Fouille de Données Massive

M2 Informatique - SISE (2022-2023)

Rapport Détection de Fraude

15/01/2023

HOUDE Titouan - IBGHI Samuel

Lien Github : https://github.com/Samibgh/FouilleDonneeMassive

Résumé

Ce document présente des méthodes de Machine Learning à employer dans un contexte de traitement de données massives et avec un jeu de données fortement déséquilibré. L'objectif final consiste à prédire si une transaction par chèque représente une fraude ou non. Plusieurs méthodes ont été testées et cinq méthodes sont présentées et détaillées dans ce document afin de déterminer si elles peuvent s'avérer pertinentes ou non pour cette problématique. Le modèle que nous avons essayé d'optimiser est le Gradient Boosting qui s'est trouvé prometteur.

Introduction

Dans le cadre du cours de Fouille de Données Massives enseigné par M. Guillaume Metzler, nous devions réaliser un projet portant sur l'apprentissage dans un contexte de données déséquilibrées avec comme application la détection de fraudes. L'objectif était d'obtenir le meilleur modèle possible en se basant sur le F-mesure créé à partir d'une matrice de confusion et dont la formule expliquée se trouve ci-dessous :

$$F = \frac{2TP}{2TP + FN + FP},$$

TP : fraude prédite fraude (vrai positif)

FN : fraude non identifiée (faux négatif)

FP : non fraude identifiée fraude (faux positif)

Les données sont issues d'une enseigne de la grande distribution ainsi que de certains organismes bancaires tels que la FNCI et la Banque de France. Chaque ligne représente une transaction effectuée par chèque dans un magasin de l'enseigne quelque part en France. Nous possédons dans le jeu de données 23 variables dont 3 sont des variables qualitatives. La variable à prédire était "FlagImpaye" qui représente l'acceptation ou le refus de la transaction. Le jeu de données était découpé en deux pour obtenir un jeu de

données d'apprentissage et un jeu de données de test. Cette étude représente un problème complexe en Machine Learning, car les données sont très déséquilibrées (99.4% de non fraudes contre 0.6% de fraudes). De plus, ce jeu de données est très volumineux (environ 4 millions de lignes), ce qui représente un deuxième challenge puisque deux paramètres seront à prendre en compte : la performance des modèles mais aussi leur temps d'exécution. En effet, le Machine Learning est un processus exploratoire et itératif, il n'est donc pas possible d'avoir des modèles qui apprennent pendant plusieurs heures puisque c'est bien l'ajustement des méthodes et l'optimisation des paramètres qui rendent un modèle robuste. Dans un premier temps, nous avons donc essayé de mettre en place des stratégies de parallélisation de calculs avec des librairies Python (Joblib, Dask, Sklearnex) pour tenter de réduire les temps de calculs, ainsi que de travailler sur un échantillon des données. Ensuite, nous avons fait recours à des méthodes d'échantillonnage (over-sampling et under-sampling) ainsi que des méthodes Cost-Sensitives (rajouter des poids sur l'importance de la classe minoritaire dans le jeu de données). Toutes ces problématiques freinent la performance des modèles, il était donc difficile d'obtenir un score dépassant les 8% en F-mesure. Le traitement de données aussi complexes demande d'utiliser aussi des algorithmes puissants comme le Gradient Boosting, Random Forest ou encore les SVM. Ajoutons à cela des moyens de booster les algorithmes comme AdaBoost pour réaliser du boosting et XGBoost pour réaliser du boosting sur les arbres, particulièrement sur des gros jeux de données. Il est également possible de combiner les modèles grâce au Bagging. Nous pouvons également tenter d'optimiser les paramètres, en utilisant des grilles de recherche, et en utilisant la cross validation avec les Kfold par exemple. La dernière stratégie est de réaliser une sélection de variables, ce qui permet de supprimer les variables qui n'apportent que très peu d'informations dans le modèle.

La section 2 sera consacrée à l'analyse de données, la section 3 présente les méthodes utilisées pour résoudre ce problème, la section 4 mettra en avant le protocole expérimental proposé et enfin la section 5 sera une conclusion de ce travail.

Analyse de données

Dans cette partie nous vous présenterons des statistiques descriptives sur le jeu de données. Nous relevons que le fichier comporte 23 variables dont 3 qualitatives. et est composé de 4 646 772 lignes.

Summary des données :

Vous trouverez ci-dessous un descriptif des variables qualitatives issues du jeu de données :

	ZIBZIN	IDAvisAutorisationCheque	FlagImpaye	DateTransaction	CodeDecision	Date	Heure_split
count	4646773	4646773	4646773	4646773	4646773	4646773	4646773
unique	1280126	3812039	2	3097881	5	302	49511
top	A075000041908023367242120	79817532	0	2017-03-04 17:18:31	0	2017-04-29	11:17:26
freq	217	2	4616778	14	3502786	54375	209

Ci-dessous les effectifs de la variables cible “FlagImpaye” :

```
0    4616778
1      29995
Name: FlagImpaye, dtype: int64
```

Méthodologie

Nous avons eu dans un premier temps, pour méthodologie de tester l’utilisation de divers algorithmes de pré-utilisation des modèles d’entraînement en les combinant . En effet, chaque méthode a été utilisée en les combinant entre elles et en ajoutant pour chaque combinaison d’un estimateur. L’objectif était d’analyser le F-mesure avec ses méthodes, mais aussi de vérifier le temps de calcul et de les mettre dans un tableau pour les comparer. Nous avons testé les estimateurs en standardisant les variables, en changeant la taille de l’échantillon ou encore en procédant à une sélection de variables. Une fois les analyses faites, nous avons sélectionné la meilleure combinaison possible. Les

estimateurs utilisés sont le Gradient Boosting, XGBoost, Random Forest, et les SVM à noyaux. Après la récupération de ce modèle, nous avons procédé au lancement de l'algorithme GridSearchCV avec cross-validation pour obtenir les meilleurs paramètres du modèle. Enfin, nous finirons par réaliser une sélection de variables sur le modèle pour améliorer ses performances.

Le gradient boosting est une technique d'apprentissage en ensemble qui combine plusieurs modèles faibles ou simplistes, tels que des arbres de décision, pour créer un modèle plus puissant et précis. La technique consiste à entraîner les modèles faibles en séquence, chaque modèle tentant de corriger les erreurs du modèle précédent. L'algorithme utilise le gradient de la fonction de perte pour guider l'entraînement de chaque modèle, d'où le nom de gradient boosting. Cela permet à l'algorithme de se concentrer sur les parties les plus difficiles de la fonction à apprendre. Il est performant sur des gros jeux de données car il est facilement parallélisable.

Algorithme de Gradient Boosting tel que présenté par [Friedman, 2000]

Input: Ensemble $S = \{\mathbf{x}_i, y_i\}_{i=1}^m$, une loss ℓ, nombre d'itérations T
Output: Un modèle sign $\left(H_0(\mathbf{x}) + \sum_{t=1}^T \alpha^t h_{a^t}(\mathbf{x})\right)$
begin
  Hypothèse initiale $H_0(\mathbf{x}_i) = \arg\min_{\rho \in \mathbb{R}} \sum_{i=1}^m \ell(y_i, \rho)\ \forall i = 1, \ldots, m$
  for $t = 1, ..., T$ **do**
    Calculer les pseudo-résidus :
    $\tilde{y}_i = -\dfrac{\partial \ell(y_i, H_{t-1}(\mathbf{x}_i))}{\partial H_{t-1}(\mathbf{x}_i)},\ \forall i = 1, ..., m$
    Apprendre un modèle pour les fitter :
    $a^t = \arg\min_{a \in \mathbb{R}^d} \sum_{i=1}^m (\tilde{y}_i - h_a(\mathbf{x}_i))^2$
    Apprendre le poids du classifieur :
    $\alpha_t = \arg\min_{a \in \mathbb{R}^+} \sum_{i=1}^m \ell(y_i, H_{t-1}(\mathbf{x}_i) + \alpha h_{a^t}(\mathbf{x}_i))$
    Mise à jour $H_t(\mathbf{x}_i) = H_{t-1}(\mathbf{x}_i) + \alpha_t h_{a^t}(\mathbf{x}_i)$
  return H^t

XGBoost (eXtreme Gradient Boosting) est une implémentation spécifique de l'algorithme de gradient boosting. Il a été développé pour améliorer les performances de l'algorithme de base en utilisant des techniques d'optimisation avancées telles que la régularisation, la parallélisation et l'échantillonnage aléatoire des données. Il est également doté d'une interface utilisateur conviviale et d'une API pour les différents langages de programmation. XGBoost est largement utilisé pour les tâches de classification et de régression et est souvent considéré comme l'un des meilleurs

algorithmes de gradient boosting disponibles. Il est utilisé dans des domaines tels que la reconnaissance d'images, la reconnaissance vocale, la détection de spam, la recommandation de produits, la détection de fraude, entre autres.. Son apprentissage est moins rapide sur des données volumineuses, c'est pourquoi il vaut mieux procéder au Stochastic Gradient Boosting où le principe est de faire varier de manière agressive les paramètres Subsample pour les échantillons de données ou encore la profondeur de l'arbre.

Voici un exemple de pseudo-code pour XGBoost :

1. Initialiser les poids des modèles faibles à 1/N

   Pour chaque itération :
   a. Entraîner un modèle faible en utilisant les poids actuels comme données d'importance
   b. Calculer les prédictions du modèle pour chaque point de données
   c. Calculer l'erreur de prédiction pour chaque point de données

2. Utiliser ces erreurs pour mettres à jour les poids des points de données
3. Combiner les modèles faibles en utilisant des poids pondérés pour obtenir la prédiction final

Ce pseudo-code est très simplifié et ne prend pas en compte les techniques d'optimisation avancées utilisées dans XGboost, telles que la régularisation et la parallélisation.

GridSearchCV est une technique utilisée pour sélectionner les meilleurs paramètres d'un modèle en utilisant une grille de valeurs de paramètres prédéfinies. Il est utilisé pour optimiser les performances d'un modèle en testant différentes combinaisons de paramètres pour trouver la combinaison qui donne les meilleurs résultats.

Le processus consiste à définir une grille de valeurs de paramètres à tester, puis à utiliser une méthode de validation croisée (comme K-fold) pour évaluer les performances du modèle pour chaque combinaison de paramètres. La combinaison de paramètres qui donne les meilleurs résultats est sélectionnée comme étant les meilleurs paramètres pour

le modèle.Il est important de noter que GridSearchCV peut être très coûteux en termes de temps de calcul en fonction de la taille de la grille de paramètres et de la complexité du modèle, il est donc souvent utilisé en combinaison avec d'autres techniques d'optimisation pour éviter les calculs inutiles.

L'oversampling et l'undersampling sont des techniques utilisées pour gérer les déséquilibres de classes dans les données d'apprentissage.

Oversampling consiste à ajouter des copies de points de données appartenant à une classe sous-représentée dans les données d'entraînement pour équilibrer les effectifs entre les classes. Cette technique est souvent utilisée lorsque les données positives sont rares par rapport aux données négatives.

Undersampling consiste à supprimer des points de données appartenant à une classe majoritaire dans les données d'entraînement pour équilibrer les effectifs entre les classes. Cette technique est souvent utilisée lorsque les données positives sont nombreuses par rapport aux données négatives.

Il est important de noter que ces techniques peuvent avoir un impact sur la performance des modèles et il est souvent nécessaire de tester différentes techniques pour trouver la meilleure solution pour un jeu de données donné. Il est également recommandé de combiner ces techniques avec d'autres techniques de traitement des données telles que la régularisation pour obtenir les meilleurs résultats.

Dans notre cas nous nous intéressons aux algorithmes BorderLineSmote qui vont imposer une règle de sélection sur les exemples de la classe minoritaire qui seront utilisés pour la génération de données et Adasyn qui est une version plus lisse du BorderLineSmote. On cherche cette fois-ci à concentrer la génération sur les exemples plus difficiles à classer.

Algorithm 1: Algorithme SMOTE [Chawla et al., 2002]

Input: Echantillon d'apprentissage S de taille $m = p + n$,
k : nombre de voisins, R taux d'exemples positif à générer
Output: Un échantillon S'
begin
  Poser $New = R \times p$: nombre d'exemples à générer Syn,
  sélectionner aléatoirement New exemples parmis p et noter $setind$
   leur indice
  for $i \in setind$ **do**
    chercher les $k-$pp positifs de l'exemple $\mathbf{p}_i$,
    sélectionner aléatoirement l'un des plus plus proches voisins
     rNN_i **for** $attributs\ attr\ de\ \mathbf{p}_i$ **do**
      choisir un nombre $\alpha \in [0, 1]$,
      poser $newattr = \alpha p_i[attr] + (1 - \alpha) rNN_i[attr]$
      poser $Syn_i[attr] = newattr$
    Poser ensuite $S' = S \cup Syn_i$
  Poser ensuite $S' = S$
return S'

Algorithme SMOTE, l'idée du BorderLine est très proche de celui-ci.

Algorithm 1: ADASYN

Input: Training dataset X_T, Hyper parameter $\beta \in [0,1]$, K=5
The i th sample in the minority sample x_i (i = 1, 2, 3, ···, m_s),
A random minority sample x_{zi} in K-nearest neighbors of x_i.
Output: Synthetic minority samples s_i , Oversampled training dataset X_{ADASYN}

1 Calculate the number of majority samples m_l and the number of minority samples m_s in Training dataset X_T
2 According to the formula $G = (m_l - m_s) \times \beta$ calculates the number of samples to be synthesized for minority class
3 **For each example $x_i \in$ minorityclass:**
4 Calculate Δi //the number of majority samples in K-nearest neighbors of minority Sample x_i
5 Calculate $r_i = \Delta i / K$ //the ratio of majority samples in K-nearest neighbors of minority Sample x_i
6 Standardize r_i through the formula $\hat{r}_i = r_i / \sum_{i=1}^{m_s} r_i$
7 Calculate $g_i = \hat{r}_i \times G$ //the number of new samples to be generated for each minority x_i
8 **Do the loop from 1 to g_i**
9 Using the formula $s_i = x_i + (x_{zi} - x_i) \times \gamma$ to synthesize data samples //γ is a random number: $\gamma \in [0,1]$
10 **End**
11 **End**
12 **Return** Oversampled training datasets X_{ADASYN}

Algorithme Adasyn

La sélection de variables permet de réduire en conséquence les variables présentes dans le jeu de données afin d'améliorer les performances du modèle.Celles supprimées font baisser les performances et nous gardons uniquement celles qui apportent de l'information.

Expérience

Présentation de données :

Nous avons commencé par définir la taille de l'échantillon de données à utiliser pour le modèle. Les tailles de ces échantillons varient entre 100000 données à 3899362 données (taille des données du fichier train). Nous avons ensuite procédé à des sélection de variables utilisant les algorithmes SelectFromModel et VarianceThreshold de Sklearn. La technique d'oversampling utilisée est celle du BorderLineSmote réalisé sur chaque échantillon. Les résultats seront inscrits dans un tableau. Pour chaque échantillon de données, dans la variable cible, les classes gardent la même proportion entre la classe minoritaire et celle majoritaire. Cependant lorsque nous utilisons l'oversampling, la classe minoritaire est rééquilibrée suivant un paramètre nommé "sampling_strategy" qui est un nombre variant de 0 à 1 et qui sert à définir l'information pour échantillonner les données. Le nombre de variables varie suivant l'algorithme choisi pour faire la sélection de variables, ou alors lorsque nous choisissons aucun algorithme, le nombre de variables reste le même qu'au départ.

Protocole expérimental :

Après des premiers tests pour comprendre l'enjeu de cette étude (temps d'exécution, difficulté de prédictions etc…), nous avons mis en place un tableau nous permettant de confronter les différents algorithmes, méthodes de sélection de variables, échantillons de données et leur temps d'exécution. Suivant les résultats obtenus, nous sélectionnions le meilleur modèle parmi l'ensemble de ceux testés pour ensuite l'améliorer et augmenter le score. Après cela, nous testons les hyperparamètres dans un GridSearchCV, en utilisant la cross-validation afin d'obtenir les meilleurs. Nous enchaînons sur une sélection de variables pour voir si les performances du modèle s'améliorent. Enfin, nous pourrons faire un Bagging permettant d'ajuster les classificateurs de base.

Résultats sans ré-échantillonnage

Voici les tableaux récapitulatifs des tests effectués. Ces tests ont été réalisés sur 3 échantillons de données (environ 2.5%, 10% et 25% du total de données disponibles)

Random Forest

Classifieur	Echantillon	Selection Variable	Performance (F-mesure)	Temps_execution (en secondes)
RandomForest	100000	NoProcessing	0.0	14.4
RandomForest	100000	SelectFromModel	0.0	12.61
RandomForest	100000	Variance	0.0	15.69
RandomForest	380000	NoProcessing	0.0	63.27
RandomForest	380000	SelectFromModel	0.0	53.07
RandomForest	380000	Variance	0.0	61.35
RandomForest	999999	NoProcessing	0.0	196.39
RandomForest	999999	SelectFromModel	0.0	182.76
RandomForest	999999	Variance	0.0	224.88

On peut voir que le Random Forest n'est pas performant dans ce contexte, un plus grand volume de données ou un échantillonnage pourrait lui permettre de le devenir. Il semblerait logique qu'une augmentation du nombre de données soit bénéfique au Random Forest.

XGBRF Classifier

Classifieur	Echantillon	Selection Variable	Performance (F-mesure)	Temps_execution (en secondes)
XGBRFClassifier	100000	NoProcessing	0.0	3.24
XGBRFClassifier	100000	SelectFromModel	0.0	2.68
XGBRFClassifier	100000	Variance	0.0	4.26
XGBRFClassifier	380000	NoProcessing	0.0	8.79
XGBRFClassifier	380000	SelectFromModel	0.0	7.35
XGBRFClassifier	380000	Variance	0.0	8.85
XGBRFClassifier	999999	NoProcessing	0.0	21.79
XGBRFClassifier	999999	SelectFromModel	0.0	16.94
XGBRFClassifier	999999	Variance	0.0	21.61

Le XGBRF se montre identique au Random Forest en termes de performance, il est cependant bien plus rapide. Bien que la rapidité soit un critère important comme énoncé précédemment, le niveau de performance est tellement bas qu'il n'est pas possible de s'en satisfaire.

Gradient Boosting

Classifieur	Echantillon	Selection Variable	Performance (F-mesure)	Temps_execution (en secondes)
GradientBoostingClassifier	100000	NoProcessing	0.028	13.15
GradientBoostingClassifier	100000	SelectFromModel	0.015	11.40
GradientBoostingClassifier	100000	Variance	0.0153	12.26
GradientBoostingClassifier	380000	NoProcessing	0.0018	49.80
GradientBoostingClassifier	380000	SelectFromModel	0.0144	42.58
GradientBoostingClassifier	380000	Variance	0.0169	45.82
GradientBoostingClassifier	999999	NoProcessing	0.0486	141.85
GradientBoostingClassifier	999999	SelectFromModel	0.0158	120.22
GradientBoostingClassifier	999999	Variance	0.0171	129.11

Le Gradient Boosting s'est révélé assez prometteur lors de ce test en termes de performance mais également de rapidité. Il semble donc très intéressant à explorer.

AdaBoost

Classifieur	Echantillon	Selection Variable	Performance (F-mesure)	Temps_execution (en secondes)
AdaBoostClassifier	100000	NoProcessing	0.0	4.22
AdaBoostClassifier	100000	SelectFromModel	0.0	3.71
AdaBoostClassifier	100000	Variance	0.0	4.5
AdaBoostClassifier	380000	NoProcessing	0.0	11.79
AdaBoostClassifier	380000	SelectFromModel	0.0	10.4
AdaBoostClassifier	380000	Variance	0.0	10.89
AdaBoostClassifier	999999	NoProcessing	0.0	30.62
AdaBoostClassifier	999999	SelectFromModel	0.0	25.71
AdaBoostClassifier	999999	Variance	0.0	27.65

Tout comme le Random Forest ou le XGBRF, AdaBoost s'est montré décevant lors de ce test.

Résultats avec ré-échantillonnage (Borderline Smote)

Pour ce deuxième test avec échantillonnage deux modèles ont été conservés : le Random Forest avec comme supposition qu'il serait bien meilleur sur un plus grand volume de données et le Gradient Boosting qui s'était montré prometteur. Un nouvel algorithme à été testé : le KNN.

KNeighborsClassifier

Classifieur	Echantillon	Selection Variable	Performance (F-mesure)	Sampling Strategy	Temps_execution (en secondes)
KNeighborsClassifier	100000	NoProcessing	0.0174	0.3	35.30
KNeighborsClassifier	100000	SelectFromModel	0.0177	0.3	5.54
KNeighborsClassifier	100000	Variance	0.0076	0.3	8.97
KNeighborsClassifier	380000	NoProcessing	0.0153	0.3	121.24
KNeighborsClassifier	380000	SelectFromModel	0.0135	0.3	5.83
KNeighborsClassifier	380000	Variance	0.0152	0.3	10.2
KNeighborsClassifier	999999	NoProcessing	0.0161	0.3	307.92
KNeighborsClassifier	999999	SelectFromModel	0.0142	0.3	6.38
KNeighborsClassifier	999999	Variance	0.0185	0.3	12.15
KNeighborsClassifier	3899362	NoProcessing	0.0171	0.3	1183.22
KNeighborsClassifier	3899362	SelectFromModel	0.0185	0.3	9.2
KNeighborsClassifier	3899362	Variance	0.0146	0.3	18.30
KNeighborsClassifier	100000	NoProcessing	0.0131	0.5	35.28
KNeighborsClassifier	100000	SelectFromModel	0.0175	0.5	5.54
KNeighborsClassifier	100000	Variance	0.0174	0.5	7.40
KNeighborsClassifier	380000	NoProcessing	0.0167	0.5	120.01
KNeighborsClassifier	380000	SelectFromModel	0.0079	0.5	5.75
KNeighborsClassifier	380000	Variance	0.013	0.5	9.50
KNeighborsClassifier	999999	NoProcessing	0.0198	0.5	309.63
KNeighborsClassifier	999999	SelectFromModel	0.017	0.5	6.20
KNeighborsClassifier	999999	Variance	0.0167	0.5	11.47
KNeighborsClassifier	3899362	NoProcessing	0.0186	0.5	1190.85

Les performances du KNN se sont montrées très régulières malgré des paramètres différents (nombre de données, sélection de variables, proportion d'échantillonnage). Malheureusement elles ne sont pas suffisamment bonnes pour être approfondies. En effet, le KNN ne dispose que d'un paramètre et ne semble donc pas optimisable. Ce type de problème est trop complexe pour que le KNN puisse être performant.

Gradient Boosting

Classifieur	Echantillon	Selection Variable	Performance (F-mesure)	Sampling Strategy	Temps_execution (en secondes)
GradientBoostingClassifier	100000	NoProcessing	0.0015	0.3	18.44
GradientBoostingClassifier	100000	SelectFromModel	0.0197	0.3	13.84
GradientBoostingClassifier	100000	Variance	0.002	0.3	16.14
GradientBoostingClassifier	380000	NoProcessing	0.0141	0.3	71.71
GradientBoostingClassifier	380000	SelectFromModel	0.0044	0.3	53.49
GradientBoostingClassifier	380000	Variance	0.0	0.3	61.94
GradientBoostingClassifier	999999	NoProcessing	0.0003	0.3	206.29
GradientBoostingClassifier	999999	SelectFromModel	0.0153	0.3	153.03
GradientBoostingClassifier	999999	Variance	0.0207	0.3	173.45
GradientBoostingClassifier	3899362	NoProcessing	0.02	0.3	953.10
GradientBoostingClassifier	3899362	SelectFromModel	0.0146	0.3	688.57
GradientBoostingClassifier	3899362	Variance	0.0201	0.3	770.70
GradientBoostingClassifier	100000	NoProcessing	0.0064	0.5	20.17
GradientBoostingClassifier	100000	SelectFromModel	0.0174	0.5	14.94
GradientBoostingClassifier	100000	Variance	0.0147	0.5	18.5
GradientBoostingClassifier	380000	NoProcessing	0.0424	0.5	79.49
GradientBoostingClassifier	380000	SelectFromModel	0.0174	0.5	58.09
GradientBoostingClassifier	380000	Variance	0.0018	0.5	69.11
GradientBoostingClassifier	999999	NoProcessing	0.005	0.5	228.21
GradientBoostingClassifier	999999	SelectFromModel	0.0208	0.5	165.95
GradientBoostingClassifier	999999	Variance	0.016	0.5	201.96
GradientBoostingClassifier	3899362	NoProcessing	0.0	0.5	1069.42
GradientBoostingClassifier	3899362	NoProcessing	0.0003	0.3	1551.80

La méthode d'échantillonnage n'a pas été bénéfique au Gradient Boosting, perdant en performance malgré quelques tests sur la totalité des données disponibles.

Random Forest

Classifieur	Echantillon	Selection Variable	Performance (F-mesure)	Sampling Strategy	Temps_execution (en secondes)
RandomForest	100000	NoProcessing	0.0114	0.3	33.10
RandomForest	100000	SelectFromModel	0.0223	0.3	27.10
RandomForest	100000	Variance	0.0041	0.3	33.14
RandomForest	380000	NoProcessing	0.0105	0.3	152.51
RandomForest	380000	SelectFromModel	0.0085	0.3	114.46
RandomForest	380000	Variance	0.01	0.3	145.57
RandomForest	999999	NoProcessing	0.0109	0.3	505.52
RandomForest	999999	SelectFromModel	0.0182	0.3	355.06
RandomForest	999999	Variance	0.0133	0.3	437.48
RandomForest	3899362	NoProcessing	0.0125	0.3	2628.39
RandomForest	3899362	SelectFromModel	0.0145	0.3	2045.88
RandomForest	3899362	Variance	0.0248	0.3	2336.78
RandomForest	100000	NoProcessing	0.0148	0.5	36.97
RandomForest	100000	SelectFromModel	0.0161	0.5	29.66
RandomForest	100000	Variance	0.0122	0.5	34.92
RandomForest	380000	NoProcessing	0.0155	0.5	161.92
RandomForest	380000	SelectFromModel	0.0212	0.5	117.71
RandomForest	380000	Variance	0.0164	0.5	142.51
RandomForest	999999	NoProcessing	0.0182	0.5	528.87
RandomForest	999999	SelectFromModel	0.0156	0.5	415.26
RandomForest	999999	Variance	0.0157	0.5	452.39

Comme on pouvait l'imaginer, les performances du Random Forest sont meilleures sur un nombre de données supérieur. Bien qu'elles soient meilleures, elles ne sont pas non plus excellentes et l'algorithme est plus lent que le Gradient Boosting.

En vu des résultats des différents tableaux, nous voyons que le meilleur modèle est celui avec le Gradient Boosting sans sélection de variables et sans oversampling sur un échantillon de 999 999 données (4.8% de F-mesure) . Suite à cela, nous avons voulu booster ce modèle et nous avons fait des recherches sur l'optimisation de celui-ci. Nous sommes tombés sur un article qui

parlait d'un algorithme Gradient Boosting mais propre à l'utilisation sur des données volumineuses. Sous python la librairie est "lightgbm". Juste en testant sans aucun paramètres, les résultats n'étaient pas satisfaisants. Nous avons alors eu l'idée d'utiliser la méthode Cost-Sensitive Learning permettant d'ajouter des poids aux classes de la variable cible afin que la classe minoritaire ait plus d'importance dans le modèle. Pour cela nous avons utilisé le module "class_weight" de sklearn permettant d'optimiser les poids des classes sur l'ensemble des données.

Le code est le suivant :

```
from sklearn.utils import class_weight

classes = np.unique(Ytrain)
cw = class_weight.compute_class_weight(class_weight = 'balanced',classes =  np.unique(Ytrain),y= Ytrain)
weights = dict(zip(classes,cw))
```

En passant les poids dans le modèle, nous avons alors obtenu 4.3% de F-mesure. Les résultats étaient satisfaisants car le code à mis très peu de temps à tourner avec toutes les données du fichier Train. Nous avons donc continué à thuner ce modèle en réalisant un GridSearchCV en variant les hyper-paramètres. Après avoir les meilleurs hyper-paramètres, nous avons relancé le modèle. Le code est le suivant :

```
from lightgbm import LGBMClassifier

lgbm = LGBMClassifier(class_weight = weights,max_depth=50, alpha=0.7, loss='deviance', learning_rate=0.09, n_estimators=1000, subsample=0.6,reg_alpha = 1.6, reg_lambda = 1.1,random_state=0).fit(Xtrain, Ytrain)

pred = lgbm.predict(Xtest)


ctest = confusion_matrix(Ytest, pred)

ftest = (2*ctest[1,1]/(2*ctest[1,1]+ctest[0,1]+ctest[1,0])) * 100

ftest
```

Nous avons réussi à obtenir un score de 5,8% après cela. Nous avons testé après ça une sélection de variables avec l'outil "SelectFromModel" ou encore de combiner les poids avec un borderLineSmote à 0.3 de sampling_strategy mais ça n'a pas été concluant.

Code ci-dessous :

```
from lightgbm import LGBMClassifier
from imblearn.over_sampling import BorderlineSMOTE


sm = BorderlineSMOTE(sampling_strategy = 0.3,random_state = 0)
Xtrain, Ytrain = sm.fit_resample(Xtrain, Ytrain)


lgbm = LGBMClassifier(class_weight = weights,max_depth=50, alpha=0.7, loss='deviance', learning_rate=0.09, n_estimators=1000, subsample=0.6,reg_alpha = 1.6, reg_lambda = 1.1,random_state=0).fit(Xtrain, Ytrain)
pred = lgbm.predict(Xtest)


ctest = confusion_matrix(Ytest, pred)
ftest = (2*ctest[1,1]/(2*ctest[1,1]+ctest[0,1]+ctest[1,0])) * 100
```

```
from sklearn.feature_selection import SelectFromModel

model = SelectFromModel(lgbm, prefit=True)
Xtrain_2 = Xtrain[Xtrain.columns[model.get_support(indices = True)]]
Xtest_2 = Xtest[Xtest.columns[model.get_support(indices = True)]]
```

Nous avons enfin testé avec la normalisation des données mais le score à diminuer de 1%.

Conclusion

En conclusion, les problèmes fortement déséquilibrés de Machine Learning sont très difficiles et nécessitent une exploration très approfondie pour commencer à avoir des performances. Le volume de données a ajouté un niveau de complexité supplémentaire. Plusieurs algorithmes et plusieurs méthodes ont été explorés dans le but de découvrir lequel d'entre eux semblait être le

plus performant, optimisable et viable par son temps d'exécution. Le classifieur qui s'est montré le plus prometteur selon notre méthodologie est le Gradient Boosting. Les librairies de parallélisation des calculs se sont révélées utiles puisque malgré leur utilisation l'exécution était parfois assez longue. Les sélections de variables et le Borderline SMOTE ne se sont pas montrés efficaces sur le Gradient Boosting, mais semblaient utiles sur d'autres algorithmes. Le côté exploratoire et itératif du Machine Learning a donc pris tout son sens dans ce projet puisque chaque algorithme réagit différemment en lien avec sa méthode. Les SVM se sont montrés très décevants dans ce projet car très long dans les temps de calculs malgré les librairies de parallélisation, nous avons donc échoué dans leur utilisation malgré le fait d'avoir réduit les échantillons pour l'apprentissage. Ils ne se sont pas montrés performants et c'est pour cette raison qu'ils n'ont pas été présentés dans les résultats.

La principale perspective pour cette étude est donc d'explorer plus en profondeur la méthode de Cost-Sensitive Learning qui semble particulièrement adaptée à ce type de problème. On peut donc supposer qu'avec cela est un Gradient Boosting optimisé dans ses hyper-paramètres, les performances de prédiction pourront devenir bien meilleures. En effet, nous aurions pu sélectionner plus de choix hyper-paramètres afin d'avoir éventuellement une performance accrue du modèle. A l'inverse, l'utilisation de la méthode d'échantillonnage n'est pas forcément la bonne solution pour cet algorithme en vue des résultats que nous avons eu dans les tableaux. L'algorithme LGBMClassifier issu du module "lightgbm" est , nous le pensons, une bonne découverte et pourrait s'avérer très efficace sur ce même type de données. En effet, il est rapide et puissant et apparemment utilisé de plus en plus souvent en vue de l'augmentation de la taille des fichiers de données. C'est une piste à explorer en profondeur, nous sommes peut-être partis sur des mauvaises pistes. Cela nous laisse d'immenses perspectives en vue d'améliorer le modèle en recherchant d'autres méthodes. Cependant, les recherches et production de ce projet nous ont appris beaucoup de choses au sujet des différentes techniques, méthodes pour par exemple traiter des fichiers volumineux ou encore traiter des données déséquilibrées.