I. Exercice 1

1. Pseudo-code de l'algorithme Gradient Boosting

L'algorithme de Gradient Boosting a trois composantes principales :

- **Fonction de perte** qui permet d'estimer la capacité du modèle à faire des prédictions avec les données.
- **Apprenti faible** qui classe nos données avec un taux d'erreur élevé
- **Modèle additif** : approche itérative et séquentielle consistant à ajouter les apprenants faibles une étape à la fois.

Après chaque itération, le but est de se rapprocher du modèle final. En d'autres termes, chaque itération devrait réduire la valeur de notre fonction de perte. Le but serait de maximiser la fonction de perte déviance ici.

Le premier apprenant faible est initialisé à la moyenne des observations. Par la suite, nous calculons l'écart entre cette moyenne et la réalité appelé *premier résidu*. On appellera *résidu* l'écart entre la prédiction de l'algorithme en cours de création et la réalité. La particularité de l'algorithme Gradient Boosting est qu'il essaye de prédire à chaque étape non pas les données mais les résidus. Ainsi, le second apprenant faible est entrainé pour prédire le résidu du premier. Les prédictions du second apprenants faibles sont ensuite multipliés par un facteur *eta* inférieur à 1 (le *learning rate*). Cela permet de réduire la taille du pas pour augmenter la précision.

Les erreurs sont minimisées par l'algorithme de descente de gradient. On corrige seulement les prédictions pour lesquelles les résidus sont élevés, c'est ce que fait le prochain apprenant faible. On utilise la descente de gradient pour optimiser la fonction de perte en générant des algorithmes d'apprenants faibles de manière séquentielle et à ce que l'apprenant actuel soit toujours plus efficace que les précédents.

L'objectif étant d'écarter petit à petit les prédictions du modèle de la moyenne pour les rapprocher de la réalité. On peut ainsi interpréter le boosting comme un problème d'optimisation.

Pour chaque apprenant faibles, on exécute toujours la même procédure :

- A partir des dernières prédictions, on calcule les nouveaux résidus (écart entre la réalité et la prédiction)
- On entraine le nouvel apprenant faible pour prédire ces résidus
- On multiplie les prédictions de cet apprenant faible par un facteur *eta* inférieur à 1
- On obtient de nouvelles prédictions souvent meilleures que les précédentes

Pour connaitre la prédiction du gradient boosting sur une observation, on interroge chaque apprenant faible et on somme toutes les réponses obtenues pour former l'apprenant fort final.

Le boosting convertit un système d'apprenants faibles en un système unique d'apprentissage fort.

On peut voir ça comme un assemblage d'apprenants faibles qui prédisent les résidus et corrigent les erreurs des apprenants faibles précédents.

2. Présentation de la méthode Gradient Boosting

Le Gradient Boosting est une méthode ensembliste non linéaire extrêmement performante. L'algorithme Gradient Boosting est un problème d'apprentissage supervisé dans le cas de la régression et de la classification. L'apprentissage se fait de manière séquentielle en boostant la précision du précédent modèle. Pour le cas de la classification, on attribue une étiquette et pour la régression, une valeur numérique.

Les méthodes de Gradient Boosting sont des méthodes basées sur des modèles plus compliqués avec moins de paramètres à identifier lors des phases d'apprentissage. Le faible nombre de paramètres à apprendre permet d'accélérer les temps d'apprentissage.

Le but de ces méthodes est de construire des modèles complexes en combinant des ensembles de modèles plus simples. Le modèle ainsi obtenu est nommé prédicteur fort (*strong learner*) tandis que les sous-modèles le composant sont nommés prédicteurs faibles (*weak learner*).

Il reprend le principe général du Boosting, qui consiste à réaliser un méga-algorithme par itérations successives. Chaque itération visant à corriger l'erreur de la précédente. A chaque itération, la mise à jour du méta-algorithme s'effectue vers le gradient négatif de la fonction de coût choisie, d'où le terme « Gradient ». En effet, le gradient boosting est l'agrégation de deux méthodes : une descente de gradient combiné avec l'algorithme Boosting.

L'assemblage final est une combinaison linéaire de chacun des algorithmes faibles, ceux réalisant l'erreur la plus faible étant surpondérés lors de l'assemblage. La méthode du gradient boosting sert donc à renforcer un modèle qui produit des prédictions faibles.

Avantages :

- Hyperparamètres facilement optimisables (possibilité de séparer les variables non pertinentes de manière automatique)
- Interprétation possible
- Implémentation facile : algorithmes faciles à comprendre et qui apprennent de leurs erreurs
- Haute performance pour des données volumineuses (plus de 10 000 lignes) car faible complexité algorithmique
- Efficacité de calcul, possibilité de paralléliser les calculs
- Méthode très robuste, précision des résultats élevés, réduction de biais (combiner plusieurs apprenants faibles dans une méthode séquentielle améliore itérativement les observations)

Limites :

- Pas possible d'extrapoler, de faire des prédictions où les données ne sont pas stationnaires
- Vulnérables aux valeurs aberrantes

Il existe diverses librairies qui implémentent le principe des méthodes de gradient boosting comme XGBoost, CatBoost ou encore LightGBM.

3. Présentation de la méthode XGBoost

La méthode XGBoost est une variante de l'algorithme du Gradient Boosting. La particularité de cette méthode réside dans le type d'apprenant faible.

XGBoost permet de sélectionner des *features* suivant le gain. Mais cette méthode implique un tri pour chaque *feature* et un calcul du gain pour chaque critère de séparation. En conséquence, c'est une approche qui demande beaucoup de temps de calcul et qui est applicable que pour des jeux de données de taille raisonnable. Ainsi, XGBoost s'assure de ne garder que des bons apprenants faibles.

Une des faiblesses de XGBoost est qu'il ne supporte pas bien les features catégorielles. Il est nécessaire d'effectuer un pré-traitement. Plusieurs options sont possibles : *one_hot encoding* ou *get_dummies* de Pandas. Elle permet de transformer une colonne avec *n* catégories différentes prenant soit des 0 ou des 1. En revanche, l'inconvénient est que le nombre de colonne ajouté augmente en fonction du nombre de modalités différentes dans la variable.

De plus, XGBoost est optimisé pour rendre les calculs plus rapides. XGBoost traite les données en plusieurs blocs permettant de les trier beaucoup plus rapidement ainsi que de les traiter en parallèle.

XGBoost propose un panel d'hyperparamètres très important. Il est possible de rajouter différentes régularisations dans la fonction de perte limitant le sur-apprentissage. Ainsi, il est possible d'avoir un contrôle sur l'implémentation du Gradient Boosting. Ajouté à ceux-là, il y a le paramètre *booster* qui permet d'apporter des techniques du Deep Learning avec l'option *Dart*.

4. Contexte

Les modèles basés sur les arbres de décision sont très puissants. S'ils ne sont pas contraints par leur nombre et leur profondeur alors ils peuvent générer des prédictions exactes pour chaque ligne du jeu de données d'entraînement. Toutes les métriques seront donc très bonnes voire idéales, le modèle aura alors sur-appris. Malheureusement dans ce cas-là, le modèle ne donnera pas de bons résultats pour toutes les données qui n'appartiennent pas au jeu de données d'entraînement, même en appliquant une validation croisée. On dit alors que sa capacité à généraliser des données inconnues est très limité et l'intérêt du modèle aussi. Plusieurs solutions existent pour éviter le sur-apprentissage.

5. Solutions

a. Optimisation des hyperparamètres

La première solution et de jouer sur les hyperparamètres de l'algorithme de Gradient Boosting. Les deux principaux paramètres à prendre en compte sont la *profondeur maximale* de l'arbre et le *nombre d'estimateur*.

En effet, une profondeur maximale trop importante implique un nombre de feuilles qui peut être égal au nombre de lignes dans le jeu de données d'entraînement. Ce qui conduirait à ce que chaque feuille donne une valeur à apprendre.

De la même manière, un très grand nombre d'estimateur, c'est-à-dire d'arbre de décision même combiné à une faible profondeur peut conduire à du sur-apprentissage.

b. Optimisation des paramètres de régularisation

On peut également s'appuyer sur les paramètres de régularisation du modèle comme *lambda*, *gamma* et *alpha*.

c. Variantes de la méthode Gradient Boosting

Il existe différentes implémentations du Gradient Boosting pour les arbres de décision pour contourner ce problème de sur-apprentissage : XGBoost, CatBoost qui proposent de garder qu'un certain pourcentage du jeu de données initial en considérant soit les lignes, soit les colonnes. Il existe aussi la variante LightGBM. Ces variantes ont été testés sur une partie du jeu de données MNIST.

d. Mise en place du « early-stopping »

Un autre mécanisme permettant de limiter la complexité des modèles et donc le sur-apprentissage est le *early-stopping*. Cette méthode consiste à stopper l'ajout de nouveaux arbres si l'ensemble existant n'apporte aucun gain. Cela permet de ne prendre aucun risque d'avoir un modèle surentraîné sur les données.

L'algorithme utilisé évalue une ou plusieurs métriques à chaque ajout d'un nouvel arbre à l'ensemble. Si la ou les métriques n'évoluent plus pendant un nombre *n* fixé d'itérations, seuls les arbres ayant apporté un gain sont conservés.

6. Optimisation des hyperparamètres basés sur XGBoost

On cherche à optimiser les hyperparamètres d'un modèle pour trouver le plus rapidement possible la meilleure combinaison. Cela permettra de gagner du temps dans la mise en œuvre du modèle mais aussi de converger vers le modèle optimal non biaisé.

La difficulté réside dans la combinatoire à explorer qui est très grande. Par exemple le produit cartésien de toutes les possibilités peut s'étendre jusqu'à 100000. En pratique, tester toutes ces combinaisons est très rare, notamment si le jeu de donné d'entraînement est très grand et que le temps d'apprentissage est long.

La méthode que nous avons utilisée pour déterminer le meilleur modèle et de tester sur beaucoup de combinaisons possibles en utilisant la méthode *GridSearchCV*. Les temps de calcul sont très grands et cette méthode n'est pas très performante. On peut remarquer que le suffixe *cv* qui se retrouve dans les méthodes d'optimisation des hyperparamètres indique que l'évaluation du critère à optimiser se fait selon la méthode de validation croisée en calculant sur plusieurs jeux de données. Ainsi, plus ce nombre de jeu de données est important, plus la valeur du critère calculé en moyennant les scores obtenus sera proche de la valeur théorique.

Une variante de la méthode *GridSearchCV* est *HalvingGridSearch* de la librairie *scikit-learn* qui permet de gagner du temps en écartant rapidement les combinaisons les moins prometteuses.

On peut également s'appuyer sur le hasard pour explorer la combinatoire. Cela ne garantit pas que la meilleure combinaison va être retenue mais cela permet de garder le contrôle sur le nombre d'itérations et le temps de calcul. On peut utiliser la classe *RandomizedSearchCV* de *scikit-learn*.

Il existe une méthode d'approche de type substitut. Le principe consiste à construire un modèle capable de prédire le score associé à une configuration d'hyperparamètres. Des modèles de type *RandomForest* ou encore d'arbres de décision entraînés avec un gradient boosting s'appuie sur cette méthode. Nous implémenterons cette méthode avec XGBoost.

Lors de l'entraînement d'un arbre de décision avec une méthode de Gradient Boosting, seuls les poids associés aux feuilles et les critères de décision sont appris.

La structure de l'arbre, c'est à dire sa profondeur maximale et le nombre d'estimateur sont fixés. Ces valeurs et le nombre minimal d'échantillons par feuille, sont très importantes pour les performances finales du modèle.

Ces types de paramètres qui n'évoluent pas lors de l'apprentissage sont appelés *hyperparamètres*. Nous allons détailler et comprendre le rôle de chacun d'eux. En effet, ils ont un impact sur la structure de la forêt d'arbres générés. Ils influencent sur le temps d'apprentissage du modèle, c'est à dire sur la puissance de calcul pour les entraîner. Enfin, ils ont un impact fort sur les performances du modèle final.

Chaque paramètre a été illustré sur des exemples très simples pour comprendre leur fonctionnement. (*Voir script Exercice1_Partie2_Question2*)

i. Nombre d'estimateurs

Nous allons nous intéresser au premier hyperparamètre à configurer lors de l'entraînement d'une forêt d'arbres de décision : le *nombre d'estimateurs*.

Celui-ci correspond au nombre d'arbres qu'on va entraîner de manière séquentielle pour construire le prédicteur fort final.

Le choix de la valeur de ce paramètre s'appuie sur la notion de compromis biais/variance. On doit alors choisir un nombre suffisamment grand pour capturer la variabilité des données tout en évitant de sur-apprendre.

La manière la plus fiable de fixer ce paramètre est de procéder à plusieurs entraînements, en évaluant les performances sur le jeu de données d'entraînement et de test. Les entraînements peuvent se faire séquentiellement, à la main ou en automatique en utilisant la validation croisée.

ii. Profondeur maximale

L'autre hyperparamètre à prendre en compte sur la structure des arbres appris est la *profondeur maximale*. Ce paramètre indique la profondeur maximale que peut atteindre un arbre. C'est un maximum qui peut ne pas être atteint, si le nombre d'échantillons n'est pas suffisant pour ajouter un étage de plus ou si le gain apporté n'est pas suffisant. La profondeur maximale influe indirectement le nombre de nœuds de l'arbre et donc le nombre de feuilles.

Ce paramètre est à mettre en relation avec le nombre de données utilisées pour l'entraînement. En effet, les arbres de décision sont quasi systématiquement des arbres binaires, une profondeur de k étages impose un nombre de feuilles de 2^k.

7. Optimisation des paramètres d'apprentissage

En plus, des hyperparamètres qui jouent un rôle sur la construction des arbres de décision, il existe une série de paramètres qui jouent sur l'apprentissage.

Il est important d'identifier quel est le rôle du paramètre d'entraînement. Deux possibilités sont à envisager :

- Le paramètre intervient lors du calcul du gain.
  Le gain quantifie l'intérêt d'ajouter un nouvel étage à l'arbre de décision. Un gain important va conduire à cet ajout, tandis qu'un gain trop faible va stopper l'expansion de l'arbre. L'impact est donc *structurel*.
- Le paramètre intervient dans le calcul du poids optimal attaché à chaque feuille de l'arbre.
  L'impact se situe directement au niveau de la correction qui va être apportée. C'est donc directement la *prédiction* qui va être affectée.

i. Taux d'apprentissage : Learning rate

Le taux d'apprentissage est une valeur comprise entre 0 et 1. Elle permet de quantifier la correction. Il s'agit donc d'un paramètre qui intervient directement sur la valeur prédite.

Plus précisément, le poids calculé à l'aide des formules de l'algorithme gradient boosting va être multiplié par ce taux d'apprentissage. S'il vaut 0, aucune correction n'est appliquée et le modèle n'apprend rien. Au contraire, si ce taux vaut 1, la correction est entièrement appliquée.

Le but de ce taux d'apprentissage est d'éviter le sur-apprentissage, en ne transférant qu'une partie du poids optimal calculé. Il faut donc trouver le compromis entre un *learning rate* trop petit, qui implique un plus grand nombre d'estimateur et un *learning rate* proche de 1 qui risque d'entraîner un sur-apprentissage.

ii. Paramètre de régularisation : Gamma

Gamma est un paramètre de régularisation des méthodes de gradient boosting. Il a un impact sur l'apprentissage des arbres de décision et gère l'ajout de nouveaux nœuds sur la base du gain apporté. En effet, la formule suivante calcule le gain optimal :

$$\frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}-\frac{(G_L-G_R)^2}{H_L+H_R+\lambda}\right]-\gamma$$

Ce gain fait apparaitre le gradient *G*, la hessienne *H*, *T* le nombre de feuilles de l'arbre, le paramètre *lambda* et *gamma* pour les nœuds de droite R et gauche L.

Ce gain est calculé pour chaque possibilité de critères de séparation des données associées aux nœuds parent. Le meilleur critère est celui qui apporte plus de gain.

Le rôle de gamma se révèlent dans ce calcul du gain. En effet, si le gain sans la régularisation est inférieur à gamma, alors il est négatif et ne sera donc pas retenu. Ce paramètre modifie donc la structure de l'arbre généré.

Jouer sur *gamma* revient à contrôler la facilité avec laquelle un nœud est découpé en 2. Si *gamma* est nul, le découpage se fait automatiquement. Avec une valeur de *gamma* strictement supérieure à 0, le découpage n'a lieu que si le gain généré dépasse ce seuil.

Comme illustré dans le code, les prédictions obtenues pour le modèle sans régularisation mettent en avant des prédictions complétement différentes. Alors que le second modèle avec régularisation, mets en avant des prédictions identiques. On voit bien que régularisé avec gamma est une bonne stratégie pour éviter le sur-apprentissage.

iii. Paramètre de régularisation : lambda

Le second paramètre donnant le gain apporté par le découpage d'un nœud et l'ajout de deux nouvelles feuilles est *lambda*.

En effet, *lambda* est utilisé dans la formule calculant le poids optimal d'une feuille, comme nous montre la formule suivante :

$$w_j^* = -\frac{G_j}{H_j + \lambda}$$

Le paramètre *lambda* joue un rôle dans la structure de l'arbre de décision dans le calcul du gain et la valeur prédite, dans le calcul du poids d'une feuille. Le fait que *lambda* est au dénominateur cela revient donc à réduire le gain. Et ce d'autant plus si les échantillons associés à ce nœud sont peu nombreux. En effet, plus il y a de points de données attachées à ce nœud, plus l'impact de *lambda* devient négligeable.

Comme illustré dans le code, avec lambda égal à 1, cela permet de limiter le sur-apprentissage car il permet de réduire le nombre de feuilles. Le paramètre *lambda* a influé la structure de l'arbre. *Gamma* réalise une régularisation de type L2.

iv. Paramètre de régularisation : alpha

Enfin, il existe un dernier paramètre *alpha* qui permet d'influer la valeur des poids. Il fait intervenir la somme de la valeur absolue des poids dans la fonction de régularisation. *Alpha* réalise une régularisation de type L1.

Les régularisations de type L1 ont pour conséquence d'ajouter *alpha* au poids quand ce dernier est négatif, et à lui retirer *alpha* lorsqu'il est positif. Cela revient donc à le faire tendre vers 0.

Enfin, il existe des librairies pour optimiser les hyperparamètres comme avec scikit-learn, scikit-optimize, SMAC, Ray Tune.

II. Exercice 2

Pour prédire la réaction du traitement (variable binaire RD ou pCR), on a utilisé la base de données Breast Cancer.

On a commencé par nettoyer le jeu de données, en traitant les valeurs manquantes et en supprimant les individus et les variables qui n'apportent aucune information pertinente (comme les variables qui ont qu'une seule modalité). Pour limiter la suppression de données, on a cherché à compléter les données si possibles.

Le jeu de données comporte des variables catégorielles, on les a transformés en variables numériques à l'aide de la commande *pd.get_dummies* pour pouvoir utiliser les méthodes.

Nous avons observé qu'il n'y a pas de valeurs aberrantes. On a cherché à réduire la taille des variables explicatives en regardant leur corrélation avec la variable réponse mais on a constaté que toutes avaient un coefficient inférieur à 0.2. On les a toutes gardés.

On a également remarqué que les données étaient déséquilibrées (213 pour le RD et 55 pour le pCR). On s'est concentrée sur la métrique *rappel* pour mesurer la qualité des modèles avec ces données. On a remarqué que les métriques sans les poids n'étaient vraiment pas bonnes, on a préféré regarder les métriques avec pondérations. On pourrait tenter de rééquilibrer le jeu de données en réduisant la partie sur-représentée ou injecter des individus pour la classe pCR. Mais cela pourrait créer de la dépendance et donc biaiser les résultats. On pourrait aussi attribuer plus de poids aux erreurs de première espèce (prévoir RD alors qu'on est pCR). Ici, j'ai choisi de ne pas modifier le jeu de données et d'observer les métriques (rappel, précision) pour voir l'effet ou non de ces deux classes déséquilibrées.

On a ensuite normalisé les données avant d'appliquer les méthodes.

Pour la méthode PCR, on a constaté qu'en projetant les individus, les individus ne sont pas bien séparés pour répondre à notre problème et elle nécessite 15 composantes principales. Le modèle ne nous a pas permis de tirer de l'information sur nos données. Cette méthode n'est pas très adaptée ici.

Pour la méthode PLS, en utilisant la validation croisée, on a obtenu un modèle qui nécessite seulement 5 composantes principales. La méthode PLS semble plus appropriée en termes de composantes nécessaires.

Pour la méthode LASSO, on a cherché à optimiser le paramètre *alpha* pour augmenter la valeur du R2 et diminuer la valeur MSE. Celui-ci nous a donné de bons résultats.

Pour la méthode Sparse PCA en régression ne nous a pas permis de prédire mais seulement de visualiser les points avec 2 composantes. On aurait pu chercher à optimiser le nombre de composantes optimales pour avoir un meilleure visualisation.

Pour la méthode Kernel PCA, on a représenté les individus avec un premier modèle non optimal pour essayer de les séparer. On a utilisé la fonction de régression logistique donc avec un problème de classification. On a pu calculer différentes métriques (*matrice de confusion, précision, accuracy, recall, F1-score*) qu'on a pu comparer avec un modèle optimal mais les résultats sont similaires tant au niveau de la prédiction.

On a choisi de comparer notre jeu de données en utilisant les méthodes SVM et Random Forest pour obtenir de meilleure prédiction. Sur les SVM, on obtient de meilleures prédictions au niveau des scores

sur deux meilleurs modèles basés sur deux critères différents. Pareil, avec l'algorithme Random Forest. On a pu représenter les variables les plus importantes (3 se distinguent un peu plus que les autres), mais les variables explicatives restent très peu corrélées avec la variable à prédire.

Enfin, il semble difficile de tirer de l'information très pertinente, notamment, sélectionner et affirmer les variables explicatives très influentes sur le modèle pour prédire la réaction au traitement.

Le manque d'observations peut en être la conséquence au vu du nombre importante de variables (cas de la grande dimension).

III. Exercice 3 – Data Challenge : Chloé Douarec & Meriem Bencheikh

Tout d'abord, nous avons réalisé un pré-traitement sur les données. Nous avons construit *slp_train* et *slp_test* (données de pression) à l'aide de la méthode présentée dans le fichier benchmark fourni. Cela revient à étaler les données d'images présentes dans des listes, puisqu'initialement chaque image est stockée sous forme d'une liste dans le dataframe.

Nous avons créé deux fonctions *get_ech_complet* et *get_ech_i*, qui permettent de générer les échantillons pour la partie prédiction. La deuxième permet d'isoler les données en 20 sous-dataframes de façon à réaliser indépendamment les prédictions des 20 valeurs (10 valeurs par ville), tandis que la première fournit les données de façon à réaliser toutes les prédictions en même temps.

On a ensuite normalisé les données avant d'appliquer les méthodes.

Après avoir chargé les données, nous avons commencé par tester la variante LightGBM mais nous n'avons pas obtenu de bons résultats. Ceci est dû aux données non stationnaires, c'est un inconvénient de cette méthode.

Nous avons poursuivi avec la méthode Random Forest sans optimisation des paramètres. On a obtenu une très faible erreur d'entrainement 0.24 mais une erreur supérieure à 1 sur l'échantillon de test. Ceci est dû au sur-apprentissage. On a alors testé différentes techniques pour chercher le meilleur modèle en utilisant GridSearchCV et RandomizedSearchCV. Le meilleur modèle obtenu était le suivant :

RandomForestRegressor(max_depth=10, min_samples_leaf=2, min_samples_split=10, n_estimators=210)

On a obtenu une erreur de 0.726 sur l'échantillon d'apprentissage et la 0.95 sur l'échantillon test. Cette méthode n'est finalement pas meilleure que les K plus proches voisins.

On a cherché à utiliser nos données sur un autre modèle en utilisant les réseaux de neurones. On a obtenu une erreur de 1.09 sur celui d'entrainement. Nous n'avons pas testé sur le jeu de données test. On a fait évoluer le nombre de couches et le nombre de neurones par couches, ainsi que la valeur du paramètre *epoch*, les différentes fonctions d'activation (*linear*, *sigmoid*, *relu*) et la valeur du dropout (0.2 et 0.5) mais sans succès.

On s'est alors penché sur des modèles de régression (PLS, Ridge, Lasso) plus adaptée à nos données.

Sur la méthode Ridge, en utilisant la validation croisée, avec un paramètre optimal *alpha* de 0.016 et un fit intercept à False, on a obtenu une erreur d'entrainement de 1.71. Cette méthode n'est pas du tout adaptée à nos données surement parce qu'elle est linéaire. Nos données ont besoin d'un modèle de régression non linéaire.

On s'est ensuite intéressée à la méthode PLS. On a d'abord testé avec 19 composantes et on a obtenu un score d'entrainement de 0.450. Sur l'échantillon test, le score était de 0.541. La méthode PLS nous a fournit un très bon score et est adaptée à nos données temporelles. On a cherché à optimiser la méthode en déterminant le nombre de composantes minimales qui minimise la métrique utilisée pour le challenge. On a représenté graphiquement l'allure de la courbe de score qui décroit, et on a pu

constater qu'à partir de 28, ça diminuait très légèrement (une forte de plateau au-delà), d'où le sur-apprentissage derrière en augmentant le nombre de composantes.

Nous avons réalisé plusieurs tests que nous pourrons regrouper dans un tableau :

Nombre de composantes PLS	Erreur d'entrainement	Score sur le site du challenge
19	0.47	0.5419
24	0.429	0.5240
26	0.4247	0.5210
28	0.4192	
30	0.4126	0.520
31	0.4096	0.5185
32	0.4078	~~0.5238~~

Le meilleur score que nous avons obtenu avec la méthode PLS, c'est avec 31 composantes.

Enfin, on a également testé avec la méthode LASSO pour voir si on obtenait de meilleurs résultats que sur la méthode PLS.

Après de nombreux tests, on s'est rapproché de la valeur optimale du coefficient *alpha* pour éviter le sur-apprentissage. Nous avons obtenu une valeur de 0.011 avec un score sur l'échantillon test de 0.4866 et également une valeur de alpha de 0.018 qui donne un léger meilleur résultat.

Alpha	Erreur d'entrainement	Score sur le site du challenge
0.008		0.4886
0.01	0.38	0.4871
0.011	0.3887	0.4866
0.018	0.4	0.4866
0.02		0.4873

Nous avons réussi à trouver une méthode le LASSO qui est meilleure que la méthode PLS. En effet, les méthodes de régression non linéaire nous ont permis d'obtenir de très bons résultats.