Projet Modèles Linéaires et Dérivée

Data Challenge

Prédire la tendance de la production de pétrole brut

Par Société Générale

Authors :

Ms. Fadoua BOUSALIM
Ms. Ndeye Gninde GUEYE
M. Matthieu PARIZET

Supervisor :

Ms. Agathe GUILLOUX

Janvier 2018

Table des matières

1 Introduction

La production de pétrole brut correspond aux quantités de pétrole extraites du sous-sol après élimination des matières inertes ou des impuretés qu'ils contenait. Elle représente l'un des principaux indicateurs du marché des ressources naturelles.

Comprendre la variation de la production par région aide à prédire l'évolution du prix du pétrole dans ces régions et cela pour les différentes qualités du pétrole brut. Ces indicateurs peuvent être très utiles pour les équipes de la SOCIETE GENERALE, car elles permettront par exemple de mettre en place une liste de prospection et d'anticiper sur les besoins des clients.

L'objectif de ce challenge est de prédire la probabilité d'augmentation de la production de pétrole brut par trimestre et par pays à partir de plusieurs indicateurs recueillis au cours de l'année 2016. Pour cela, nous avons choisi de tester différents algorithmes et de comparer les scores obtenus afin d'améliorer les prévisions sur l'échantillon test.

Nous commencerons le rapport par une étude exploratoire des données, puis par le choix des modèles de prédiction. Ensuite, nous ferons une pipeline résumant la démarche suivie avant de conclure.

2 Les données

2.1 Description des données

L'échantillon d'entraînement contient des données sur 10159 lignes répartis entre 76 pays et 123 colonnes représentant les variables :

- **ID**
- **Month**
- **Country**
- **1-diffClosing stocks(kmt)** : niveau du stock primaire à la fin du mois 1 (janvier) dans les territoires nationaux.
- **1-diffExports(kmt)** et **1-diffImports(kmt)** : quantité de pétrole ayant traversé physiquement les frontières internationales en janvier.
- **1-diffRefinery intake(kmt)** : quantité de pétrole observé pour entrer dans le processus de raffinage en janvier.
- **1-diffWTI** : valeur correspondant au cours de clôture du dernier jour ouvrable de janvier.
- **1-diffSumClosing stocks(kmt)** : somme des lignes de 1-diffClosing stocks(kmt).
- **1-diffSumExports(kmt)** : somme de lignes de 1-diffExports(kmt).
- **1-diffSumImports(kmt)** :somme de lignes de 1-diffImports(kmt).
- **1-diffSumProduction(kmt)** : somme de lignes de 1-diffWTI.
- **1-diffSumRefinery intake(kmt)** : somme de lignes de 1-diffRefinery intake(kmt).

Il est également indiqué si la production augmente le trimestre suivant (1 si elle augmente, 0 sinon). Le but est alors de prévoir l'augmentation ou non d'un second échantillon test contenant les mêmes données.

L'échantillon test contient le même nombre de variables avec 2000 lignes représentant les données les plus récentes.

Nous avons calculé différentes mesures statistiques : la moyenne, l'écart type, etc. Puis nous nous sommes intéressés aux variables paire par paire pour savoir lesquelles sont les

plus corrélées.

```
In [6]: a=pd.DataFrame(np.abs(corr)>0.8)
        a
```

	month	country	1_diffExports(kmt)	1_diffImports(kmt)	1_diffRefinery intake(kmt)	1_diffSumClosing stocks(kmt)	1_diffSumExports(kmt)
month	True	False	False	False	False	False	False
country	False	True	False	False	False	False	False
1_diffExports(kmt)	False	False	True	False	False	False	False
1_diffImports(kmt)	False	False	False	True	False	False	False
1_diffRefinery intake(kmt)	False	False	False	False	True	False	False
1_diffSumClosing stocks(kmt)	False	False	False	False	False	True	False
1_diffSumExports(kmt)	False	False	False	False	False	False	True
1_diffSumImports(kmt)	False	False	False	False	False	False	False
1_diffSumProduction(kmt)	False	False	False	False	False	False	True
1_diffSumRefinery intake(kmt)	False	False	False	False	False	False	False

FIGURE 1 – Matrice de corrélations

Ce qui nous a permis de conclure que les variables SumProduction,SumImports et SumExports sont corrélées pour tous les mois.

Après avoir étudié les données, nous pouvons rapidement mettre en place un modèle naïf basé sur l'augmentation ou la diminution de la production de pétrole par mois. En effet, on peut observer sur le graphe ci dessous que la production de pétrole à tendance à baisser.

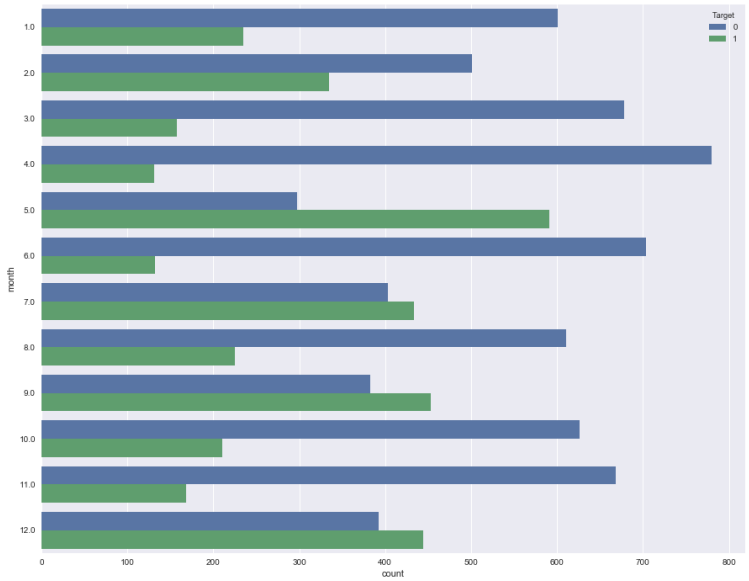

FIGURE 2 – La production en fonction du mois

Ce premier modèle nous a donné sans aucun traitement de données un score de 0.78 sur le jeu de test avec l'algorithme XGboost.

2.2 Pré-traitement des données

La première étape logiquement parlant est le traitement des données, aussi appelé feature engineering. Le but est d'adapter les variables pour qu'elles apportent le plus d'information possible. Cette étape se réalise par tâtonnement car on ne peut pas prévoir à l'avance les conséquences d'une certaine adaptation. Cette phase comporte surtout de l'intuition et de la recherche, il faut tester chaque idée et la conserver si elle augmente notre score.

Ainsi, nous avons essayé de faire un mapping des données tout d'abord. Nous avons regroupé les données par mois et par pays, ce qui a permis de créer des clusters selon ces deux paramètres. Malheureusement, ce mapping n'a pas permis d'améliorer les résultats (les résultats des différents modèles ne changeaient pas grâce au mapping). Ceci est peut être dû au fait qu'il y de fortes corrélations entre certaines variables. Une réduction des données s'est avérée nécessaire. Nous avons ainsi abandonné l'idée du mapping.

Ensuite, nous avons dû compléter les données lorsqu'elles n'étaient pas présentes pour certaines variables. Nous avons essayé de substituer ce manque par le biais de deux méthodes, que nous avons comparées :

AMELIA : C'est un package qui existe et qui a été développé afin de compléter les données manquantes. Il prend en entrée le fichier des données, et sort m (paramètre choisi) fichiers avec des valeurs remplies différentes (imputation multiple). La procédure utilisée est une combinaison de bootstrap sur les colonnes, et d'application de l'algorithme EM (algorithme itératif qui permet de trouver les paramètres du maximum de vraisemblance d'un modèle probabiliste lorsque ce dernier dépend de variables latentes non observables). Plus de détails sur le package peuvent être trouvés dans [3].
L'utilisation d'Amelia n'a pas permis d'améliorer le score, et ne parvenait pas faire mieux que l'extrême gradient de boosting basé sur les arbres (XGBtree, notation de R), qui lui traite mieux et automatiquement les données manquantes. Au meilleurs des cas, le score frôlait 0.790, ce qui reste inférieur au score 0.7934 de XGBtree .

K-NN : La deuxième technique utilisée est la complétion des données par l'algorithme des k plus proches voisins. Cela se fait facilement sur Python, mais requiert l'installation de beaucoup de packages. Cette technique a permis finalement de donner un meilleur score que le 0.7934 obtenu par l'extrême gradient de boosting basé sur les arbres. Avec ces données, l'extrême gradient de boosting atteignait à lui seul 0.81.

Notre choix final s'est porté sur la méthode des k plus proches voisins, avec k choisi égale à 10, puisque c'est le meilleur choix lorsqu'il s'agit de données volumineuses.

Après avoir résolu le problème des données manquantes, nous avons procédé à la réduction des données. Nous avons supprimé toutes les variables fortement corrélées, à savoir SumProduction,SumImports et SumExports, ainsi que toutes les variables relativement inutiles, c'est-à-dire dont la corrélation avec le Target est faible. Nous avons choisi un seuil de corrélation égale à 0.01. L'influence du seuil à été aussi étudiée, et il s'est avéré que tant que le seuil choisi est petit (inférieur à 0.01), l'effet de son changement est très négligeable (le score ne s'améliore pas). Donc on a jugé le seuil 0.01 comme bon, et on l'a gardé pour tous les tests.

Finalement, nous avons binarisé les deux variables "month" et "country".

3 Choix des modèles de prédiction

3.1 Critère de validation

Le nombre de soumissions sur le site Kaggle étant limité à 2 par jour pour ce challenge, nous devions trouver un moyen pour évaluer nous-même la qualité de nos prédictions afin de ne soumettre que les meilleures. Pour cela, nous avons mis en place un script de validation croisée 5-fold ou 3-fold, selon le temps de calcul nécessaire qui dépend du nombre de paramètres à tester, et de l'algorithme utilisé (certains algorithmes comme le Random Forest sont gourmands en temps de calcul). Celui-ci sépare les données de l'échantillon d'entraînement en 5 (respectivement 3) ensembles de taille égale ou presque. Un apprentissage est alors effectué sur 4/5 ((respectivement 2/3) de l'échantillon et une prédiction sur le 1/5 (respectivement 1/3) restant, et cela pour chaque combinaison possible. Nous regardons ensuite la moyenne des scores afin de valider ou non notre modèle. Évidemment, pour un algorithme donné, ce score de prédiction ne sera pas le même que celui obtenu sur Kaggle car les données sur lesquelles il est calculé ne sont pas les mêmes. Ce score est toutefois un bon indicateur car il est la plupart du temps un peu supérieur à celui de Kaggle mais reste très proche.

3.2 Algorithmes

3.2.1 Régression logistique

Dans ce cadre où nous devons attribuer des probabilités d'appartenance à deux classes, nous utilisons en première approche une régression logistique et estimons :

$$ln(\frac{\mathbb{P}(Y=1|X))}{1-\mathbb{P}(Y=1|X))} = \beta_0 + \sum_{i=1}^{n} \beta_i x_i \tag{1}$$

Sklearn l'implémente avec une pénalisation L_2, mais les résultats sont décevants. En effet, on ne dépasse pas le score publique de 0.66. Cela nous conduit naturellement à chercher un modèle dans une classe de famille plus étendue, ce qui nous amène à nous intéresser aux modèles GAM.

3.2.2 Modèles Additifs Généralisés (GAM)

Nous avons décidé d'utiliser et tester un modèles linéaire généralisé. Puisqu'il s'agit d'un problème de classification, le choix s'est porté automatiquement sur le modèle LogisticGAM qui est plus adapté (plus de détails théoriques dans la section 6.2).

Le LogisticGAM sur Python requiert des packages comme ScikitSparse qui permettent de faire de l'optimisation plus raidement. Malheureusement, l'installation de ces packages sur Windows a posé problème. Nous avons quand même décidé de lancer l'algorithme sans ces packages. Le temps de calcul a été long, mais le résultat final était meilleur que nos attentes : juste avec le LogisticGAM, on a obtenu un score de 0.8052 sur les données publiques, et de 0.8255 sur les données privées, ce qui est largement bien comme score.

3.2.3 Machines à vecteurs de supports (SVM)

Nous avons utilisé du SVM multiclasse avec différents paramètres afin d'optimiser.

Les paramètres considérés sont :

- C (paramètre de pénalité) : 0.001, 0.01, 0.1, 1, 10, 100, 1000
- penalty (norme de pénalisation pour la régularisation) : l1, l2
- multi-class (stratégie multiclasse) : ovr, crammer-singer

L'optimisation est faite par recherche sur grille avec validation croisée.

Les valeurs prédites sur l'échantillon de test permettent d'évaluer la précision de la méthode SVM.

```
[35]: lsvm = LinearSVC(loss="squared_hinge",dual=False)

      params = { 'C': np.logspace(-3, 3, 7), 'penalty':["l1","l2"], 'multi_class':["ovr","crammer_singer"] }

      gs_svm = GridSearchCV(lsvm, params, cv=3)
      gs_svm.fit(X_train_std, y_train)

      print("\nMeilleurs hyperparamètres svm sur le jeu d'entraînement:", gs_svm.best_params_)

      y_pred_svm = gs_svm.predict(X_test_std)

      acc_svm = metrics.accuracy_score(y_test, y_pred_svm)
      print("\nAccuracy de la méthode svm sur le jeu de test : %0.3f" % acc_svm )

      Meilleurs hyperparamètres svm sur le jeu d'entraînement: {'C': 0.001, 'multi_class': 'ovr', 'penalty': 'l2'}

      Accuracy de la méthode svm sur le jeu de test : 0.5121
```

FIGURE 3 – Algorithme SVM

Nous obtenons un score public de 0.65 sur le jeu de test après soumission sur le site ChallengeData.

3.2.4 Extreme Gradient de Boosting

Nous avons appliqué l'extême gradient de boosting sur nos données, avec validation croisée sur les paramètres. Le résultat obtenu était encore mieux que celui obtenu par le modèle additif généralisé : 0.81.

Nous remarquons donc que, parmi les modèles testés individuellement, le meilleur est XGBoost basé sur les arbres. On a donc envie de prendre en considération les modèles qui sont basés sur les arbres de décision, plus que les autres modèles. Cette intuition sera confirmée dans la section traitant du modèle final.

3.2.5 Réseaux neuronaux

Nous avons décidé de tester les réseaux neuronaux pour voir s'ils donnent de meilleurs résultats que les méthodes d'arbres. Après plusieurs validations croisées sur les différents paramètres afin de trouver un compromis et éviter le sur-apprentissage, les résultats restaient médiocres. Afin de pouvoir éviter le sur-apprentissage, nous avons utilisé la librairie sklearn.metrics qui permet de donner des résultats sur la précision et les taux des points bien/mal classifiés. Le meilleur score a été obtenu pour des paramètres qui donne les taux mentionnés dans la figure 5.

On remarque que les scores obtenus sur les données d'apprentissage sont bons, et on peut même obtenir des scores plus grands, qui atteignent 1. Ceci ne fait que dégrader le score sur les données test. Le compromis trouvé a permis d'obtenir un score de 0.739 sur les données publiques, et 0.764 sur les données privées. Ces scores étant faibles, nous ne nous sommes pas intéressés après à l'inclusion de cet algorithme dans le modèle final.

```
In [211]: from sklearn.metrics import classification_report,confusion_matrix

          #print(confusion_matrix(y,predictions))

In [212]: print(classification_report(y,predictions))
```

	precision	recall	f1-score	support
0	0.88	0.85	0.86	6644
1	0.73	0.77	0.75	3515
avg / total	0.83	0.82	0.82	10159

FIGURE 4 – Précision du réseau neuronal utilisé

3.3 Comparaison des modèles

Compte tenu des différents algorithmes vu en cours, nous allons créer une fonction d'évaluation afin de comparer les scores. Cette fonction sera appliquée sur les données pré-traitées. On obtient les résultats suivant :

```
In [6]: def Evaluation(clfs):
            for clf in clfs:
                clfs[clf]['score'] = cross_val_score(clfs[clf]['clf'], X_train, y_train, cv=3,sc
                print(clfs[clf]['name'] + ": %0.4f (+/- %0.4f)" % (clfs[clf]['score'].mean(), cl

In [7]: clfs = {}
        clfs['gbc'] = {'clf': GradientBoostingClassifier(), 'name': 'GradientBoostingClassifier'
        clfs['rf'] = {'clf': RandomForestClassifier( n_jobs=-1), 'name':'RandomForest'}
        clfs['tree'] = {'clf': DecisionTreeClassifier(), 'name':'DecisionTreeClassifier'}
        clfs['svc'] = {'clf': SVC(), 'name': 'SupportVectorClassifier'}
        clfs['knn'] = {'clf': KNeighborsClassifier(), 'name': 'KNeighborsClassifier'}
        clfs['xgb'] = {'clf' : XGBClassifier(), 'name':'XGBClassifier'}
        clfs['etc'] = {'clf' : ExtraTreesClassifier(), 'name':'ExtraTreesClassifier'}

        Evaluation(clfs)

        GradientBoostingClassifier: 0.7844 (+/- 0.0099)
        RandomForest: 0.7393 (+/- 0.0095)
        DecisionTreeClassifier: 0.6347 (+/- 0.0222)
        SupportVectorClassifier: 0.5121 (+/- 0.0027)
        KNeighborsClassifier: 0.6450 (+/- 0.0138)
        XGBClassifier: 0.7867 (+/- 0.0105)
        ExtraTreesClassifier: 0.7129 (+/- 0.0210)
```

FIGURE 5 – Score sur les différents modèles

Nous remarquons que le modèle XGboost donne de meilleurs résultats comparé aux autres modèles. Nous avons également un moin bon résultat que le même modèle ayant subi une une validation croisée. C'est ce qui nous poussera par la suite à optimiser ce modèle grâce au stacking afin d'améliorer notre score.

4 Pipeline finale

4.1 Modèle final

Face à l'impossibilité d'améliorer davantage le score des modèles précédents, il est naturel de ne plus s'arrêter à la prédiction d'un modèle mais de la considérer comme une information supplémentaire. Nous avons donc décidé d'utiliser la méthode de stacking avec des modèles basés sur des arbres de décision. Le stacking est un procédé qui consiste à appliquer un algorithme de machine learning à des classifieurs générés par un autre algorithme de machine learning, et construire ainsi des "couches de classification", où à chaque étape, un algorithme est appliqué sur les résultats du précédent. Ainsi avons-nous procédé en deux étapes :

- Entraîner des modèles intermédiaires sur les données complétées et sélectionnées
- Entraîner un modèle final sur ces prédictions, et utiliser ses résultats comme prédiction finale

Le choix des modèles intermédiaires et du modèle final se sont fait empiriquement, en sélectionnant pour les premiers niveaux des modèles affichant des performances convenables sur les données d'entraînement et dont les paramètres à ajuster étaient les mieux appréhendés. Enfin, le modèle final retenu est le boosting car c'est celui ci qui donnait les meilleurs résultats sur les données d'entraînement, et c'est aussi celui dont les paramètres sont les plus rapides à ajuster grâce à la vitesse d'exécution de XGBoost. La pipeline finale se résume donc ainsi :

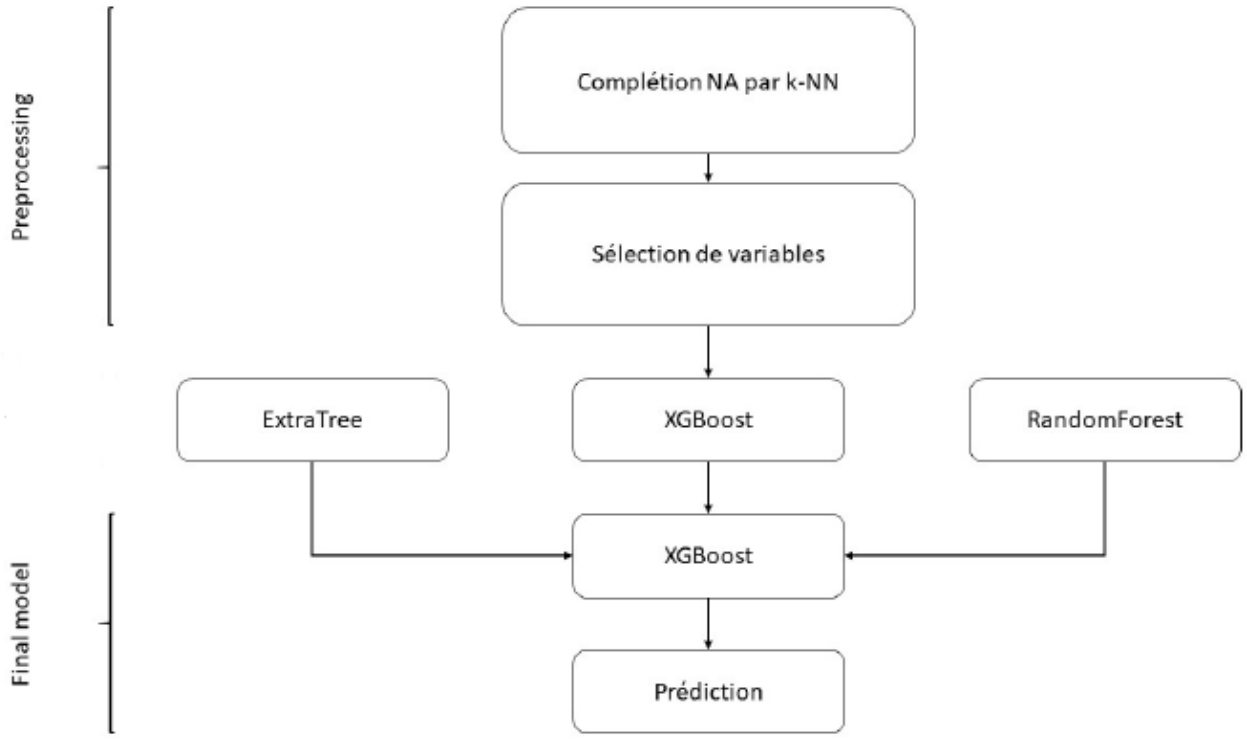

FIGURE 6 – Pipeline finale

4.2 Résultats finaux

Pour les trois premiers modèles, nous jouons sur les paramètres suivants :

- ExtratreeRegressor : nous explorons les valeurs de max_depth, $max\_features$, min_sampls_split, $min\_samples\_leaf$, et $n\_estimators$ pour arriver à un score moyen de 0.83 pour la validation croisée
- RandomForestRegressor : nous explorons les valeurs de $n\_estimators$ pour arriver a un score moyen de 0.815 pour la validation croisée
- XGBRegressor : pour arriver a un score moyen de 0.82 pour la validation croisée nous explorons successivement les valeurs de :
  1 $learning\_rate$ et $n\_estimators$
  2 max_depth et min_child_weight
  3 $learning\_rate$ et $n\_estimators$
  4 $colsample_bytree$ et $subsample$
  5 reg_alpha

Une fois ces trois modèles entraînés, nous ajustons un XGBRegressor sur leurs prédictions, en appliquons le même procédé que ci-dessus pour ajuster ses paramètres et obtenir un score moyen de 0.84 pour la validation croisée et un score public de 0.836. L'amélioration des capacités du modèles aurait sans doute nécessité de sélectionner plus finement les modèles utilisés. En effet, il semble naturel de vouloir combiner des modèles ayant des forces et des faiblesses complémentaires, c'est à dire se trompant sur des points structurellement différents. C'est pourquoi notre choix s'est porté sur des modèles basés sur les arbres de décision uniquement.

Afin d'arriver au meilleur score, qui était de 0.836, nous avons dû faire beaucoup de tests (qui prenaient souvent plusieurs heures vu le nombre de validations croisées effectuées). Les progressions peuvent être résumées comme suit :

- Premier modèle de stacking avec Random Forest, Extra Trees, et XGBoost avec XGBoost comme modèle utilisé à la fin aussi. Le score obtenu était de 0.833
- Une second couche de Random Forest a été rajoutée à la procédure de stacking. LE score obtenu était de 0.835
- Le premier modèle a été gardé, mais la validation croisée des trois régresseurs a été encore plus poussée. Le score obtenu état de 0.836

Ceci résume les étapes qui ont abouti à une amélioration du score. Il y a eu des tests qui n'ont pas permis d'améliorer le score (on a essayé de changer le régresseur final, ou rajouter d'autres couches de stacking).

Remarque : On avait dit avant que le LogisticGAM donnait un bon résultat à lui seul (0.805 est le score publique et 0.825 le score privé). On pourrait se dire que cela peut être une bonne idée de le rajouter au stacking. Nous avons eu cette idée, et elle a été testée. Le score obtenu était inférieur à 0.832. Le résultat a donc confirmé notre intuition au départ : les méthodes basées sur les arbres sont plus adaptées à notre problème.

5 Discussion et conclusion

Dans ce projet nous nous sommes intéressés à la prédiction de production de pétrole brut. Ces données sont connues pour avoir une grande dimensionnalité et souvent peu d'exemples d'apprentissage.

Nous avons commencé par faire une exploration des données. Principalement, afin de révéler les corrélations soit entre les variables, soit avec la classe à prédire. Nous avons

conclu après cela qu'une sélection de variables était primordiale afin d'entraîner des modèles performants.

D'une part, l'extrême gradient de boosting sans réduction nous donnait un score de 0.79. En y rajoutant le mapping de données nous avons obtenu un sore de 0.80. Cette méthode est plus lente que celle du GAM qui est à préférer dans ce cas. La lenteur du gradient de boosting était dû au stacking sans grande amélioration par rapport au GAM qui est bien plus simple.

D'autre part, la réduction de données est meilleure que le mapping dans notre cas, ce qui est probablement dû à une forte corrélation entre les variables. La réduction de données exécutée puis XGboost nous a permis d'obtenir un score de 0.81. C'est la raison pour laquelle la réduction a été la méthode utilisée dans le modèle final de stacking.

Après le pré-traitement des données, l'extrême gradient de boosting était le meilleur. Des méthodes basées sur les arbres de décision ont ainsi été préférées. En faisant du stacking avec un modèle composé de RandomForest, ExtraTrees, et XGBoost, nous obtenons un score de 0.833. Un meilleur score égal à 0.836 sur les données publiques, et 0.85342 sur les données privées, est obtenu en réalisant une validation croisée avancée sur les différents modèles. Ce score nous a permis d'être classés $6^{\text{ièmes}}$ pour les score publique.

L'avantage du modèle final que nous avons choisi, est qu'il est facilement adaptable à d'autres données. Selon ce qui marche le mieux sur chaque échantillon, les modèles utilisés dans le stacking peuvent être adaptés ou changés par d'autres. La validation croisée reste aussi adaptable selon les cas. La démarche et le modèle restent donc facilement généralisables.

À travers ce sujet, nous avons été exposés aux problématiques concrètes qui se posent régulièrement en machine learning, telles que la taille exubérante des données, la puissance limitée de nos outils de travail ou encore le nombre important de classes pour le critère à prédire. Ce fut difficilement gérable au commencement du projet mais nous avons ensuite réussi à contourner habilement ces problèmes, notamment en exerçant notre algorithme sur un sous-ensemble des données et non sur son intégralité. Évidemment, cela réduisait l'information disponible et diminuait donc la qualité possible de nos algorithmes, cependant cela nous a permis un premier travail de base sur les données et la manière de les utiliser. L'utilisation de packages spécialisés de python nous a ensuite permis de travailler sur un ensemble de données plus conséquent et de finalement aboutir à notre meilleur score sur ChallengeData de 0.85342.

6 Annexe et principaux algorithmes utilisés

6.1 Régression logistique

La régression logistique est un modèle de classification. Les variables sont catégorielles. Le but est de modéliser la probabilité d'appartenance d'un individu à une catégorie «k». On considère $Y(\omega)$ la modalité de Y prise par un individu ω. $(X_1(\omega), ..., X_j(\omega))$ est la description d'un individu ω dans l'espace des variables explicatives.

$$\Pi_k(\omega) = \mathbb{P}(Y(\omega) = k|X(\omega)) \tag{2}$$

Avec $\sum_{k=1}^{n} \Pi_k = 1$

On a $(K-1)$ équations LOGIT :

$$LOGIT_k(\omega) = ln\Big(\frac{\Pi_k(\omega)}{\Pi_K(\omega)}\Big) = \beta_{0,k} + \sum_{i=1}^{n} \beta_{i,k} X_i(\omega) \tag{3}$$

On en déduit les $(K-1)$ probabilités d'affectation :

$$\Pi_k(\omega) = \frac{e^{LOGIT_k(\omega)}}{1 + \sum_{k=1}^{K-1} e^{LOGIT_k(\omega)}} \tag{4}$$

et :

$$\Pi_K(\omega) = 1 - \sum_{k=1}^{K-1} \Pi_k(\omega) \tag{5}$$

6.2 Logistic GAM

L'idée est similaire à la régression logistique, mais l'on va chercher à exprimer les probabilités d'appartenance aux classes dans un plus grand ensemble de fonctions, ainsi on cherche :

$$LOGIT_k(\omega) = ln\Big(\frac{\Pi_k(\omega)}{\Pi_K(\omega)}\Big) = \beta_{0,k} + \sum_{i=1}^{n} f_{i,k}(X_i(\omega)) \tag{6}$$

En autorisant les ajustements non paramétriques, les GAMs bien conçus permettent de bons ajustements aux données d'apprentissage avec des hypothèses non contraignantes. Les fonctions $f_{i,k}$ sont ensuite estimées par décomposition sur des bases de fonctions adaptées. Dans le package utilisé (pygam), elles sont construites via des B-splines pénalisés (combinaison linéaires de splines à support compact minimal).

6.3 SVM

Le Support Vector Machine est un algorithme géométrique. Étant donnés des points répartis en 2 classes (ou plus) dans un espace vectoriel de dimension au moins 2, il faut trouver un hyperplan affine de telle sorte que tous les points d'une même classe soient d'un même côté de l'hyperplan. Concrètement, il s'agit de maximiser $min\{l_i h_i d(x_i, H)\}$ où l vaut 1 ou -1 selon la classe du point, $d(x_i, H)$ est la distance du point x_i à l'hyperplan H , et h_i qui vaut aussi 1 ou -1 selon si le produit scalaire $< x, n_H >$ est positif ou non.

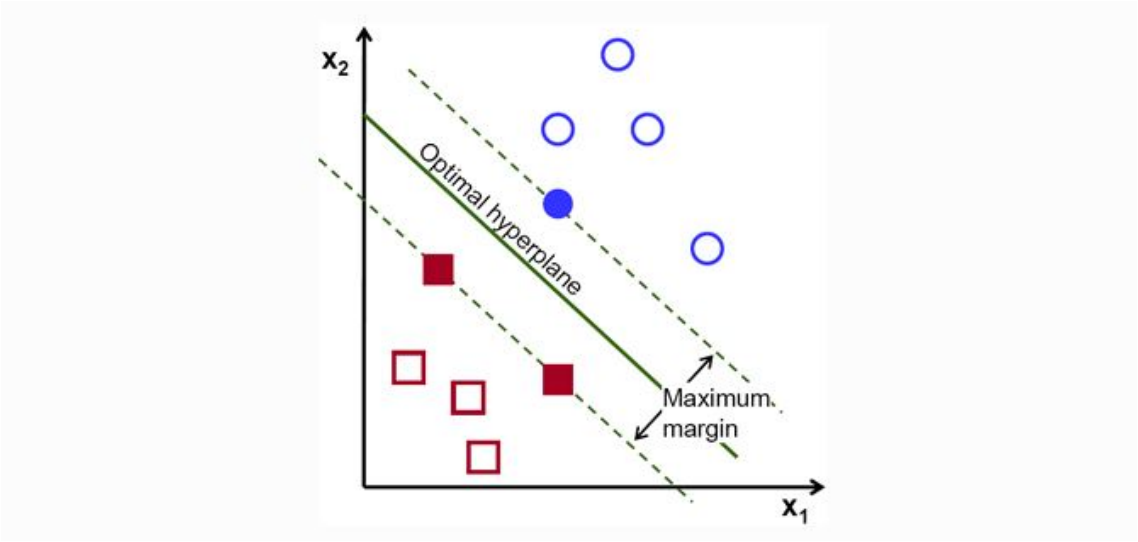

FIGURE 7 – Exemple de recherche de marge optimale entre le point le plus proche du plan et le plan lui-même

Cet algorithme a une complexité comprise entre $d.n^2$ et $d.n^3$ où d est le nombre de classes à distinguer et n le nombre de points à considérer. On peut souligner par la même occasion que cet algorithme ne passe donc pas très bien à l'échelle puisque que pour un grand nombre de données, le temps de calcul explose. On a pu le vérifier durant le projet puisque l'application du SVM aux données du challenge a été bien plus longue que les autres algorithmes utilisés. Il est aussi possible de choisir un noyau (linéaire par défaut) qui permet d'appliquer une transformation à l'espace d'origine. C'est ensuite dans ce nouvel espace que l'on va rechercher l'hyperplan optimal.

6.4 Random Forest

Cet algorithme est une version améliorée de celui des arbres de décision. Il effectue un apprentissage sur de multiples arbres de décision entraînés sur des sous-ensembles de données légèrement différentes. Chaque noeud dans un arbre de décision représente une condition sur un seul attribut (variable) qui divise le data set en deux sous ensembles dans le but que les exemples de même classe apparaissent dans le même sous ensemble.

Si une variable n'est pas importante, aucune diminution de précision ne sera constatée. Si en revanche la variable est importante, la précision du modèle sera largement diminuée. L'importance des variables est donc déterminée par l'effet des permutations de chaque variable sur la précision, et sont donc ordonnées par ordre décroissant.

6.5 Extreme Gradient de Boosting

Cette technique de boosting est majoritairement employée avec des arbres de décision. L'idée principale est là encore d'agréger plusieurs classificateurs ensembles mais en les créant itérativement. Ces classificateurs faibles sont généralement des fonctions simples et paramétrées, le plus souvent des arbres de décision. Le super-classificateur final est une pondération (par un vecteur w) de ces classificateurs faibles. Une approche pour construire ce super-classificateur est de :

- Prendre une pondération quelconque (poids w_i) de classificateurs faibles (para-

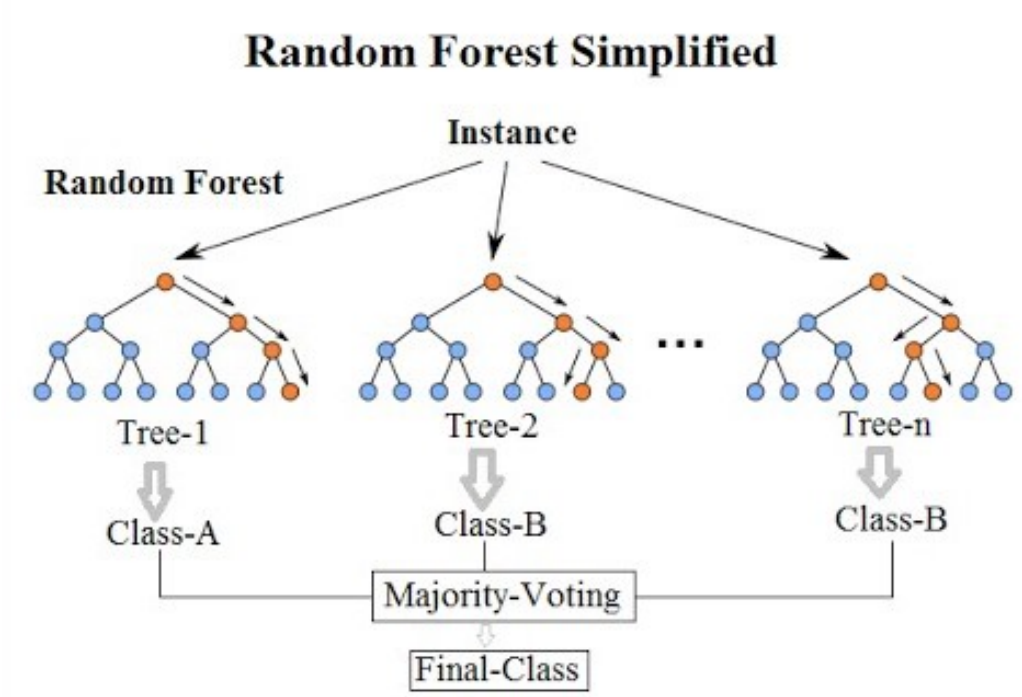

FIGURE 8 – Exemple d'apprentissage sur plusieurs arbres de décision

mètres a_i) et former son super-classificateur
- Calculer l'erreur induite par ce super-classificateur, et chercher le classificateur faible qui s'approche le plus de cette erreur (ce qui revient à le chercher dans l'espace des caractères)
- Retrancher le classificateur faible au super-classificateur tout en optimisant son poids par rapport à une fonction de perte
- Répéter le procédé itérativement. Le classificateur du gradient boosting est donc au final paramétré par les poids de pondération des différents classificateurs faibles, ainsi que par les paramètres des fonctions utilisées. Il s'agit donc d'explorer un espace de fonctions simples par une descente de gradient sur l'erreur

L'algorithme Extreme Gradient Boosting (XGBoost) est similaire à l'algorithme du Gradient boosting, néanmoins il est plus efficace et plus rapide puisqu'il est composé à la fois d'un modèle linéaire et des modèles d'arbres. Cela en plus de sa capacité à effectuer des calculs parallèles sur une seule machine. Essentiellement, les informations contenues dans chaque colonnes correspondant à une variable peuvent avoir des statistiques calculées en parallèle. L'importance des variables est calculée de la même manière que pour les forêts aléatoires, en calculant et moyennant sur les valeurs par laquelle une variable diminue l'impureté de l'arbre à chaque étape.

6.6 Extra Trees

Extremely randomized trees (Extra Trees) sont une autre classe de méthodes d'ensembles désignée spécifiquement pour les régresseurs basés sur les arbres de décisions.

- Chaque arbre est construit à partir de l'échantillon d'apprentissage

- A chaque noeud de test, la meilleur division est déterminée parmi des divisions aléatoires, and chaque division aléatoire est déterminée par une sélection aléatoire sans remise parmi un ensemble d'entrée (input) et un seuil

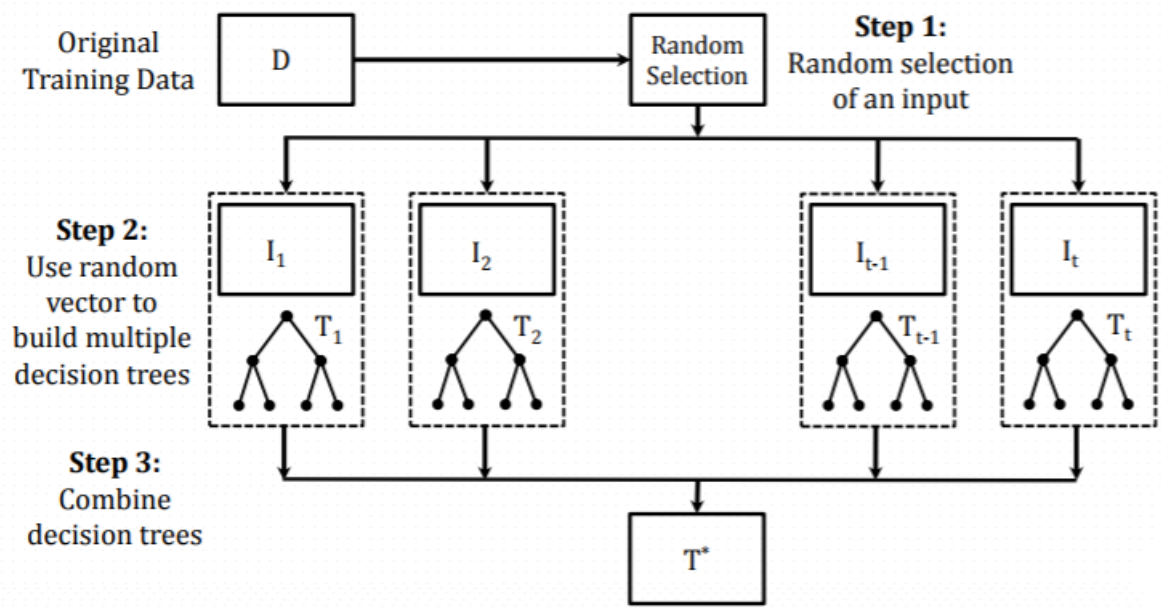

FIGURE 9 – Illustration du processus des Extra Trees

6.7 Réseaux neuronaux

Un réseau neuronal s'inspire du fonctionnement des neurones biologiques et prend corps dans un ordinateur sous forme d'un algorithme. Le réseau neuronal peut se modifier lui-même en fonction des résultats de ses actions, ce qui permet l'apprentissage et la résolution de problèmes sans algorithme, donc sans programmation classique. La large majorité des réseaux de neurones possède un algorithme « d'entraînement » qui consiste à modifier des poids en fonction d'un jeu de données présentées en entrée du réseau.

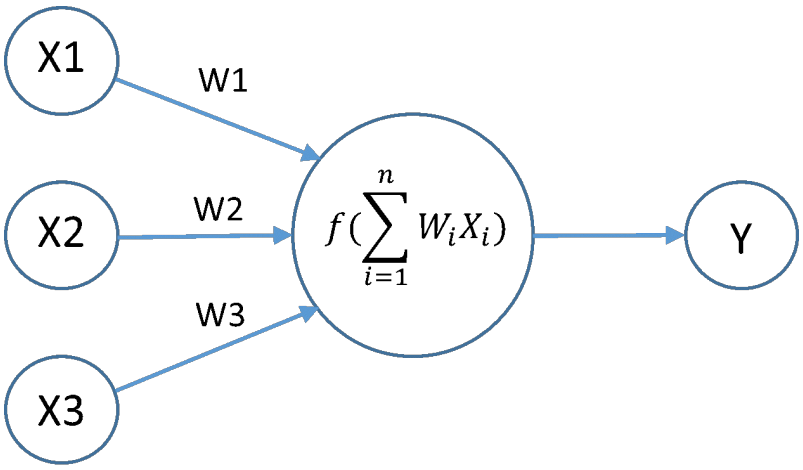

FIGURE 10 – Illustration du processus des Réseaux Neuronaux

6.8 Méthode des k plus proches voisins (K-Nearest Neighbours)

Dans le modèle des K-Nearest Neighbours, K étant choisi et fixé, l'algorithme va déterminer la classe d'appartenance d'une entrée comme étant la classe la plus représentée parmi ses k plus proches voisins selon une distance qu'il faut définir.

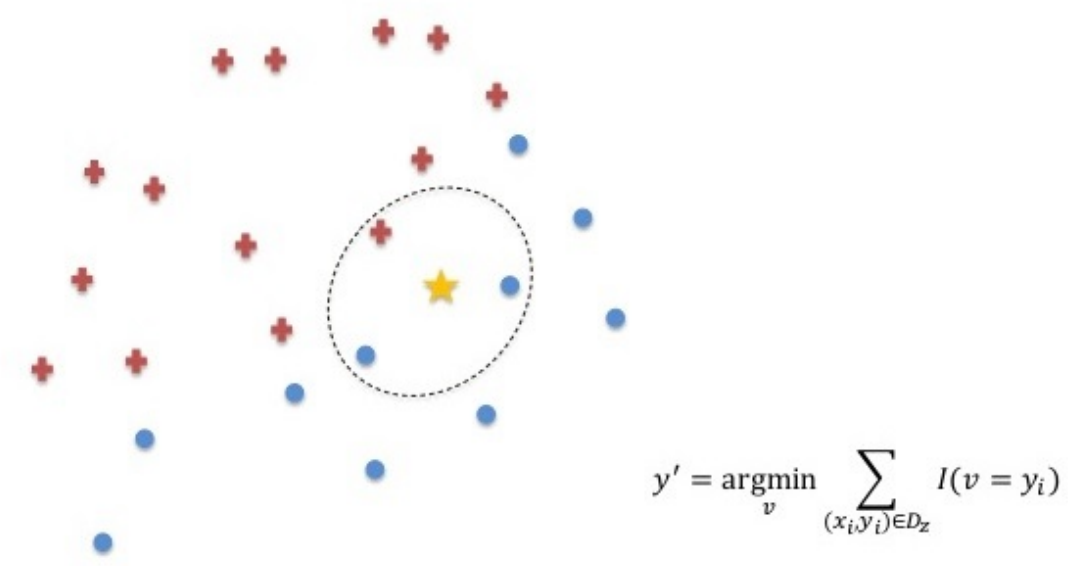

FIGURE 11 – Illustration de la méthode des K plus proches voisins

Références

[1] `https://data.oecd.org/fr/energy/production-de-petrole-brut.htm`

[2] `https://ensiwiki.ensimag.fr/`

[3] James Honaker, Gary King, and Matthew Blackwell. *Amelia* 𝕀𝕀 *: A Program for Missing Data*. Version 1.7.4. December 5, 2015.