Master SISE

Classification dans un contexte déséquilibré : la fraude bancaire

Fouille de données massives - M.Metzler

Hamza FASSI, Pierre LE GALÈZE,

Promo 2021/2022

16 Janvier 2022

Table des matières

Chapitre 1

Introduction

Dans le cadre de notre projet de Fouille de donnée de deuxième année de Master SISE (Statistique et Informatique pour la Science des Données), nous avons conduit une étude de cas visant à classifier de manière la plus fiable possible les fraudes bancaires

Afin d'appliquer ces différentes méthodes, nous avons utilisé des données réelles, issues d'une enseigne de la grande distribution ainsi que de certains organismes bancaires (FNCI et Banque de France). Chaque ligne représente une transaction effectuée par chèque dans un magasin de l'enseigne quelque part en France, elles ne sont pas brutes et plusieurs variables sont déjà des variables créées (feature engineering). Le jeu de donnée comprend 10 mois de transactions du "2017-02-01" au "2017-11-30" avec un total de 4646773 lignes pour 23 variables. On peut retrouver ce jeu de donnée à l'adresse suivante : https://filez.univ-lyon2.fr/9dzynu

Chapitre 2

Préparation des données et Statistiques descriptives

La partie statistique descriptive est importante, car elle peut révéler certains phénomènes et même écarter certaines directions de recherche. Nous utiliserons principalement la librairie "seaborn", qui agrège les données et affiche des intervalles de confiance à 95%. On commence par regarder notre variable cible : " FlagImapaye". Il s'agit d'une variable qui ne peut prendre que deux valeurs possibles :

- 0 : la transaction est acceptée et considérée comme "normale".
- 1 : la transaction est refusée, car considérée comme "frauduleuse".

Mais avant de commencer cette partie, on va jetter un premier coup d'œil sur nos données après les avoir manipulé grâce à la librairie "pandas".

On commence par regarder les types de nos différentes variables et on remarque que nos données sont principalement numériques :

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2231369 entries, 0 to 2231368
Data columns (total 23 columns):
 #   Column                    Dtype
---  ------                    -----
 0   ZIBZIN                    object
 1   IDAvisAutorisAtionCheque  int64
 2   MontAnt                   float64
 3   DAteTrAnsAction           object
 4   CodeDecision              int64
 5   VerifiAnceCPT1            int64
 6   VerifiAnceCPT2            int64
 7   VerifiAnceCPT3            int64
 8   D2CB                      int64
 9   ScoringFP1                float64
 10  ScoringFP2                float64
 11  ScoringFP3                float64
 12  TAuxImpNb_RB              float64
 13  TAuxImpNB_CPM             float64
 14  EcArtNumCheq              int64
 15  NbrMAgAsin3J              int64
 16  DiffDAteTr1               float64
 17  DiffDAteTr2               float64
 18  DiffDAteTr3               float64
 19  CA3TRetMtt                float64
 20  CA3TR                     float64
 21  Heure                     int64
 22  FlAgImpAye                int64
dtypes: float64(11), int64(10), object(2)
memory usage: 391.6+ MB
```

Ensuite on va voir la signification de ces variables :

- **ZIBZIN** : identifiant relatif à la personne, i.e. il s'agit de son identifiant bancaire (relatif au chéquier en cours d'utilisation)

- **IDAvisAutorisAtionCheque** : identifiant de la transaction en cours

- **Montant** : montant de la transaction

- **DateTransaction** : date de la transaction

- **CodeDecision** : il s'agit d'une variable qui peut prendre ici 4 valeurs
0 : la transaction a été acceptée par le magasin
1 : la transaction et donc le client fait partie d'une liste blanche (bons payeurs).
2 : le client fait d'une partie d'une liste noire, son historique indique cet un mauvais payer (des impayés en cours ou des incidents bancaires en cours), sa transaction est alors automatiquement refusée
3 : client ayant était arrêté par le système par le passé pour une raison plus ou moins fondée

- **VérifianceCPT1** : nombre de transactions effectuées par le même identifiant bancaire au cours du même jour.

- **VérifianceCPT2** : nombre de transactions effectuées par le même identifiant bancaire au cours des trois derniers jours.

- **VérifianceCPT3** : nombre de transactions effectuées par le même identifiant bancaire au cours des sept derniers jours.

- **D2CB** : durée de connaissance du client (par son identifiant bancaire), en jours. Pour des contraintes légales, cette durée de connaissance ne peut excéder deux ans.

- **ScoringFP1** : score d'anormalité du panier relatif à une première famille de produits (ex : denrées alimentaires).

- **ScoringFP2** : score d'anormalité du panier relatif à une deuxième famille de produits (ex : électroniques).

- **ScoringFP3** : score d'anormalité du panier relatif à une troisième famille de produits (ex : autres).

- **TauxImpNb_RB** : taux impayés enregistrés selon la région où a lieu la transaction.

- **TauxImpNB_CPM** : taux d'impayés relatif au magasin où a lieu la transaction.

- **EcartNumCheq** : différence entre les numéros de chèques.

- **NbrMagasin3J** : nombre de magasins différents fréquentés les 3 derniers jours.

- **DiffDateTr1** : écart (en jours) à la précédente transaction.

- **DiffDateTr2** : écart (en jours) à l'avant-dernière transaction.

- **DiffDateTr3** : écart (en jours) à l'antépénultième transaction.

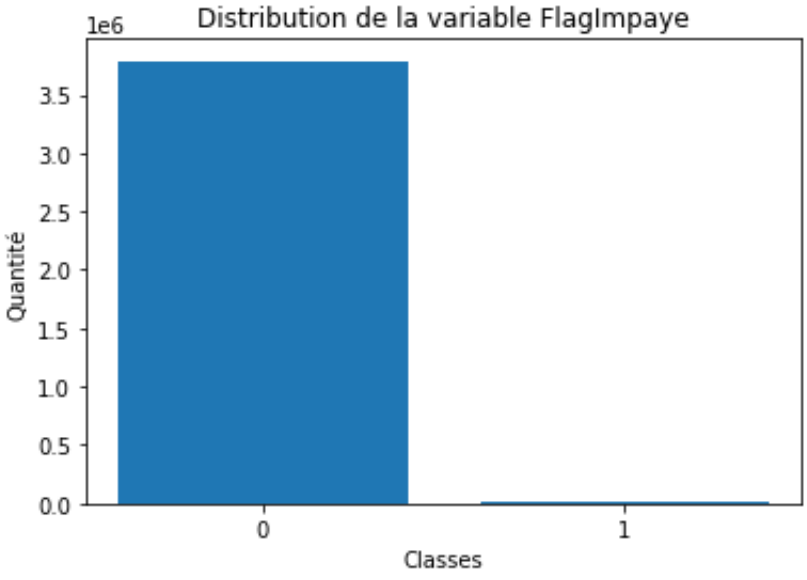
- **CA3TRetMtt** : montant des dernières transactions + montant de la transaction en cours.

- **CA3TR** : montant des trois dernières transactions.

- **Heure** : heure de la transaction.

- **FlagImpaye** : acception (0) ou refus de la transaction (1).

Comme on l'a déjà mentionné, la variable cible est donc **FlagImpaye** et est noté 0, lorsque la transaction est considérée comme normal et 1 si elle est considéré comme frauduleuse. Pour plus de précisions, observons précisément combien d'observation il y a dans chaque classe :



Comme on peut le voir, le jeu de donnée est très peu équilibré avec un ratio de 183.75 : 1.

2.1 Préparation de la base de donnée

À noter qu'aucune donnée manquante n'a été trouvée dans le jeu de donnée, ainsi que les doublons et les deux colonnes identifiants (ZIBZIN et IDAvisAutorisationCheque) ont été supprimées. Il en va de même pour une ligne contenant les mêmes valeurs que les noms de colonnes.
On a pu retrouver trois types de variables dans le jeu de donnée, des catégorielles, des numériques et des dates.
Pour les données de types catégorielles et dates, il suffisait de les convertir tandis que pour les numériques, il a fallu d'abord remplacé toutes les "," par des ".".

Nous avons aussi profité de la conversion en date pour récupérer à partir de cette dernière, le mois, le jour et l'heure pour pouvoir les ajouter en tant que colonnes.

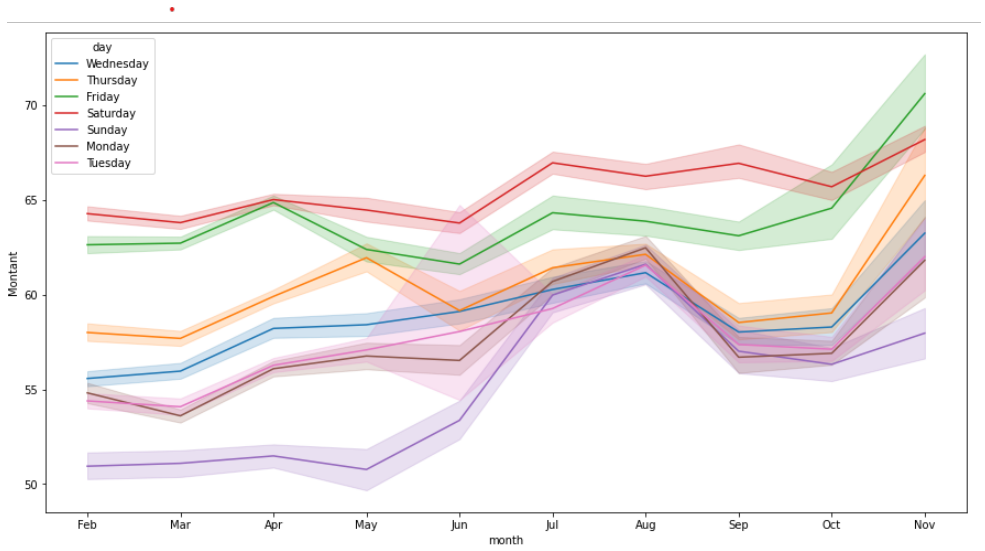
Sur les 23 variables de départs, nous avons supprimé la variable ZIBZIN, IDAvisAutorisAtionCheque et DateTransaction mais aussi rajouté deux variables qui sont le mois et le jour de la transaction (Month / Weekday).

Continuons alors nos analyses par l'observation des différents graphes ci-dessous.

2.2 Statistiques descriptives

2.2.1 Montant dépensé par mois selon les jours de la semaine

Le premier graphe qu'on va voir nous présente le montant dépensé par mois selon les jours de la semaine :

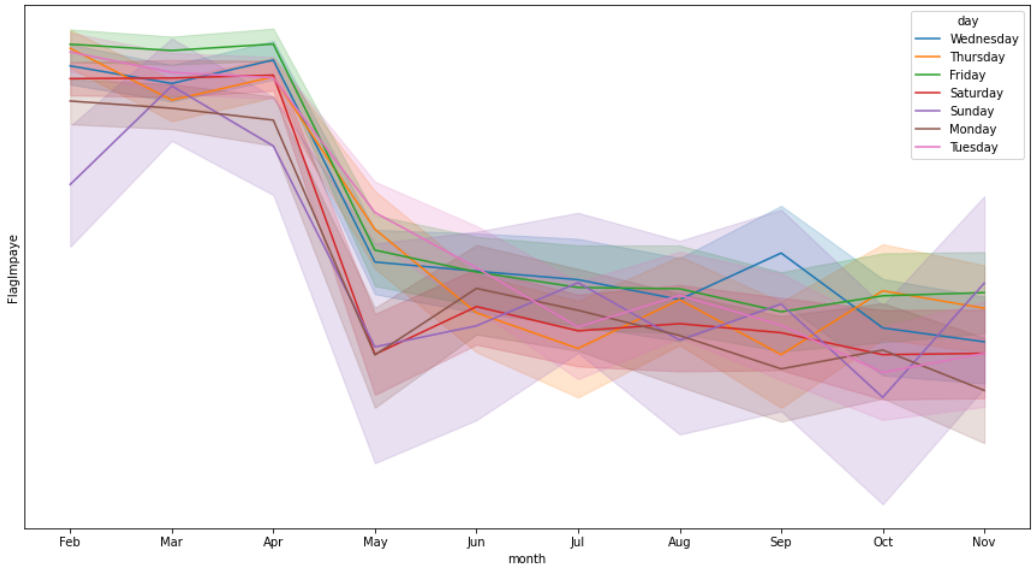


Si on regarde de plus prêt le graphe ci-dessus, on remarque que le jour où on dépense le plus est le **samedi** et celui où l'on dépense le moins est le **dimanche**.
Au niveau des mois, c'est le mois de juin qui apparaît comme le moins productif niveau vente

tandis que le mois de décembre est le plus productif.

2.2.2 Nombre de transactions refusées par mois selon les jours de la semaine

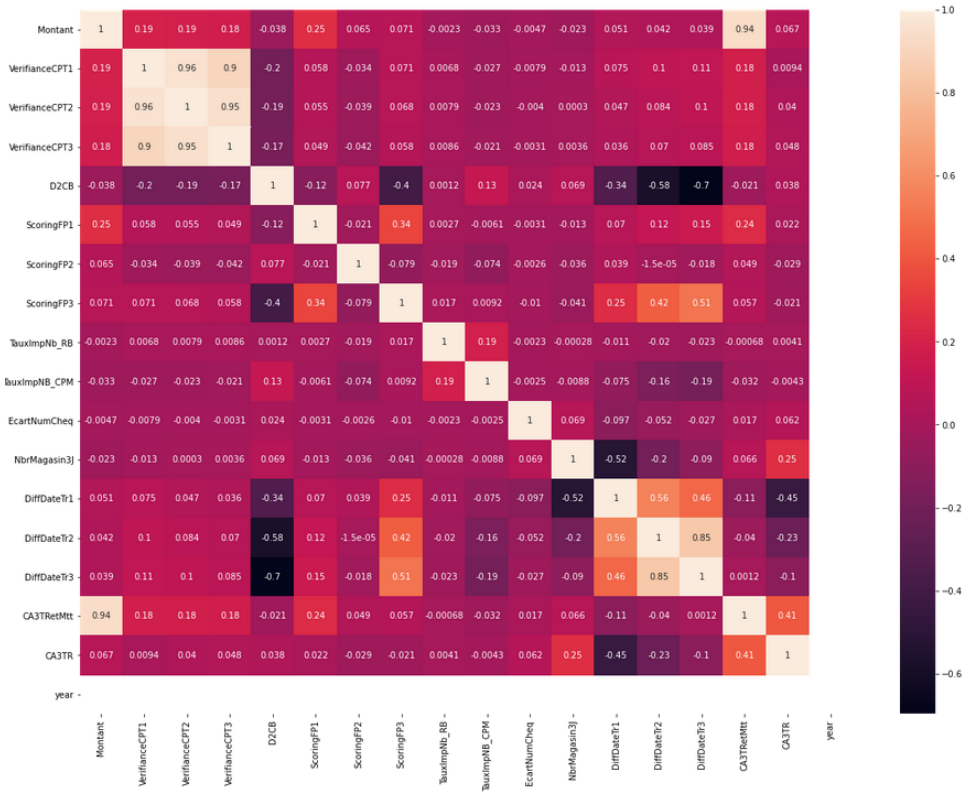
Le graphe ci-dessous représente le nombre de transactions refusées par mois selon les jours de la semaine.



On arrive à voir que le jour qui présente le plus de transactions frauduleuses en moyennes est le vendredi et que le mois où on a le plus grand nombre de ces dernières est le mois d'avril.

2.2.3 Matrice de corrélation

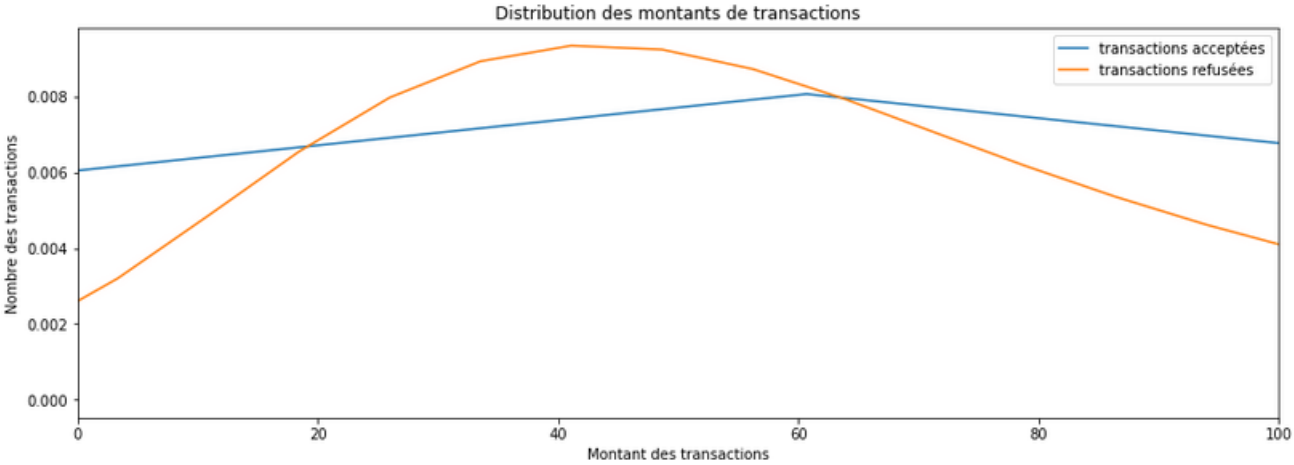
Dans cette partie, on va s'intéresser à la matrice de corrélation de toutes nos variables.



La matrice de corrélation est utilisée pour évaluer la dépendence entre plusieurs variables en même temps.
En observant bien la matrice de corrélation obtenue pour notre jeu de données, on remarque par exemple que :
- Les variables **diffdatetr** sont corrélées entre elles même.
- Le nombre de magasins fréquentés les 3 derniers jours et l'écart (en jours) à la précédente transaction : -0.7.
- Le montant et le score d'anormalité du panier relatif à une première famille de produits : 0.25.
- Les variables **verifiancecptt** sont aussi corrélées entre elles même.

2.2.4 Distribution des montants de transaction

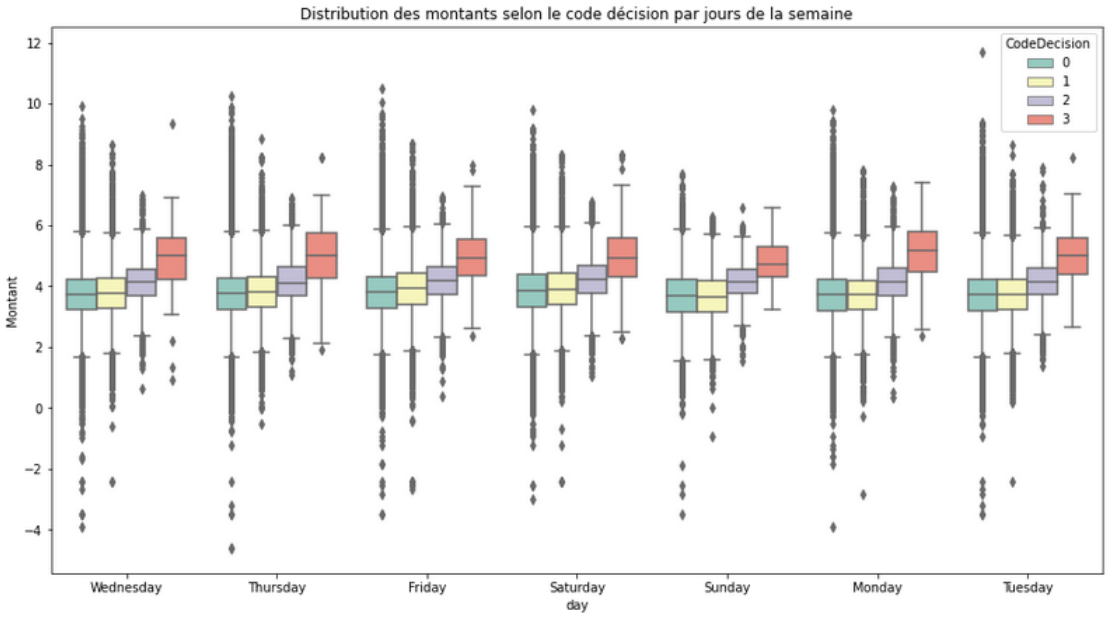
On a commencé par voir le pourcentage des transactions qui sont considérées comme frauduleuses et qui est égale à 0.6%.
Ensuite dans le graphe ci-dessous, on remarque que peu importe le type de transaction, les distributions ont relativement la même forme. On remarque également que passé 60 euros de dépenses, il y a moins de chance à ce que la transaction soit frauduleuse.

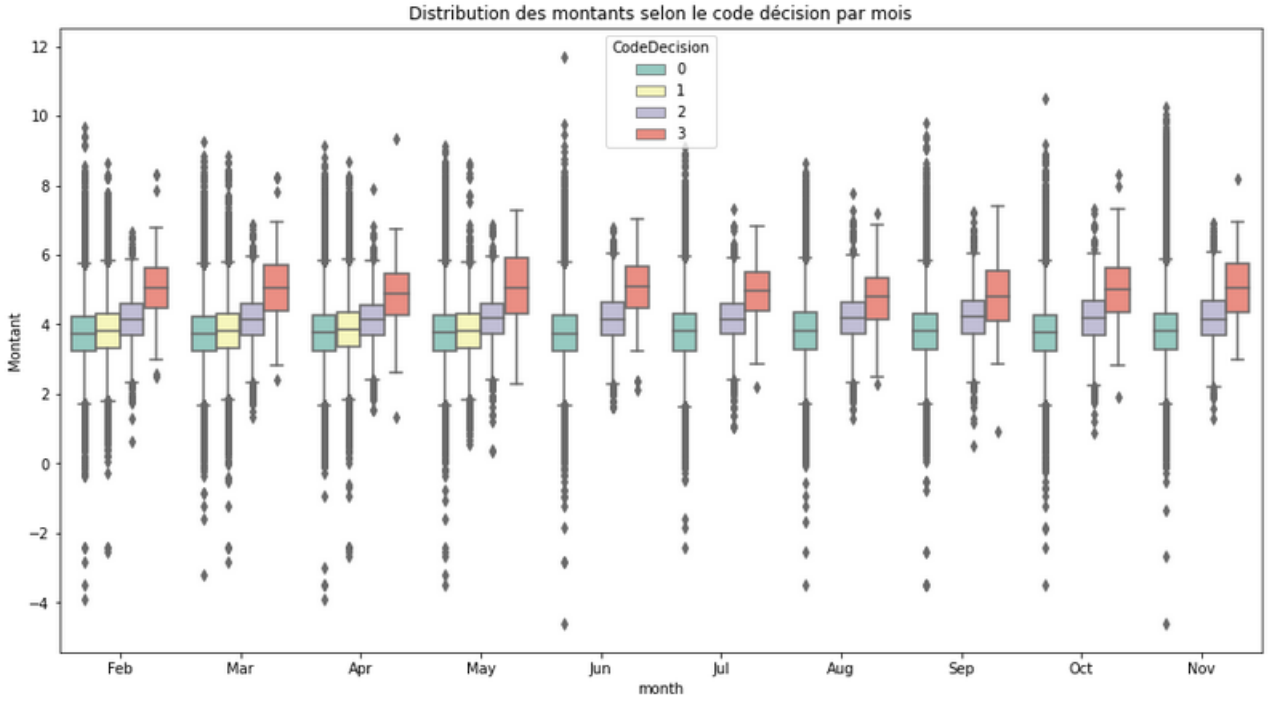


Si on observe encore plus ce graphe, on peut dire que les transactions ont plus tendances à être refusées lorsque les dépenses sont entre 20 euros et 60 euros.

2.2.5 Etude du code de décision

Le code de décision peut savoir si la transaction est acceptée, si le client appartient à la liste blanche ou à la liste noire, ou encore si le client a été arrêté par le système pour une raison valable. On peut donc penser que cette variable va probalement influer sur le fait qu'une décision soit frauduleuse ou non.
Nous examinons d'abord la répartition des montants quotidiens/mensuels en fonction du code de décision. Cela peut nous permettre de distinguer des phénomènes spécifiques.





Pour rappel, les graphes ci-dessus s'intéresse à notre variable **CodeDecision** qui s'agit d'une variable qui peut prendre ici 4 valeurs :
0 : la transaction a été acceptée par le magasin
1 : la transaction et donc le client fait partie d'une liste blanche (bons payeurs).
2 : le client fait d'une partie d'une liste noire, son historique indique qu'il existe des impayés en cours ou des incidents bancaires en cours, sa transaction est alors automatiquement refusée.
3 : client ayant était arrêté par le système par le passé pour une raison plus ou moins fondée.

CodeDecision	0	1	2	3	All
FlagImpaye					
0	0.699266	0.294158	0.000143	0.000384	0.993952
1	0.001986	0.000074	0.003797	0.000190	0.006048
All	0.701252	0.294233	0.003941	0.000574	1.000000

À l'aide de ce tableau aussi, on voit bien ici que les codes de décision 2 et 3 n'ont aucun effet sur la validité de la transaction. Même les transactions considérées comme frauduleuses ont souvent un code de décision de 0. Nous pouvons alors garder ou exclure ces variables de notre étude.

Chapitre 3

Algorithmes et recherches

3.1 Nos recherches

En ce qui concerne les algorithmes utilisés, 5 ont été sélectionnés au vu de nos recherches sur internet, de nos tests et ainsi que d'après l'étude de cas du cours de Fouille de donnée :

— **Linear Discriminant Analysis** : L'analyse discriminante linéaire est un algorithme d'apprentissage automatique de classification linéaire. L'algorithme consiste à développer un modèle probabiliste par classe basé sur la distribution spécifique des observations pour chaque variable d'entrée. Un nouvel exemple est ensuite classé en calculant la probabilité conditionnelle de son appartenance à chaque classe et en sélectionnant la classe dont la probabilité est la plus élevée.

— **XGBoost** :XGBoost est une implémentation des arbres de décision à gradient boosté conçue pour la vitesse et la performance. Il s'agit d'une mise en œuvre de machines d'amplification du gradient créée par Tianqi Chen, à laquelle contribuent désormais de nombreux développeurs.

— **AdaBoost** : AdaBoost, également appelé Adaptive Boosting, est une technique d'apprentissage automatique utilisée comme méthode d'ensemble. L'algorithme le plus communément utilisé avec AdaBoost est celui des arbres de décision à un niveau, c'est-à-dire avec des arbres de décision à une seule division. Ces arbres sont également appelés "Decision Stumps".

— **Logistic Regression** : La régression logistique est un algorithme de classification utilisé pour affecter des observations à un ensemble discret de classes. Voici quelques exemples de problèmes de classification : spam ou non, transactions en ligne frauduleuses ou non, tumeur maligne ou bénigne. La régression logistique transforme son résultat à l'aide de la fonction sigmoïde logistique pour renvoyer une valeur de probabilité.

— **Random Forest Classifier** : Random Forest Classifier est un algorithme d'apprentissage supervisé. La "forêt" qu'il construit est un ensemble d'arbres de décision, généralement formés avec la méthode "bagging". L'idée générale de cette méthode est qu'une

combinaison de modèles d'apprentissage augmente le résultat global.

Ces différents algorithmes ont ensuite été appliqués sur le jeu de donnée afin de regarder principalement le F1-Score, mais aussi l'AUC de la courbe ROC du modèle. D'après nos recherches, nous avons le choix sur plusieurs méthodes pour corrigé les déséquilibres d'un jeu de donnée. Ces méthodes peuvent agir à 3 niveaux : au niveau des données (data-level solutions), au niveau des algorithmes (algorithm-level solutions), ou au niveau de l'erreur de classification.

Data-level solutions

Plusieurs techniques existent pour ajuster la distribution des classes d'un ensemble de données :

- **SMOTE** (Synthetic Minority Oversampling Technique) : méthode qui génère artificiellement de nouveaux exemples de la classe minoritaire en utilisant les plus proches voisins de ces cas.
- **Random Oversampling** : méthode qui tire au hasard des individus minoritaires pour les rajouter aux données. Les individus minoritaires se voient ainsi « clonés » de multiples fois, raison pour laquelle cette méthode est parfois appelée « Replicative oversampling ». Elle est peu efficace pour les arbres, car elle ne leur permet pas de généraliser, cela est à prendre en compte lorsque nous l'utiliserons.
- **Random Undersampling** : méthode qui retire aléatoirement x% des observations majoritaires. Il peut aussi être spécifique, avec des méthodes de clustering-based undersampling ou d'undersampling des observations frontalières.

Algorithm-level solutions

On peut aussi s'intéresser à certains paramètres des modèles qui permette d'augmenter la pondération pour les individus positifs (et minoritaire).

Par exemple :

- **Via le gradient** (boosting) : c'est le rôle du paramètre scale_pos_weight dans XGBoost qui permet de donner davantage de poids aux observations minoritaires.
- **Via le Gini** (bagging) : c'est le rôle du paramètre class_weight de RandomForestClassifier qui accroît le poids des classes minoritaires. Le paramètre class_weight existe aussi pour le modèle de Logistic Regression.

3.1.1 Jeu de donnée initiale sans aucun changement

Tout d'abord, nous avons donc exécuté les 5 algorithmes sur le jeu de donnée initial, afin d'avoir un point de comparaison avec le reste. Au départ, cela n'était pas censé marché, car nous pensions que seul une des techniques de ré-échantillonnage vu plus haut permettait à un algorithme de bien fonctionné sur un jeu de donnée déséquilibré. Malgré tout, nous avons été surpris par les performances des modèles que nous avions choisis. Pour d'autre modèle, par exemple, les Arbres de décision ou encore les SVM, les résultats étaient bien inférieurs (un F1-Score de 10% voir moins).

On a donc finalement :

— Pour la Logistic Regression, un F1-Score de 0.778.

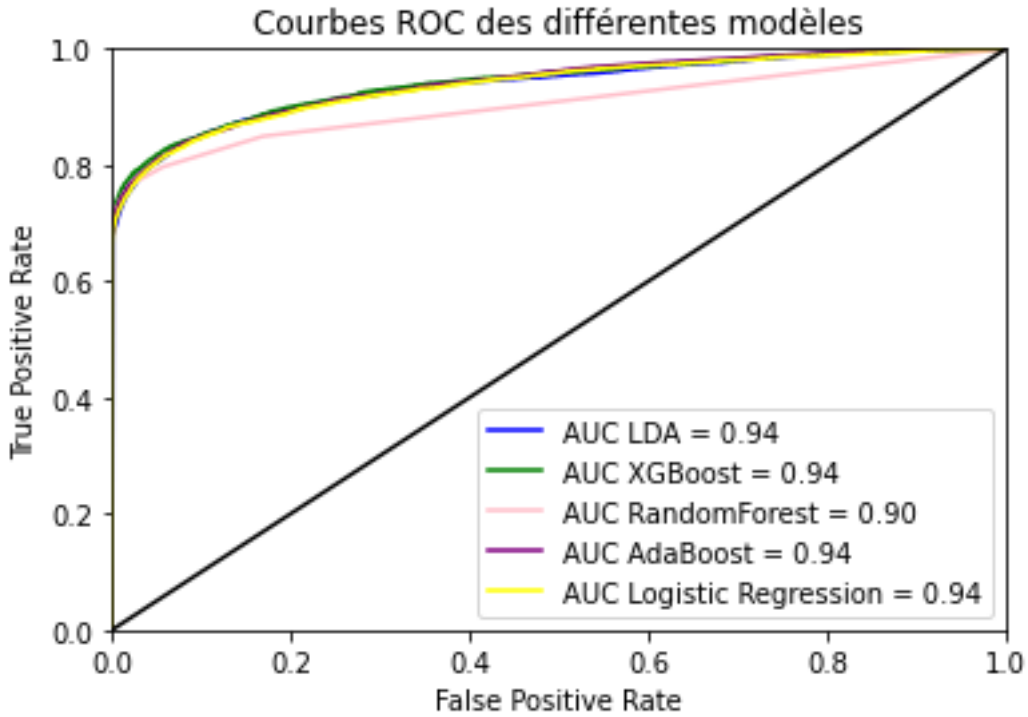
— Pour Random Forest Classifier, un F1-Score de 0.794.

— Pour AdaBoost, un F1-Score de 0.792.

— Pour l'algorithme Linear Discriminant Analysis, un F1-Score de 0.779.

— Pour XGBoost, un F1-Score de 0.796.

On se retrouve aussi avec :



Soit pour résumé :

	F1-Score	AUC
Logistic Regression	0.778	0.94
Random Forest Classifier	0.794	0.90
AdaBoost	0.792	0.94
LDA	0.779	0.94
XGBoost	0.796	0.94

On remarque que, malgré que le Random Forest fasse partie des meilleurs algorithmes en termes de F1-Score, son AUC est la seule inférieure à 0.94. Le meilleur modèle dans ce cas-ci est donc XGBoost avec 0.796 de F1-Score et 0.94 de AUC (même si la différence avec les autres n'est pas si grande). On pourrait sans aucun doute améliorer ces modèles via un Grid-Search mais le processus étant trop long, on a préféré partir sur autre chose.

3.1.2 Jeu de donnée rééquilibré

Maintenant que nous avions les résultats sur le jeu de donnée initial, il nous fallait tester les modèles avec 3 différentes techniques de rééquilibrage, les comparer ensembles, mais aussi avec les résultats du jeu de donnée de base.

Les 3 techniques que nous avons choisies d'utilisées sont donc un SMOTE, un Random Oversampling et un Random Undersampling.

SMOTE

Pour cette partie, nous allons simplement effectué un SMOTE sur les données et testé directement chaque modèle. Le paramètre sampling_strategy de SMOTE sera configurer sur 0.1 de sorte à ne pas trop rajouter de donnée de la classe minoritaire (on se retrouve donc avec un ratio 1/10 entre la classe minoritaire et la majoritaire). On initialisera aussi le paramètre k_neigbors à 1700 (tester via GridSearch, on a remarqué que plus k_neigbors était grand, meilleurs était le F1-Score, mais nous n'avons pas pu aller au-delà de 1800 à cause d'une erreur qui aurait été trop compliqué à résoudre).

On se retrouve avec :

— Pour la Logistic Regression, un F1-Score de 0.765.

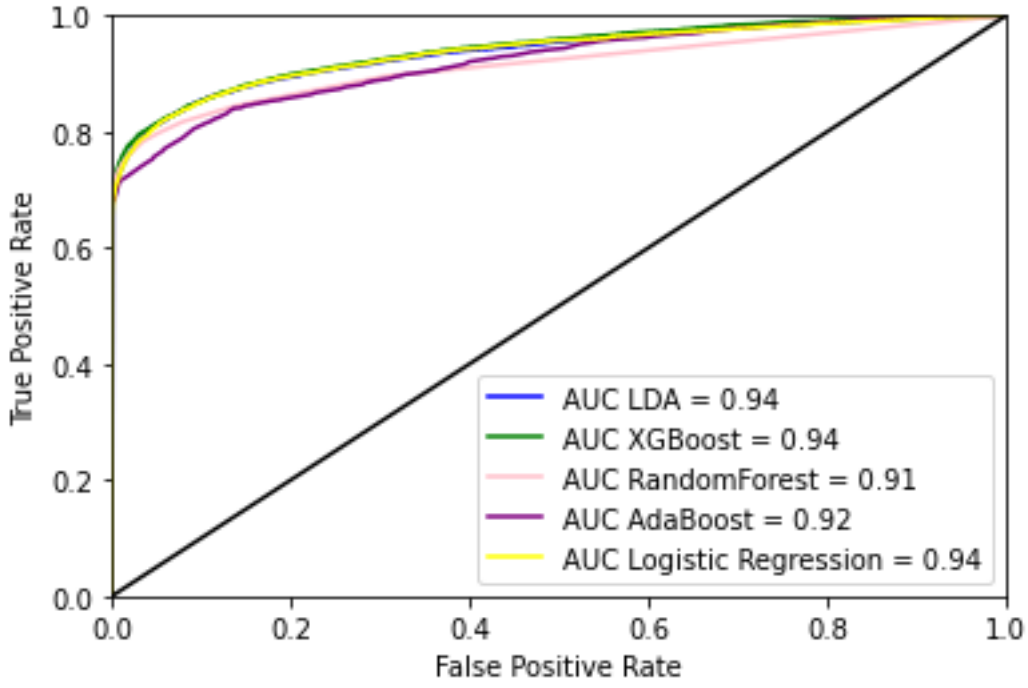
— Pour Random Forest Classifier, un F1-Score de 0.796.

— Pour AdaBoost, un F1-Score de 0.794.

— Pour l'algorithme Linear Discriminant Analysis, un F1-Score de 0.776.

— Pour XGBoost, un F1-Score de 0.796.

Si on s'intéresse à la courbe ROC maintenant, on a :



SMOTE et Undersampling

Ici, nous allons combiner l'oversampling de SMOTE et un Random Undersampling. De même qu'auparavant, le paramètre sampling_strategy de SMOTE sera configurer sur 0.1 et celui du Random Oversampling à 0.5. On a donc d'abord appliqué le suréchantillonnage pour augmenter le rapport à 1 :10 en dupliquant les exemples de la classe minoritaire, puis appliquer le sous-échantillonnage pour améliorer le rapport à 1 :2 en supprimant les exemples de la classe majoritaire.

On a :

— Pour la Logistic Regression, un F1-Score de 0.514.

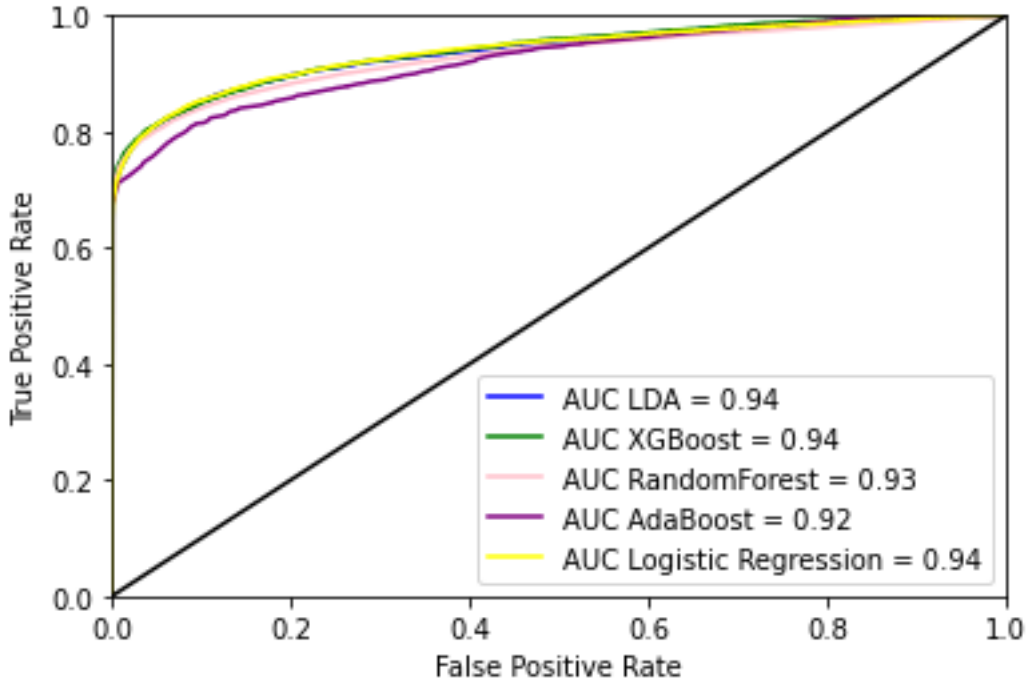
— Pour Random Forest Classifier, un F1-Score de 0.767.

— Pour AdaBoost, un F1-Score de 0.759.

— Pour l'algorithme Linear Discriminant Analysis, un F1-Score de 0.760.

— Pour XGBoost, un F1-Score de 0.786.

La courbe ROC nous montre alors :



Undersampling et Oversampling

Même chose qu'auparavant pour ce cas-ci, à la différence que l'on remplace SMOTE par un Random Undersampling

On se retrouve avec :

— Pour la Logistic Regression, un F1-Score de 0.318.

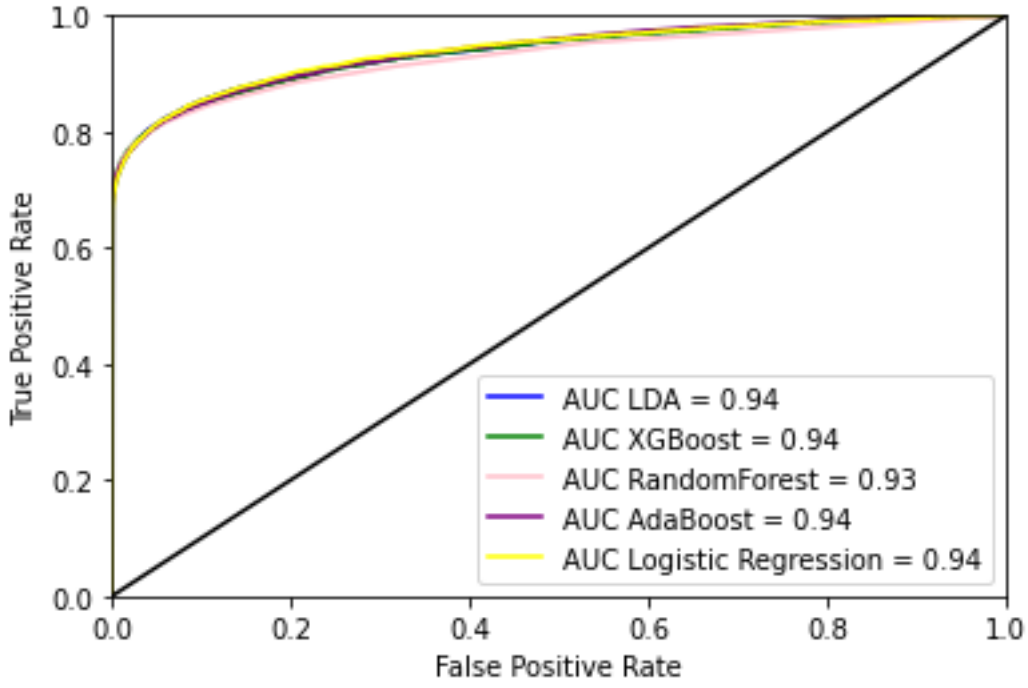
— Pour Random Forest Classifier, un F1-Score de 0.793.

— Pour AdaBoost, un F1-Score de 0.248.

— Pour l'algorithme Linear Discriminant Analysis, un F1-Score de 0.676.

— Pour XGBoost, un F1-Score de 0.305.

Ainsi que :



Comparaison

On compare ensuite les résultats entre chaque méthode, mais aussi avec les résultats obtenus sur le jeu de donnée initial. Pour qu'il n'y ait pas trop de score dans le tableau, nous indiquerons si le modèle se voit améliorer avec les méthodes d'échantillonnage via + (a augmenté de quelque pourcent) et + + (a beaucoup augmenté) et s'il est moins performant avec - (a baissé de quelque pourcent) et - - (a beaucoup baissé). L'égal indique si les résultats sont les mêmes ou simplement peu significatif.

	F1-Score	AUC
Logistic Regression SMOTE	0.514 (- -)	0.94 (=)
Logistic Regression SMOTE - Under	0.764 (-)	0.94 (=)
Logistic Regression Over - Under	0.318 (- -)	0.94 (=)
Random Forest Classifier SMOTE	0.796 (=)	0.91 (+)
Random Forest Classifier SMOTE - Under	0.767 (-)	0.93 (+)
Random Forest Classifier Over - Under	0.793 (=)	0.93 (+)
AdaBoost SMOTE	0.794 (=)	0.92 (-)
AdaBoost SMOTE - Under	0.759 (-)	0.92 (-)
AdaBoost Over - Under	0.248 (- -)	0.94 (=)
LDA SMOTE	0.776 (=)	0.94 (=)
LDA SMOTE - Under	0.760 (-)	0.94 (=)
LDA Over - Under	0.676 (- -)	0.94 (=)
XGBoost SMOTE	0.796 (=)	0.94 (=)
XGBoost SMOTE - Under	0.786 (-)	0.94 (=)
XGBoost Over - Under	0.305 (- -)	0.94 (=)

On remarque qu'aucunes méthodes n'améliorent un des modèles de manières significatives, que ce soit avec le F1-Score ou l'AUC. Afin d'améliorer les résultats, on pourrait tester toutes les combinaisons possibles pour chaque méthode, notamment avec le paramètre sampling_strategy. On voit aussi que les pires scores arrivent lorsque l'on utilise le Random Oversampling, notamment pour tous algorithmes utilisant les arbres de décisions.

3.1.3 Jeu de donnée initiale avec ajustement des poids de la classe minoritaire

Étant donné que l'ont réussi à peine à améliorer nos résultats avec le rééquilibrage, on reprend notre jeu de donnée initial, mais, cette fois-ci, on s'intéresse aux hyper-paramètres pouvant directement influer sur le poids de certaines observations comme indiqué dans la partie "Nos recherches".

— Le modèle XGBoost met à disposition l'hyper-paramètre scale_pos_weight qui permet de donner davantage de poids aux observations minoritaires. Pour ce dernier, il est recommandé de l'utiliser avec la formule sqrt(somme(négatif) / somme(positif)) si on se trouve dans un contexte déséquilibré (et sans le sqrt si le contexte est normal), ce qui donne scale_pos_weight = 14. On va donc tester avec différentes valeurs situer autour de ce résultat :

scale_pos_weight	F1-Score
1	0.774
5	0.754
10	0.715
14	0.675
20	0.632

— On a aussi à disposition avec le modèle RandomForestClassifier un hyper-paramètre du même style permettant d'accroître le poids des classes minoritaires : class_weight. En testant plusieurs ratios pour class_weight, on se retrouve avec :

class_weight	F1-Score
('0' : 1, '1' : 1)	0.770
('0' : 1, '1' : 2)	0.769
('0' : 1, '1' : 3)	0.769
('0' : 1, '1' : 4)	0.752
('0' : 1, '1' : 5)	0.748

— Enfin, on retrouve le même hyper-paramètre class_weight avec le modèle LogisticRegression :

class_weight	F1-Score
('0' : 1, '1' : 1)	0.754
('0' : 1, '1' : 2)	0.754
('0' : 1, '1' : 3)	0.752
('0' : 1, '1' : 4)	0.748
('0' : 1, '1' : 5)	0.744

Cet hyper-paramètre peut aussi prendre la valeur 'balanced' (ajuste automatiquement le poids en fonction des effectifs de classe) mais cela empire les choses avec un F1-Score de 0.13.

À noter que pour le Grid-Search, on a cross valider avec TimeSeriesSplit() de sklearn, et ce, sur les données train afin de comparer ensuite les résultats du modèle avec son meilleur paramètre et le modèle initial. Malheureusement, étant donné que les meilleurs paramètres sont tous ceux du départ, cela ne sert pas à grand-chose.

3.1.4 Test en combinant les scores

Pour cette dernière partie, étant donné qu'aucune des solutions précédentes n'avaient marché, nous avons essayé de combiner les résultats obtenus. On a donc récupéré les probabilités de prédictions de chaque modèle pour la classe 1 (fraude) pour les rajouter aux données, puis on les a réentrainé. À noter que la prédiction du modèle entrainé n'apparait pas dans le dataset.

On se retrouve donc avec :

— Pour la Logistic Regression, un F1-Score de 0.797 (contre 0.778).

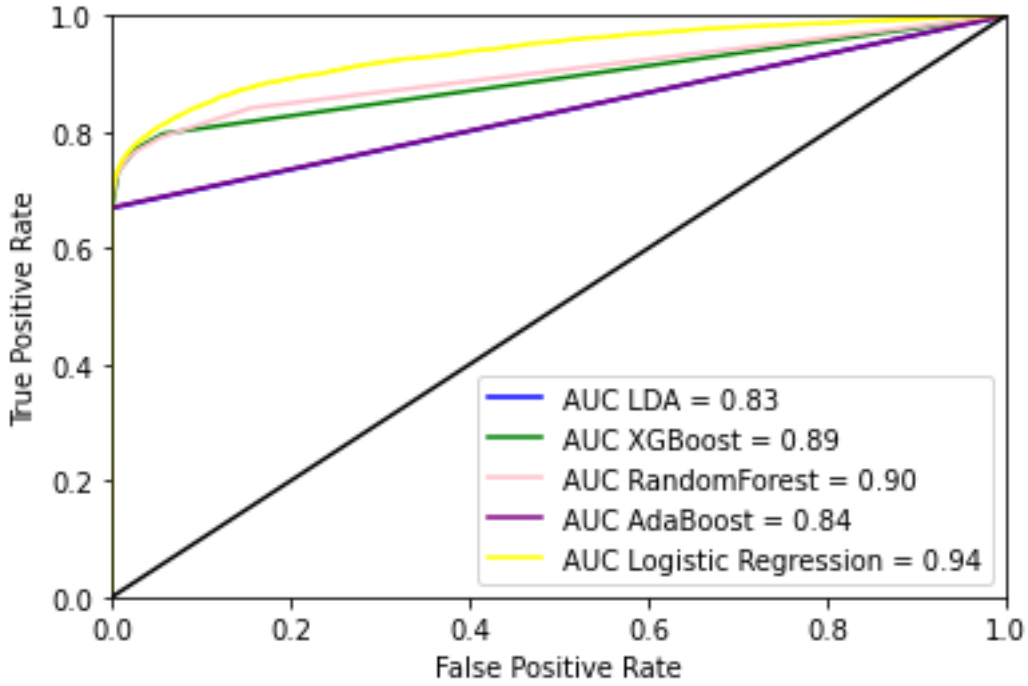
— Pour Random Forest Classifier, un F1-Score de 0.795 (contre 0.794).

— Pour AdaBoost, un F1-Score de 0.796 (contre 0.792).

— Pour l'algorithme Linear Discriminant Analysis, un F1-Score de 0.794 (contre 0.779).

— Pour XGBoost, un F1-Score de 0.796 (contre 0.796).

Avec l'ensemble des courbes ROC :

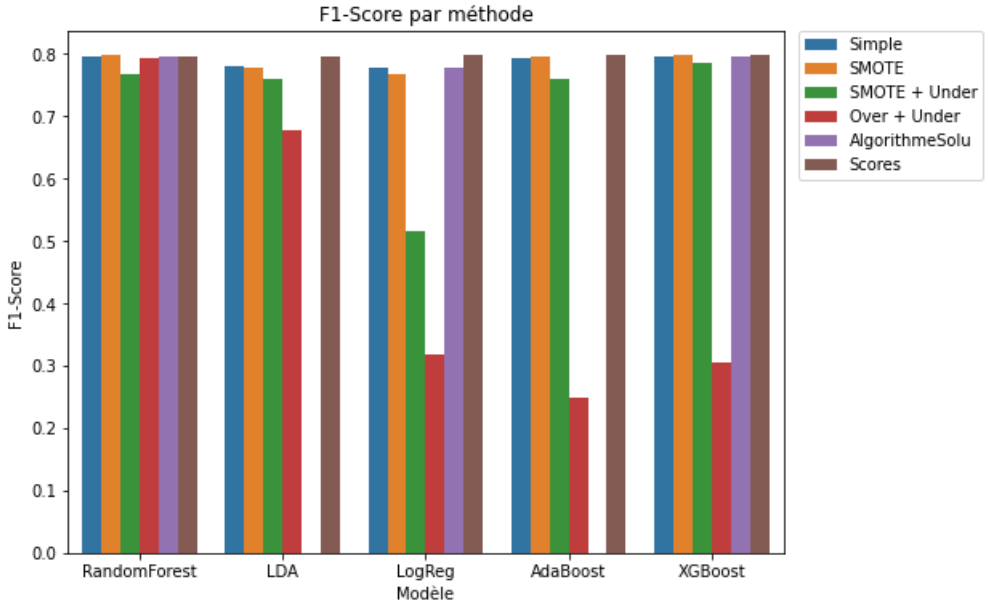


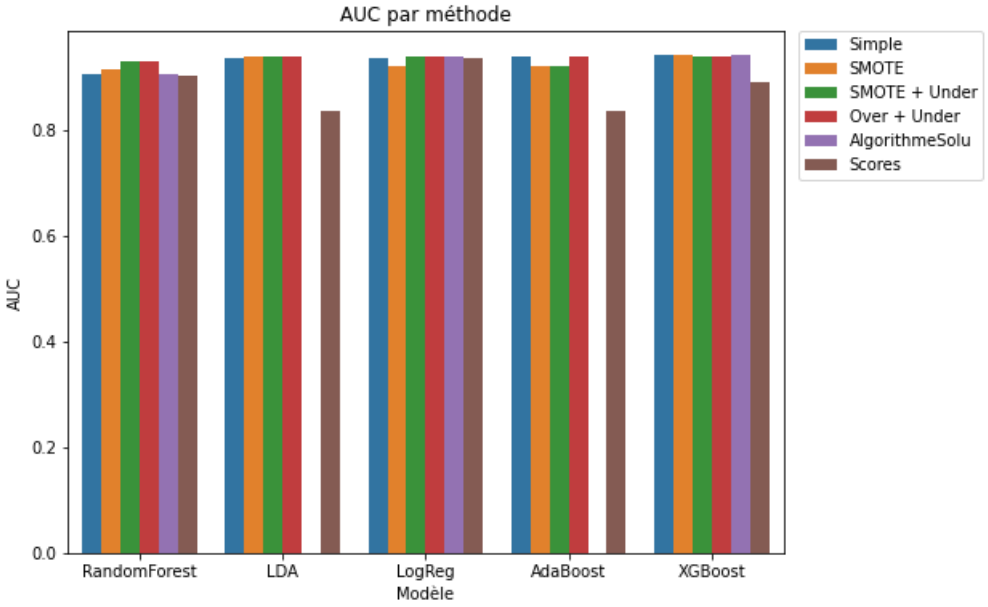
Si on se concentre sur les résultats du F1-Score, grâce à l'ajout des scores des différentes modèles, celui-ci se voit toujours augmenter. Malgré tout, si on se penche maintenant sur l'AUC, seule celle de la Logistic Regression est resté la même, toute les autres ont baissé.

3.2 Conclusion

3.2.1 Résultats

Comparons désormais tous les résultats de chacune des méthodes entre elles en les regroupant dans un bar plot :





Le modèle qui nous parait alors le meilleur est la Logistic Regression avec la méthode des scores. On se retrouve alors avec un F1-Score de 0.797 et une AUC de 0.94.

3.2.2 Pour aller plus loin

Pour résumé, on a donc testé plusieurs méthodes permettant de gérer un jeu de donnée déséquilibré : des méthodes apportant des solutions à partir des données, mais aussi directement via l'algorithme. Une autre méthode existe aussi et gère le cost-sensitive learning, c'est-à-dire permet d'accorder un poids plus grand dans l'apprentissage aux faux négatifs (les positifs qui sont prédits négatifs) qu'aux faux positifs (les négatifs qui sont prédits positifs). On pourrait notamment utiliser le paquetage Python "costcla" qui permet d'ajuster une régression logistique sensible aux coûts ou un arbre de décision (ainsi que plusieurs autres modèles).

Il faudrait aussi tuner tous les modèles que l'on utilise dans chacune des méthodes (que ce soit pour le sampling ou les modèles tel que la Logistic Regression) afin d'avoir une généralisation des résultats. Étant donné que cela prend énormément de temps si l'on veut tester toutes les combinaisons d'hyper-paramètres pour chaque modèle, on pourrait regarder du côté de la parallélisation ou bien de package tel que h2o / Dask.

Une autre idée est que l'on pourrait aussi mettre en place un K-Means (avec PCA au préalable) afin d'obtenir n clusters (définit via la méthode du coude avec le score de la silhouette), prendre les barycentres de ces n clusters, les regrouper sous un seul dataframe et reproduire ensuite un K-Means ou bien un dendrogramme sur ce dernier (si la taille est suffisamment réduite). On regrouperait finalement une seconde fois les clusters ayant des barycentres encore trop proches, ce qui nous permettrait de regarder si un des clusters finaux représente la classe frauduleuse.