**
X_low_scal_pca85	0.90799	0.07408
X_up_scal_pca85	0.90802	0.07402
X_scal_pca85	0.90802	0.07346

Les deux métriques vont dans le même sens et les meilleurs scores sont obtenus avec X_scaled. Ici aussi, une trop grande réduction du nombre de features n'est pas optimale pour la modélisation. Les métriques sont du même ordre de grandeur que celles obtenues avec LinearRegression.

Nous retenons comme meilleur modèle de ElasticNet celui entrainé avec X_scaled.

XGBRegressor

Les résultats sont présentés dans le tableau ci-dessous :

X_data	RMSE	Pearson_coef
X_pca85	**0.906451**	**0.091055**
X_scal_pca85	0.907621	0.075637

Les deux métriques vont dans le même sens et les meilleurs scores sont obtenus avec X_pca85.
Nous gardons néanmoins les deux modèles XGBRegressor pour la comparaison finale.

Conv1D

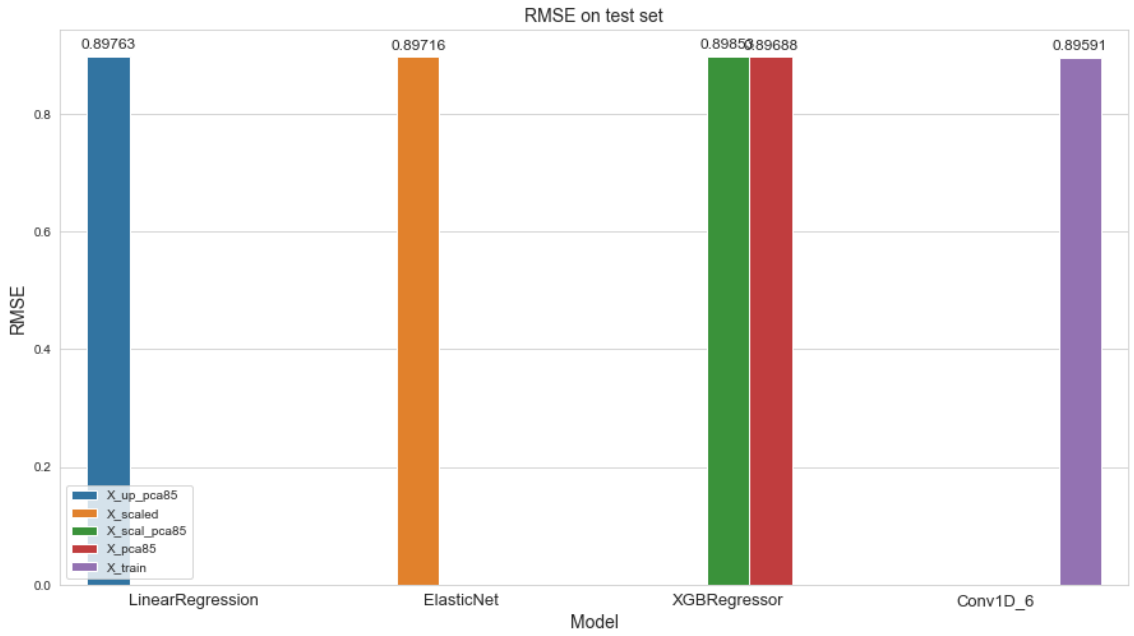
Les résultats sont présentés dans le tableau ci-dessous :

Modèle	X_data	RMSE	Loss = MSE	MAE
Conv1D_6	X_train	**0.9041**	**0.81741**	0.61136
Conv1D_6_sc_1	X_scaled	0.90418	0.81755	0.61078
Conv1D_5	X_train	0.90451	0.82206	0.61128
Conv1D_4	X_train	0.90456	0.82366	0.61056
Conv1D_3	X_train	0.9049	0.81885	0.61167
Conv1D_1_1	X_train	0.90516	0.81931	0.61157
Conv1D_2	X_train	0.90526	0.8195	0.6135
Conv1D_1	X_train	0.90528	0.81954	0.61126
Conv1D_scal_1	X_scaled	0.90531	0.8196	**0.60981**
Conv1D_pca_1	X_pca85	0.90548	0.81989	0.61169
Conv1D_7	X_train	0.90569	0.82028	0.61073
Conv1D_6_sc	X_scaled	0.90599	0.82081	0.611

Les RMSE et MSE vont dans le même sens et les meilleurs scores sont obtenus avec X_train et le modèle Conv1D_6. Par contre, pour la MAE le meilleur modèle est Conv1D_scal_1 avec X_scaled.
Nous gardons comme modèle final Conv1D_6 avec X_train.

Sur le jeu de test

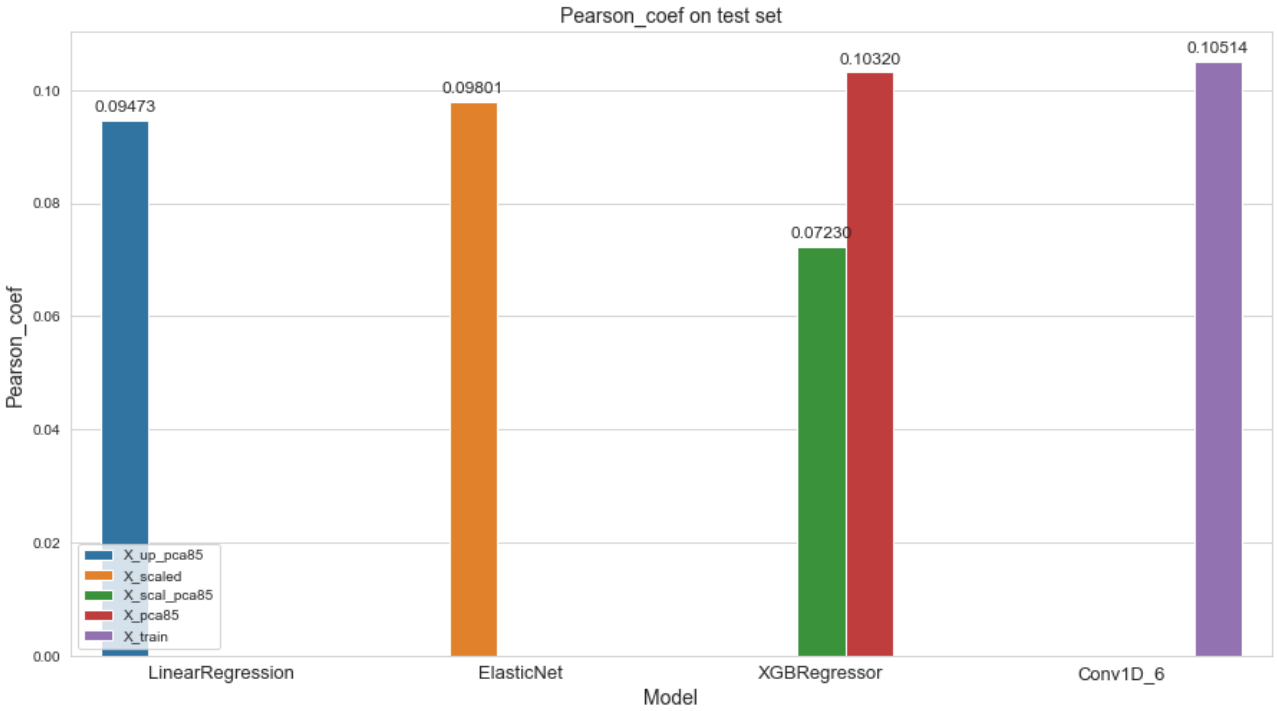
Nous comparons les cinq modèles retenus en évaluant la RMSE et le coefficient de Pearson. Visualisons dans un premier temps les résultats pour la RMSE :



COMPARAISON DES SCORES DE RMSE SUR LE JEU DE TEST

Les scores sont du même ordre de grandeur, il n'y a pas vraiment de très grandes différence (nous devons regarder à 10^{-3} pour comparer les scores). Nous constatons que ces scores sont meilleurs que ceux obtenus sur les jeux de validation. Pour la RMSE, le meilleur score est obtenu avec Conv1D_6, sur des données non préprocessées (pas de réduction de features, ni de scaling).

Voyons les résultats pour le coefficient de Pearson :



COMPARAISON DES SCORES DU COEFFICIENT DE CORRÉLATION DE PEARSON SUR LE JEU DE TEST

Les scores sont plus variés par rapport à la RMSE. Pour LinearRegression, ElasticNet XGBRegressor avec X_scal_pca85, les scores sont moins bons que sur le jeu de validation. Pour XGBRegressor, le modèle entrainé avec X_pca85 est toujours meilleur que celui entrainé sur X_scal_pca85 et nous notons un score meilleur sur le jeu de test. Parmi tous les modèles, là encore Conv1D_6 est le meilleur.

Conclusion

Concernant l'influence du preprocessing, sur ce dataset, nous avons vu que la réduction de dimension permettait d'obtenir de meilleurs résultats pour LinearRegression. Quant au scaling, il concerne plutôt ElasticNet. Il apporte de moins résultats pour LinearRegression et XGBRegressor.

En ne considérant que les features, le modèle basique de LinearRegression ne s'en sort pas si mal par rapport aux autres modèles. Les meilleurs résultats ont été obtenus avec un réseau de neurones Conv1D.

En pistes d'amélioration, nous pouvons envisager:

- d'autres architectures de réseaux de neurones Conv1D
- un callback réduisant le learning rate si une métrique sur le jeu de validation ne s'est pas améliorée sur 2-3 epochs
- l'intégration des variables time_id ou investment_id afin d'introduire une nouvelle dimension et d'utiliser des réseaux de neurones convolutionnels bidimensionnels (Conv2D).

Les notebooks relatifs à l'analyse et la modélisation sont disponibles sur Github.

Deux kernels Kaggle ont été publiés :

- un sur les problèmes engendrés lors du changement de data type pour réduire la taille (en octets) du dataset
- un sur la partie exploratoire, PCA et LinearRegression.

Annexe - Architecture des réseaux de neurones convolutionnels unidimensionnels testés

Conv1D_1[5]

Activation = ReLu
Dropout rate = 0.5
Optimizer = Adam avec un learning rate de 0.001
Nombre maximal d'epochs = 30
Earlystopping patience = 5

Layer (type)	Output Shape	Param #
Conv1D_1 (Conv1D)	(None, 294, 64)	512
dropout (Dropout)	(None, 294, 64)	0
Conv1D_2 (Conv1D)	(None, 292, 32)	6176
Conv1D_3 (Conv1D)	(None, 291, 16)	1040
MaxPooling1D (MaxPooling1D)	(None, 145, 16)	0
flatten (Flatten)	(None, 2320)	0
Dense_1 (Dense)	(None, 32)	74272
Dense_2 (Dense)	(None, 1)	33

Total params: 82,033
Trainable params: 82,033
Non-trainable params: 0

Conv1D_1_1

Mêmes choses que Conv1D sauf
Nombre maximal d'epochs = 50
Earlystopping patience = 10

Conv1D_2

Activation = LeakyReLU pour convolution et swish pour Dense_1
Dropout rate = 0.5
Optimizer = Adam avec un learning rate de 0.001
Nombre maximal d'epochs = 50
Earlystopping patience = 5

Layer (type)	Output Shape	Param #
Conv1D_1 (Conv1D)	(None, 294, 64)	512
LeakyReLU_1 (LeakyReLU)	(None, 294, 64)	0
dropout_6 (Dropout)	(None, 294, 64)	0
Conv1D_2 (Conv1D)	(None, 292, 32)	6176
LeakyReLU_2 (LeakyReLU)	(None, 292, 32)	0
Conv1D_3 (Conv1D)	(None, 291, 16)	1040
LeakyReLU_3 (LeakyReLU)	(None, 291, 16)	0
MaxPooling1D (MaxPooling1D)	(None, 145, 16)	0
flatten_6 (Flatten)	(None, 2320)	0
Dense_1 (Dense)	(None, 32)	74272
Dense_2 (Dense)	(None, 1)	33

Total params: 82,033
Trainable params: 82,033
Non-trainable params: 0

Conv1D_3[7]

Activation = LeakyReLU pour convolution et swish pour Dense_2 et 3
Dropout rate = 0.5
Optimizer = Adam avec un learning rate de 0.001
Nombre maximal d'epochs = 30
Earlystopping patience = 5

Conv1D_4

Mêmes choses que Conv1D_3 sauf utilisation de kernel_regularizer de type L2 pour Dense_2 et 3

Layer (type)	Output Shape	Param #
Conv1D_1 (Conv1D)	(None, 300, 64)	320
LeakyReLU_1 (LeakyReLU)	(None, 300, 64)	0
dropout_61 (Dropout)	(None, 300, 64)	0
Conv1D_2 (Conv1D)	(None, 98, 16)	8208
LeakyReLU_2 (LeakyReLU)	(None, 98, 16)	0
Conv1D_3 (Conv1D)	(None, 96, 16)	784
LeakyReLU_3 (LeakyReLU)	(None, 96, 16)	0
Conv1D_4 (Conv1D)	(None, 32, 32)	2080
LeakyReLU_4 (LeakyReLU)	(None, 32, 32)	0
Conv1D_5 (Conv1D)	(None, 8, 64)	8256
LeakyReLU_5 (LeakyReLU)	(None, 8, 64)	0
flatten_36 (Flatten)	(None, 512)	0
Dense_2 (Dense)	(None, 64)	32832
Dense_3 (Dense)	(None, 32)	2080
Dense_4 (Dense)	(None, 1)	33

Total params: 54,593
Trainable params: 54,593
Non-trainable params: 0

[7] inspiré de https://www.kaggle.com/code/ghostcxs/prediction-including-spatial-info-with-conv1d

Conv1D_5

Activation = ReLu
Dropout rate = 0.5
Optimizer = Adam avec un learning rate de 0.001
Nombre maximal d'epochs = 30
Earlystopping patience = 5
Kernel_regularizer de type L2

Layer (type)	Output Shape	Param #
Conv1D_1 (Conv1D)	(None, 300, 64)	320
dropout_63 (Dropout)	(None, 300, 64)	0
Conv1D_2 (Conv1D)	(None, 98, 16)	8208
Conv1D_3 (Conv1D)	(None, 96, 16)	784
Conv1D_4 (Conv1D)	(None, 32, 32)	2080
Conv1D_5 (Conv1D)	(None, 8, 64)	8256
flatten_38 (Flatten)	(None, 512)	0
Dense_2 (Dense)	(None, 64)	32832
Dense_3 (Dense)	(None, 32)	2080
Dense_4 (Dense)	(None, 1)	33

Total params: 54,593
Trainable params: 54,593
Non-trainable params: 0

Conv1D_6

Mêmes choses que Conv1D_5 mais sans kernel_regularizer
Une variante de ce modèle est utilisée avec X_scaled et Earlystopping patience = 10

Conv1D_7

Mêmes choses que Conv1D_6 sauf
Optimizer = Adam avec un learning rate de 0.0001