Rapport de stage : projet fin d'étude

Sciences et Technologies de l'Information et de la Communication

Année : 2020/2021

Détection de la fraude bancaire avec l'apprentissage automatique et la théorie des jeux

---

Le contenu de ce rapport n'est pas confidentiel

Auteur : Mohamed Ali Elbouri ***Promotion :*** 2021

Enseignant référent ENSTA : M. Zacharie Ales

Tuteur organisme d'accueil : M. James Kouthon

Stage effectué du 26/04/2021 au 3/11/2021

Nom de l'organisme d'accueil : Acensi Finance

Adresse : 14 Rue du Général Audran, 92400 Courbevoie, France

Remerciement

Je tiens à remercier toutes les personnes qui m'ont aidé de proche ou de loin, lors de ce stage, ainsi lors de toutes mes études.

Je voudrais remercier dans un premier temps mon tuteur d'entreprise M. James Kouthon pour son encadrement et son soutien durant ce stage, ainsi que sa confiance. À travers ses remarques et ses conseils, il a bien orienté ce projet et a assuré le bon déroulement de ce stage.

Je remercie M. Zacharie Ales pour avoir accepté d'être mon tuteur académique dans ce projet.

Enfin, j'adresse un grand merci à ma famille et mes proches pour leur soutien continu et inconditionnel.

Résumé

La fraude à la carte crédit cause des Milliards de Dollars de pertes chaque année [3]. Dans ce projet, on construit la fondation d'un système de prédiction de fraude à la carte crédit. Les résultats présents dans ce rapport sont basées sur des observations faites en utilisant une base de données des fraudes artificielles.

Dans un premier temps, on sélectionne plusieurs modèles de *Machine Learning*. Après, on optimise cette sélection. On compare ainsi leurs performances et leurs temps de réponse. Finalement, on applique une approche basée de la Théorie des Jeux, dans l'objectif de construire des modèles plus robustes.

Mots-clés : Apprentissage Automatique, Fraude, Carte Crédit, Données Déséquilibrées, Théorie des Jeux, Apprentissage Adverse

Abstract

Credit Card Fraud causes Billions of Dollars of loss each year [3]. In this project, we aim to build an effective Credit Card Fraud Prediction System. The work described in this report is based on observations made with the use of an artificially generated dataset.

In the first step, we compare some Machine Learning models. We then optimize a selected group of these models to obtain better performances. We also compare the execution time of these selected models. We finally apply and study a Game Theoretical-based approach with the goal of building more robust models.

Keywords : Machine Learning, Fraud, Credit Card, Unbalanced Data, Game Theory, Adversarial Learning

Table des matières

Table des figures

Chapitre I.

Introduction

La fraude bancaire cause des pertes immenses pour les entreprises du monde entier. En particulier [3], la fraude à la carte de crédit a causé des pertes de 24.26 Milliards de Dollars en 2018. En plus de ça, ce montant (en 2018) représente une hausse de 18.4% par rapport à 2017, ce qui reflète l'incapacité des entreprises en question de trouver des solutions efficaces contre la fraude à la carte de crédit. Ceci nous amène à notre objectif principal : créer un système efficace de détection de fraude à la carte bancaire pour minimiser les coûts économiques de ce phénomène.

Les algorithmes de *Machine Learning* ont montré leur efficacité dans plusieurs domaines. On peut observer l'évolution des outils développés avec du *Machine Learning* ou les techniques d'*Intelligence artificielle* en général à travers un *Smartphone* : la reconnaissance faciale, les filtres, reconnaissance de voix et de mots, la traduction ... Par conséquent, on s'intéresse dans un premier temps à l'efficacité des méthodes de *Machine Learning* contre ce type de fraude.

Cependant, même avec un modèle de *Machine Learning* qui semble efficace au temps du développement, les performances peuvent se dégrader à court ou à long terme après le déploiement. En effet, lorsqu'on entraîne ou développe un modèle de *Machine Learning*, on fait l'hypothèse que les données d'entraînement et les données réelles sont générées à partir de la même distribution des données [13]. Dans la réalité, les données évoluent, et c'est assuré dans notre cas par le comportement dynamique du fraudeur. Pour cela, on fait recours à des techniques de la *Théorie des Jeux* pour minimiser les dégâts résultants de ce comportement.

Ce rapport va comporter 3 parties majeures : on va montrer en premier temps les performances de quelques modèles de Machine Learning. Après, on va optimiser les meilleurs modèles rencontrés et on va étudier leurs temps de réponse. Finalement, on va étudier l'apport d'une approche basée sur la *Théorie des Jeux*.

Chapitre II.

Cadre de stage

1. Entreprise d'accueil : Acensi Finance

Acensi Finance fait partie du groupe *Acensi* qui se spécialise dans le conseil au secteur informatique et fournit des services numériques, ce groupe était créé en 2003. Avec 1300 collaborateurs, *Acensi* accompagne 90 clients dans la gestion de leurs projets IT et atteint un chiffre d'affaires de 130 Millions d'Euros [1]. Le siège social d'*Acensi* est situé à Courbevoie, île-de-France. En 2010, *Acensi* est devenu un groupe international avec ses deux branches à Belgique et au Canada.

Les services d'*Acensi* sont divisés en quatre :

— Le conseil : transformation digitale (Big Data, Cloud), accompagnement (gouvernance des projets, PMO), Expertise (BPM, CRM) et méthodologies (Agile, Devops).

— BI et Big Data : Restitution, Flux et qualité (Profilage, Modélisation, Cleaning).

— Technologies et Digital : Développement Objets (Front End, Jenkins), Web et Mobilité.

— Infrastructure : SI, Télécom et Cloud Computing.

Le groupe *Acensi* travaille sur plusieurs secteurs, notamment la *Télécommunication*, l'*Énergie*, la *Banque et Finance de marché* et d'autres secteurs. *Acensi Finance* gère le secteur *Banque et Finance de marché*, cette entreprise a été créée en 2006. L'effectif d'*Acensi Finance* est de 250 salariés et cette entreprise à réalisé un chiffre d'affaires [2] de 22 Millions d'Euros.

1. En 2020 selon [2]
2. En 2019 selon [1]

2. Objectifs et motivations

Comme évoqué dans l'introduction, notre objectif principal consiste à créer un système efficace de détection de fraude à la carte bancaire pour minimiser les coûts économiques de ce phénomène. Cependant, il y a plusieurs caractéristiques qui rendent la fraude à la carte bancaire difficile à détecter :

— *Données déséquilibrées* [24] : les transactions normales forment la majorité des données et les transactions frauduleuses sont relativement rares. Ce déséquilibre rend la bonne classification par des systèmes basés sur l'apprentissage automatique plus dure, et aussi nécessite un bon choix d'indicateurs pour bien évaluer la performance [3].

— *Comportement dynamique* [24] : quand on essaie de créer un système de détection de fraude, les fraudeurs s'adaptent rapidement et trouvent des manières pour échapper à la détection. Ce comportement dynamique ou adaptatif crée ce qu'on appelle un *Adversarial Environment* [4], ce qui cause la dégradation des performances du système.

— *Mimic behaviour* [24] : le fraudeur essaie d'imiter le comportement d'un utilisateur normal pour donner l'impression que ses transactions sont légitimes. Du coup, si on essaie de considérer les transactions frauduleuses comme anomalies, on n'arrivera pas à détecter les fraudes qui sont bien déguisées comme transactions normales.

— *Réponse en temps réel* : en faisant un achat, le client veut que la transaction soit approuvée rapidement, on ne peut pas par exemple prendre quelques minutes pour répondre à chaque transaction. En fait, le temps alloué à la prédiction doit être à l'ordre de ms (10^{-3} seconde).

— *Datasets réelles* : pour des raisons liées à la confidentialité, il est difficile de trouver des bases de données contenant des transactions réelles. Pour cela, on a utilisé deux datasets : la première contient des transactions réelles cryptées et la deuxième contient des transactions générées par une simulation.

Dans ce projet, on parcourt ces problèmes un par un, tout en documentant les résultats des approches utilisées, et on met enfin la fondation d'un système de détection de fraude efficace.

3. On explique le choix des indicateurs dans la partie 1.
4. On entre plus dans les détails dans la section VI.

Chapitre III.

Datasets

Dans le cadre du travail d'équipe, la recherche des bases de données et l'analyse de données exploratoire qu'on va présenter dans ce chapitre ont été effectuées par un collègue avant mon arrivée. On va présenter cette étape pour mieux comprendre les autres phases du projet.

1. Données réelles cryptées

Pour la première partie du stage, on a essayé des algorithmes de *Machine Learning* et des techniques de *Sampling*[1] sur une base de données de transaction réelles cryptées (lien kaggle).

On ne va pas entrer dans les détails de cette base de données, car on l'a utilisée pour essayer les différentes approches, pas pour faire des comparaisons entre les modèles. Elle contenait 28 colonnes nommées V_i pour i de 1 à 28, une colonne *Time*, une colonne *Amount* et une colonne *Class* (0 ou 1 pour indiquer si fraude ou pas). Cette base de données représente des données réelles d'une banque transformées avec le *PCA* (Analyse en Composantes Principales) pour des raisons de confidentialité.

2. Données de fraude artificielles

2.1. Introduction générale

Cette base de données est générée avec un simulateur [16] et on l'a obtenue à partir de ce lien Kaggle. Au contraire de la première base de données, son contenu n'est pas réel, mais les noms des colonnes ne sont pas chiffrés et on trouve des valeurs similaires aux transactions réelles .

1. On va entrer dans les détails dans le chapitre suivant

La base de données est composée de deux fichiers *csv* :

— *fraudTrain.csv* : 1 296 675 lignes et 22 colonnes, ce fichier comporte des transactions bancaires (artificielles) entre le 01/01/2019 et le 21/06/2020

— *fraudTest.csv* : 555 719 lignes et 22 colonnes, ce fichier comporte des transactions bancaires (artificielles) entre le 21/06/2020 et le 01/01/2021

Dans cette base de données, il n'y a pas des valeurs manquantes (*missing values*), donc on n'a pas implémenté une stratégie de remplissages des valeurs vides. On a 21 attributs et une colonne qui concerne la classe *is_fraud* : elle prend soit 0, soit 1, c'est un problème de classification binaire.

2.2. Analyse de données exploratoire (EDA)

Avant de commencer à entraîner des modèles d'apprentissage automatique sur les données, on a fait une *EDA* pour avoir une première intuition sur l'importance de chaque attribut et l'interaction des attributs entre eux.

2.2 1 Densités

Dans cette partie, on explique le contenu de chaque colonne de la base de données de fraude artificielles 1.. On montre aussi les densités de chaque attribut dans *trainFraud.csv* :

— **trans_date_trans_time** : la date et le temps de la transaction, dans le format année-mois-jour heure :minute :seconde (Y-M-D H :m :s). Il contient des dates entre le 01/01/2019 et le 21/06/2020.

— **cc_num** : numéro de la carte crédit, c'est un entier.

— **merchant** : le commerçant, c'est une chaîne de caractères. Il comporte 693 valeurs uniques.

— **category** : la catégorie d'achat, c'est une chaîne de caractères. Il contient 14 catégories différentes.

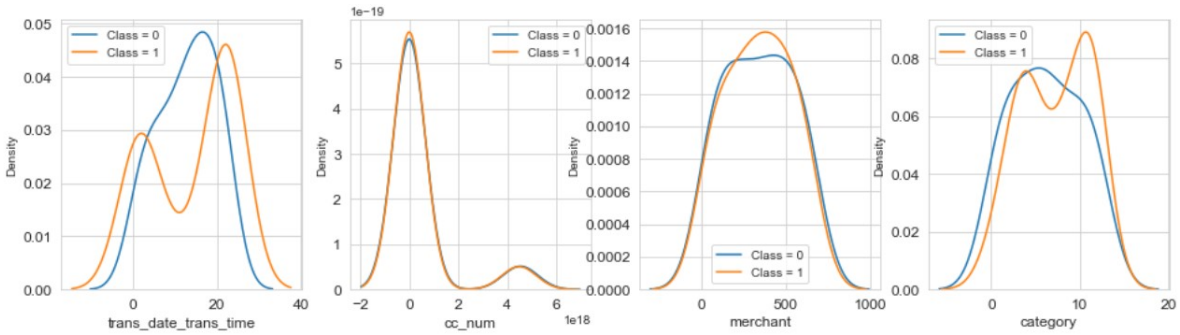

FIGURE 1 – Densités de trans_date_trans_time, cc_num, merchant et category

— **amt** : le montant de la transaction, c'est un réel positif. Sa valeur minimale est 1, sa valeur maximale est 28 948.90.

— **first** : le prénom du porteur de la carte, c'est une chaîne de caractères.

— **last** : le nom de famille du porteur de la carte, c'est une chaîne de caractères.

— **gender** : le genre du porteur de la carte, c'est soit F (femme) ou M (homme).

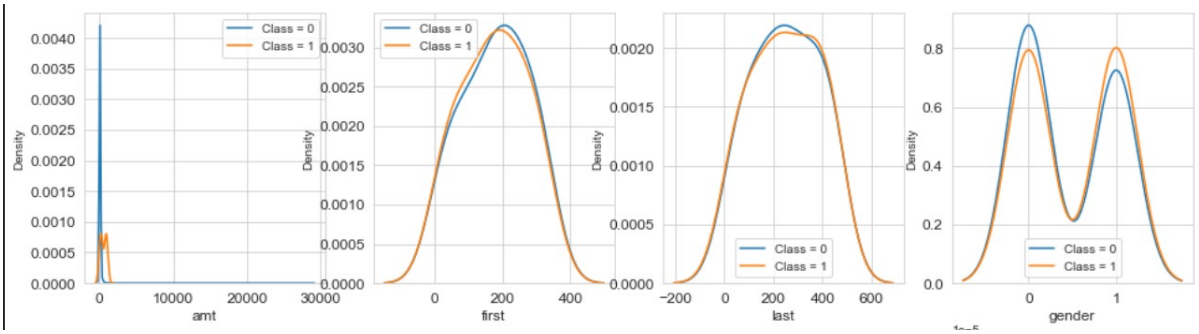

FIGURE 2 – Densités de amt, first, last et gender

— **street** : la rue où habite le porteur de la carte, c'est une chaîne de caractères. Il contient 983 valeurs uniques.

— **city** : la ville ou habite le porteur de la carte, c'est une chaîne de caractères. Il contient 894 valeurs uniques.

— **state** : l'état ou habite le porteur de la carte, c'est une chaîne de caractères. Il contient 50 valeurs uniques (les états dans les États-Unis).

— **zip** : le code postal du porteur de la carte, c'est un entier.

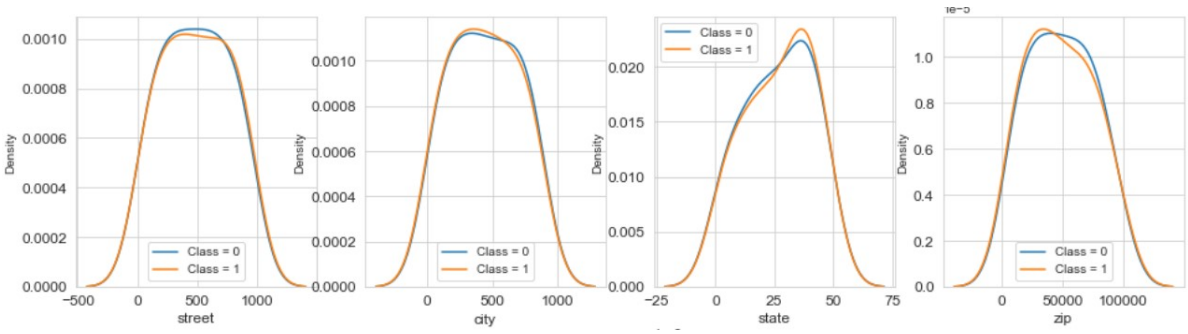

FIGURE 3 – Densités de street, city, state et zip

— **lat** : la latitude de l'adresse du porteur de la carte, c'est un réel.

— **long** : longitude de l'adresse du porteur de la carte, c'est un réel.

— **city_pop** : le nombre de personnes qui habitent dans le la ville du porteur de la carte, c'est un entier.

— **job** : le travail du porteur de la carte, c'est une chaîne de caractères. Il contient 494 valeurs différentes.

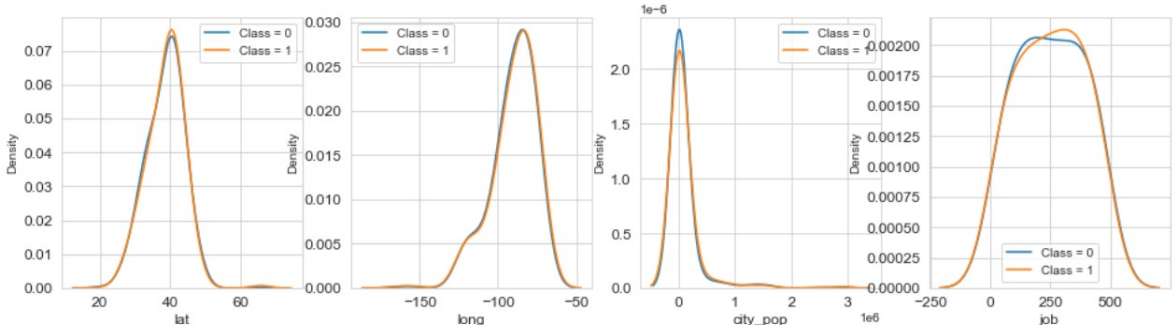

FIGURE 4 – Densités de lat, long, city_pop et job

— **dob** : la date de naissance du porteur de la carte, dans le format année-mois-jour (Y-M-D). Il contient des dates entre le 30/10/1924 et 29/01/2005.

— **trans_num** : le numéro de la transaction, c'est une chaîne de caractères.

— **unix_time** : le temps *unix* de la transaction, c'est le nombre de secondes écoulées depuis 01/01/1970, c'est un entier.

— **merch_lat** : la latitude du commerçant, c'est un réel.

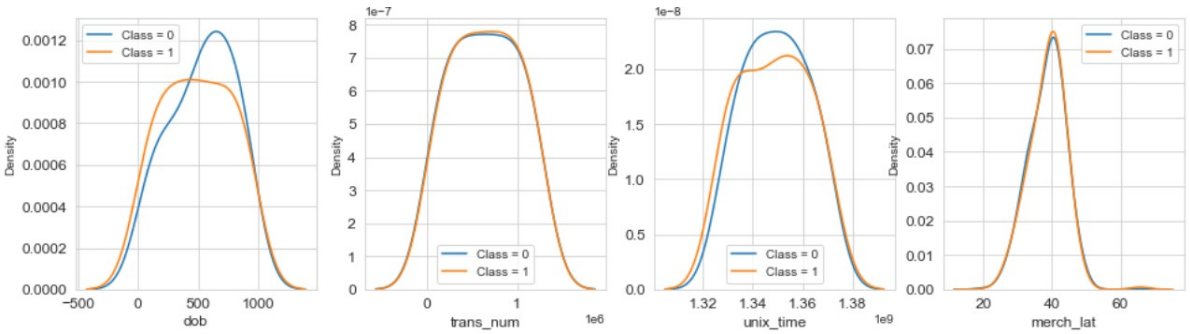

FIGURE 5 – Densités de dob, trans_num, unix_time et merch_lat

— **merch_long** : la longitude du commerçant, c'est un réel.

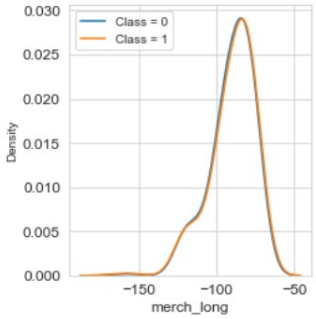

FIGURE 6 – Densité merch_long

— **is_fraud** : le label de la transaction, 0 si légitime et 1 si frauduleuse. On trouve 1 289 169 transactions légitimes contre 7506 transactions frauduleuses.

2.2 2 Corrélation

Dans la figure 7 on observe la corrélation entre les colonnes numériques de la base de données. Si on regarde les corrélations avec le *target (is_fraud)*, on trouve que *amt* (le montant) est le plus corrélé avec *is_fraud* (la corrélation est 0.23). Le fait que *amt* est le plus corrélé reflète son importance dans la prédiction. On peut aussi remarquer la forte corrélation entre *lat* et *merch_lat* et aussi *long* et *merch_long*, donc il sera envisageable dans le futur de supprimer soit *lat* et *long*, soit *merch_lat* et *merch_long*.

2.2 3 Boxplot du montant

La figure 8 montre la différence des montants dépensés entre les transactions frauduleuses et légitimes. À gauche, on affiche les valeurs extrêmes (ou *outliers*), et on remarque que la valeur maximale des transactions légitimes (arrivant à 30 000) est supérieure à

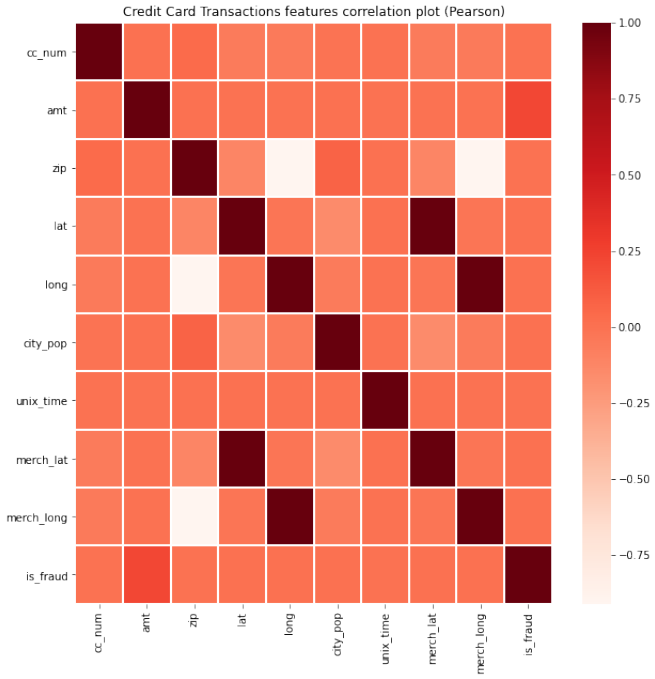

FIGURE 7 – Corrélation entre les colonnes numériques

celle des transactions frauduleuses (qui ne dépassent pas 2000), on peut expliquer ça par le renforcement d'authentification en cas de grands montants. À droite, on observe que la moyenne des dépenses pour les transactions frauduleuses dépasse la moyenne des dépenses des transactions légitimes.

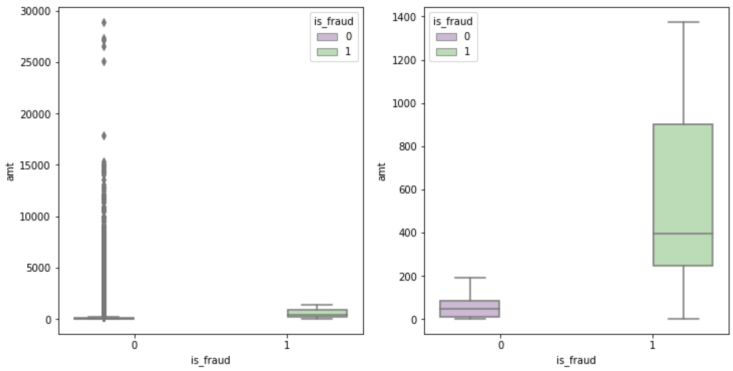

FIGURE 8 – Boxplot des montants dépensés pour chaque classe (0 : non-fraude, 1 : fraude) : avec et sans outliers (à gauche et à droite respectivement)

Chapitre IV.

Développement des premiers modèles

1. Critères de comparaison

Dans les deux bases de données considérées, le taux de fraude ne dépasse pas 1%. En posant un modèle naïf, qui prédit toujours 0 (pas fraude), on aura un *accuracy* qui dépasse 99% (puisqu'on a bien classé toutes les transactions légitimes). Par conséquent, on ne peut se contenter à mesurer les performances avec l'*accuracy*, et on va introduire les 3 indicateurs suivants :

- Précision de la classe 1 : TP/(TP + FP) [1], la précision de prédire 1 (fraude)
- Recall de la classe 1 : TP/(TP + FN) [2], le taux des fraudes détectées
- F1 score de la classe 1 : dépend des deux derniers indicateurs, c'est notre indicateur de choix qui permet de trier les modèles

$$f1score = 2 \times \frac{recall \times précision}{recall + précision}$$

Dans le reste du rapport, tous les résultats sont issus de la base de données de fraude artificielles 2..

2. Algorithmes d'apprentissage automatique

Dans cette partie, on va introduire les modèles d'apprentissage automatique qu'on a utilisés. Dans cette étape, on a essayé plusieurs algorithmes dans le but de créer un *Benchmark* initial, qui contient les performances de chaque modèle. Cependant, on

1. TP : True positives, FP : False positives
2. TP : True positives, FN : False negatives

n'a pas poussé les performances jusqu'au bout, on visait à obtenir un premier aperçu, seulement les modèles les plus performants ont bénéficié par la suite d'une phase d'optimisation dans (V.).

2.1. Modèles supervisés

Dans ce paragraphe, on va parler des modèles supervisés qu'on a utilisés. En premier lieu, un modèle supervisé est un modèle qui utilise des données labelisées, et utilisent ce *label* dans le processus d'apprentissage. Dans notre cas, le label est *is_fraud*. Les modèles utilisés sont les suivants :

— *Forêt aléatoire (Random Forest)* [18] : cet algorithme consiste à définir plusieurs arbres de décisions, ces arbres contribuent tous à formuler enfin une prédiction finale. Chaque arbre est entraîné sur une partie différente de donnée (les parties sont choisies avec tirage au hasard avec remplacement du dataset initial). Les arbres font un vote majoritaire pour choisir la classe prédite. Ce processus est plus performant qu'utiliser un seul arbre, car il permet de minimiser la variance : d'où réduire le risque l'*Overfitting*, en revanche c'est plus coûteux en termes de complexité temps.

— *Extreme Gradient Boosting (XGBoost)* [8] : XGBoost est un algorithme basé sur le principe de *gradient boosting*. Il utilise des arbres qui ne sont pas très complexes (*weak learners*) qu'il construit d'une manière séquentielle. Dans le *boosting*, à chaque étape, les erreurs de prédictions de l'itération précédente ont plus de poids, pour que l'arbre actuel essaie de les corriger.

— *K-Plus-Proches voisins (K-Nearest-Neighbours)* [7] : cette méthode se compose de deux phases. Phase d'apprentissage : le modèle enregistre les instances d'entraînement sans apporter des modifications. Phase de prédiction : pour une instance donnée, le modèle cherche les k plus proches voisins en utilisant une métrique définie (distance euclidienne par défaut pour *Sci-kit Learn*). La classe prédite est celle présente en majorité parmi ces k plus proches voisins.

— *Réseaux de Neurones (Neural Network)* [23] : ils contiennent un ensemble de couches : couche d'entrée, une ou plusieurs couches cachées et une couche de sortie. Le processus d'apprentissage est divisé sur 2 parties. En premier lieu, on fait le *Forward pass* : appliquer des multiplications par des matrices et des fonctions souvent non linéaires. La deuxième phase consiste à évaluer les résultats du *Foward pass*, et mettre à jour les poids des couches pour avoir des résultats plus précis dans le prochain *Forward pass* : c'est le *Back propagation*.

— *Long-Short-Term-Memory (LSTM)* [20] : c'est un type de réseaux de neurones et plus spécifiquement les *RNNs : Recursive Neural Networks*. Les *RNNs* sont capables de travailler sur des séquences de données, cependant, dans des séquences

longues, quelques informations peuvent converger vers 0 (ou diverger) à cause de la dérivation (*Back Propoagation*). Les *LSTMs* contiennent plus de cellules pour faire face à ce problème, ces cellules définissent quelles informations à garder à long terme et quelles informations à oublier.

2.2. Techniques utilisées

Les techniques utilisées dans cette partie concernent la majorité des modèles utilisés. On va expliquer ces techniques dans ce paragraphe et puis donner plus de détails sur les traitements spécifiques pour chaque algorithme dans le paragraphe suivant.

2.2 1 Preprocessing

Avant de passer à l'entraînement des modèles, les données doivent passer avant par une étape des preprocessing :

— *StandardScaler* (documentation) [6] : ça consiste de faire un traitement sur les données pour rendre la moyenne des valeurs de chaque attribut nulle et la variance unitaire. Ce traitement permet d'éviter de trouver des valeurs très grandes ou très petites (par rapport à 1) et causer des problèmes : soit de computation (divergence ou convergence vers 0 dans le cas de *NN* par exemple), soit dans le calcul des distances entre les points (les algorithmes qui utilisent des distances comme *KNN*).

$$X = (X - u)/s$$

$$\text{avec : } \begin{cases} u = \frac{1}{n}\sum_{i=1}^{n} X_i \\ s = \sqrt{\sum_{i=1}^{n}(X_i - u)^2} \end{cases}$$

— *LabelEncoder* (documentation) [4] : il permet de transformer les variables catégoriques (non numériques) vers des entiers. Avec le *LabelEncoder*, on garde le même nombre de colonnes, mais on définit une relation d'ordre entre les valeurs transformées (ça impacte en particulier les algorithmes basés sur des distances).

— *OneHotEncoder* (documentation) [5] : pour une variable catégorique de m valeurs uniques, le *OneHotEncoder* crée m cases, ou chaque case contient 0 ou 1 pour indiquer si cette valeur est prise par l'instance transformée (pour chaque instance, on trouve alors une seule case qui prend 1, et m-1 cases qui prennent 0) : cette technique est plus coûteuse en mémoire, mais ne définit pas une relation d'ordre.

Avant d'appliquer ces outils de *preprocessing*, on doit tout d'abord choisir les attributs concernés, ce qui nous amène à la partie suivante : *feature selection*.

2.2 2 Feature selection

La sélection des attributs est basée sur l'analyse des données exploratoire dans un premier temps, puis validée à partir des résultats sur le *Validation Set*. On essaie alors plusieurs représentations de données, en variant les attributs sélectionnés et aussi le *pre-processing* :

— Valeurs numériques uniquement : comme *baseline* pour quelques algorithmes, on a choisi d'entraîner nos modèles uniquement sur les valeurs numériques transformées avec le *StandardScaler*.

— Une deuxième représentation consiste à appliquer le *LabelEncoder* sur **category**, **gender**, **state** et **merchant**, transformer **dob** (date of birth : type date) vers un entier (age), choisir **city_pop, unix_time, zip et amt** parmi les variables numériques et finalement appliquer le *StandardScaler*. En faisant ce choix, on a utilisé que 9 colonnes, et on a obtenu des résultats meilleurs que la première représentation avec 9 variables numériques. Les attributs éliminés sont :

  — **cc_num** : numéro de la carte bancaire, puisqu'on a plusieurs numéros de cartes, il y a beaucoup de variabilité dans cet attribut. On observe aussi dans l'*EDA* que la densité pour les fraudes et non-fraudes est pratiquement la même en variant la valeur de **cc_num**.
  — **trans_date_trans_time** : c'est équivalent à *unix_time*.
  — **first** et **last** : ce sont des variables catégoriques qui indiquent le nom et prénom sur la carte bancaire. En faisant un *OHE* on aura plus de 200 000 colonnes supplémentaires, et si on applique le *LE* on va introduire une relation d'ordre entre les noms qui n'existe pas en réalité. De plus, dans l'*EDA* avec le *LE* appliqué sur ces deux attributs, on observe les mêmes densités pour les fraudes et non-fraudes.
  — **street** et **city** : le *OHE* ne sera pas pratique (plus que 900 et 800 valeurs uniques respectivement). Avec le *LE*, on ne trouve pas une variation entre les densités des fraudes et non-fraudes. De plus, le **zip**, **state** et **city_pop** peuvent factoriser quelques informations présentes dans ces deux colonnes.
  — **lat**, **long**, **merch_lat** et **merch_long** : on a décidé d'éliminer **merch_lat** et **merch_long** à cause de la forte corrélations avec **lat** et **long**. **lat** et **long** ne semblent pas apporter des informations supplémentaires : densités en *EDA* et relation avec **zip** et **state**.
  — **trans_num** : c'est le numéro de transaction qui est un variable catégorique. Cette variable est unique pour chaque transaction, et les densités des fraudes et non-fraudes dans l'*EDA* ne montrent pas d'intêret pour l'utiliser.
  — **job** : c'est le travail du porteur de la carte, c'est un variable catégorique avec 494 valeurs uniques. Un *OHE* ne sera pas pratique. Un *embedding* ou une transformation spécifique des différentes valeurs de **job** peuvent être utiles pour factoriser ces 494 *job* en quelques groupes (selon domaine par exemple)

ou avec une relation d'ordre (selon salaire moyen par exemple).

Ces deux premières approches étaient utilisées sur *Random Forest* et *KNN*. On a défini 6 configurations spécifiques (appliquées sur NN, XGB et éventuellement RF et SVM). On va évoquer les deux premières (étant les plus efficaces). Le reste des configurations sont expliquées dans la documentation du code.

- Pour les deux configurations, on élimine **trans_date_trans_time**, et **trans_num**, **cc_num**. On transforme **dob** vers un entier (age). Finalement, on effectue le *LE* sur **gender** et **merchant** (il y a 693 valeurs uniques de ce dernier attribut, *OHE* n'est pas pratique).
- **config_1_1** : pour les variables qui restent, on garde toutes les variables numériques, et on garde que **category** et **state** parmi les variables catégoriques. Dans cette configuration, on effectue le *OHE* sur **category** et **state**.
- **config_1_2** : on utilise les mêmes attributs, la seule différence, c'est qu'on applique le *LE* sur **category** et **state**.

À ce stade-là, on n'a pas effectué des traitements de *Feature Engineering*. On effectuera ces traitements dans la partie V..

2.2 3 Sélection des paramètres

Pour chaque modèle traité, il y a des paramètres à configurer, et pour obtenir des meilleurs résultats, on doit les calibrer sur nos données. Les techniques qu'on a utilisées pour le choix des paramètres sont :

- *GridSearchCV* (documentation) : c'est une fonction prédéfinie par *Scikit-learn*, elle sert à donner le meilleur modèle selon un indicateur (f1 score de la classe 1 dans notre cas) et en essayant toutes les combinaisons des paramètres qu'on définit manuellement. *GridSearchCV* calcule le score de chaque modèle en faisant un *Cross Validation* pour avoir des résultats plus proche de la réalité et éviter l'*Overfitting*.

- *Variation des paramètres* : lorsqu'on a plusieurs paramètres à calibrer avec plusieurs valeurs, le *GridSearchCV* devient lent, puisqu'il essaie toutes les combinaisons. À titre d'exemple, avec 5 paramètres, et 10 valeurs pour chaque paramètre : le *GridSearchCV* doit tester 10^5 modèles. Aussi avec le *Cross validation*, si on prend k=5, on entraîne chaque modèle 5 fois, et en total on va faire le processus d'entraînement 500 000 fois.
  Si on choisit de varier chaque paramètre seul, on va tester $5 \times 10 = 50$ modèles (250 éventuellement si on fait le *Cross Validation*). On observe comment les performances varient pour chaque paramètre pour faire nos choix après. Cette approche n'est pas aussi exhaustive que le *GridSearchCV*, puisque les paramètres ne sont pas indépendants, mais on gagne beaucoup en termes de temps de calcul.

2.3. Traitements spécifiques

Comme on a évoqué dans les parties précédentes, on doit faire des choix concernant les valeurs des paramètres spécifiques à chaque algorithme, ainsi que la représentation des données la plus adéquate. Donc pour pouvoir comparer les performances conséquentes des différents choix, on a divisé le *fraudTrain.csv* en 2 parties : *Train set* (80%) et *Validation Set* (20%). La *Train set* est utilisée pour entraîner les modèles, et la *Validation set* pour faire les choix [3].

2.3 1 Random Forest et KNN

Pour ces deux algorithmes, on a fait une comparaison entre l'utilisation des attributs numériques et ces attributs : **category**, **gender**, **state**, **merchant**, **dob**, **city_pop**, **unix_time**, **zip** et **amt**. En dexuième lieu, on a appliqué l'*Undersampling* [4] sur les données (pour réduire la taille), on a effectué un *GridSearchCv* sur les data de taille réduite (pour avoir un temps d'exécution plus petit). Les paramètres considérés pour *Random Forest* étaient : **n_estimators**, **max_depth**, **min_samples_split** et **min_samples_leaf**. Pour *KNN* on a essayé plusieurs valeurs de **K**.

Les paramètres optimaux pour *Random Forest* étaient : **n_estimators**=140, **max_depth**=30, **min_samples_split**=2 et **min_samples_leaf**=3. On a obtenu **k**=21 pour *KNN*.

2.3 2 XGBoost

Pour le *XGBoost*, on a commencé par choisir la meilleure représentation des données parmi 6 configurations (**Config_1_1** et **Config_1_2** mentionnées en 2.2 2 et 4 autres configurations). En premier lieu, on entraîne le modèle sur **X_train** et on montre les performances sur **X_val**. On fait la comparaison en utilisant le *f1 score* de la classe 1, mesuré sur **X_val** : en finale, on a choisi **Config_1_1**.
Maintenant qu'on a choisi une représentation de données, on essaie de trouver les paramètres optimaux en variant chaque paramètre seul. À chaque étape, on modifie la valeur du paramètre **p**, on choisit la valeur par défaut et la valeur optimale (avec le meilleur f1 score sur le *Validation Set*). Les paramètres qu'on a variés sont [9] :

- **min_child_weight** : l'importance ou le poids de la classe minoritaire (par défaut, c'est 1, comme la classe majoritaire).
- **max_depth** : profondeur maximale des arbres. Ce paramètre intervient pour contrôler la complexité des arbres : trop petit on peut tomber dans l'*Underfitting*, trop grand on risque l'*Overfitting*.
- **eta** : c'est le learning rate.

3. Dans le cas de NN, RF et KNN : on a utilisé un Cross validation sur *fraudTrain.csv* tout entier sans faire cette division
4. Définition en 3.2.

— **gamma** : c'est un paramètre de régularisation qui contrôle la profondeur. Avec gamma très grand, on coupe les arbres plus rapidement et on trouve des arbres moins complexes.
— **n_estimators** : c'est le nombre d'arbres utilisés.

On finit par fixer les valeurs suivantes : **min_child_weight**=1, **max_depth**=7, **eta**=0.05, **gamma**=0.05 et **n_estimators**=50.

2.3 3 Neural Network

Comme pour *XGBoost*, le premier choix était la représentation des données. On a fini par choisir **config_1_1**.
La deuxième étape consistait à trouver les valeurs optimales de **batch_size** et **learning_rate**. Pour cela, on a implémenté une expérience de *Cross validation*. Les résultats sont présentés dans la figure 9.

Average results for val_f1

	0.0001	0.0010	0.0100
32	0.397891	0.556302	0.124167
128	0.251931	0.566706	0.502611
512	0.357037	0.579305	0.573378
1024	0.274436	0.576542	0.620776

FIGURE 9 – f1 score 1 : batch size en lignes, learning rate en colonnes

On choisit alors **batch_size**=1024 et **learning_rate**=0.01.

2.3 4 LSTM

Pour cet algorithme, le premier prototype consistait à utiliser les variables suivantes : *amt* pour indiquer les variations du montant, *unix_time* pour indiquer le temps de la transaction et finalement *merch_long* et *merch_lat* pour l'emplacement géographique. On a divisé les transactions en des séquences de taille 100. Chaque séquence contient des transactions d'une seule personne (même *cc_num*) ordonnées. On fait une prédiction *Many to Many* : pour chaque séquence, le modèle retourne une liste de 100 éléments (0 ou 1) pour indiquer la classe de chaque transaction de la séquence.

On a travaillé avec un **batch_size** = 32, un **learning rate** par défaut et avec l'optimiseur *Adam* pour 20 epochs. En premier lieu, on a utilisé **StandardScaler** sur les 4 colonnes choisies. On a aussi essayé **MinMaxScaler** (documentation) et on a constaté

que pour *LSTM* en utilisant ces 4 attributs, c'était plus pertinent d'utiliser ce dernier.

Finalement, on a obtenu un *f1 score* qui est relativement élevé (par rapport aux autres modèles) en utilisant que 4 colonnes. Cela montre l'importance du contexte (historique de l'utilisateur) pour la classification, on développe sur cette idée davantage dans le chapitre V..

2.4. Kmeans

Principe de Kmeans [10] : pour k donné, cet algorithme regroupe les données fournies en k groupes appelés *clusters*, tout en essayant de minimiser la somme des distances (euclidiennes par exemple) entre les points et leurs centres de *clusters* respectifs. Ce type d'apprentissage est non supervisé : l'algorithme n'utilise pas le *label* pour regrouper les données.

En faisant le *clustering*, on est partie de deux hypothèses (qui rassemblent aux hypothèses mentionnées dans [25]) :

— **H1 :** les transactions frauduleuses et les transactions légitimes vont appartenir dans des *clusters* différents.

— **H2 :** les transactions légitimes vont être condensées et plus proches du centre de chaque *cluster*, et les transactions frauduleuses seront situées sur les bords des *clusters*, comme des anomalies.

Pour étudier ces deux hypothèses, on doit avant décider le nombre de *clusters*, on a fait recours à l'*elbow method*. On a calculé pour les différents nombres de *clusters* considérés le *WCSS : Within Cluster Sum of Squares*. On cherche à minimiser cette somme pour avoir des *clusters* plus condensés sans utiliser beaucoup de *clusters*, d'où l'utilisation de l'*elbow method* : on choisit le nombre de *clusters* où on trouve qu'on n'aura pas une amélioration considérable (en termes de minimisation de *WCSS*) en ajoutant 1 *cluster* supplémentaire. La figure 10 montre l'évolution de *WCSS* en fonction du nombre de *clusters*. À partir de cette courbe, ce n'est pas évident quelle valeur choisir, donc on a décidé d'essayer avec deux valeurs : 9 et 16. Dans le reste de cette partie, on va traiter le cas avec 9 *clusters* à titre d'exemple.

Pour la première hypothèse : on a calculé les pourcentages de fraude dans chaque *cluster* et le plus grand pourcentage était 34%. Donc on n'a aucun *cluster* de partie majoritaire fraude.
Pour la deuxième hypothèse, on a affiché des histogrammes de distances des points frauduleux et légitimes par rapport aux centres de leurs *clusters*. On présente les résultats du *cluster* 8 à titre d'exemple dans la figure 11.

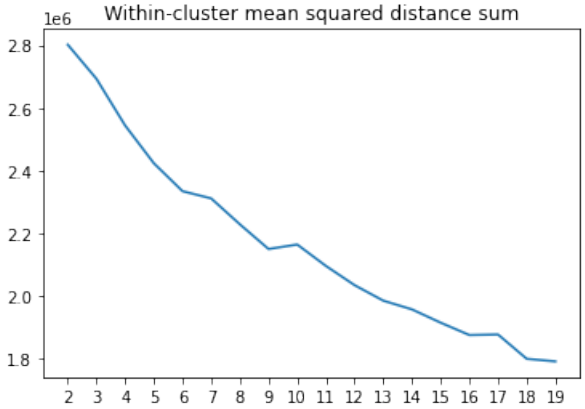

FIGURE 10 – WCSS en fonction du nombre de clusters

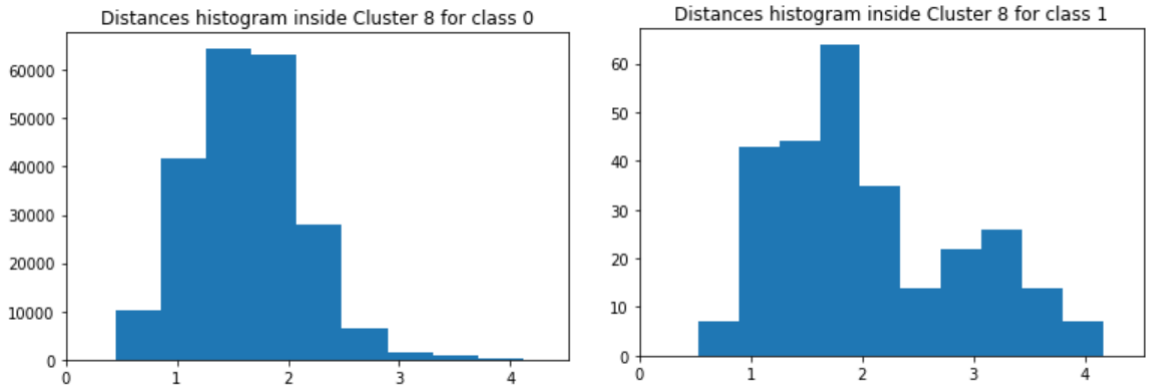

FIGURE 11 – Distances au centre : non fraudes-fraudes

Dans ce *cluster* (appelé 8) et dans le reste des *clusters*, on observe que les fraudes ne sont pas situées sur les bords de *cluster*. On inspecte cette hypothèse de plus, en utilisant le *Test set* : chaque point est associé au *cluster* le plus proche (en utilisant la distance euclidienne avec les centres de *clusters*). On compare cette distance avec la distance moyenne des points de ce *cluster* : elle est considérée comme frauduleuse si elle est située plus loin de la distance moyenne. On obtient alors un *recall* de 67% et une *précision* de 0.2%. On voit qu'en utilisant un seuil relativement bas (pour définir les bords) comme la moyenne, on ne capte que 67% des fraudes, et on obtient une précision très faible. Si on augmente ce seuil, on aura un *recall* plus petit.

Pour conclure cette partie, avec la distance euclidienne et en utilisant certains attributs, on n'arrive pas à valider l'hypothèse 1 ou 2. Cependant, il peut être utile d'améliorer cette approche en utilisant d'autres métriques, ou d'autres attributs. En plus de ça, on peut utiliser les informations des *clusters* pour aider les autres modèles (comme dans le

cas de *cluster* à 34% de fraudes) à faire les prédictions.

3. Techniques de sampling

Pour traiter le problème de déséquilibre, on a fait recours aux méthodes d'Oversampling et Undersampling. Dans cette partie, on fait des expériences pour observer l'effet de ces méthodes sur nos 3 critères de comparaisons (dont f1 score est le critère décisif).

3.1. Oversampling

L'Oversampling consiste à ajouter des instances à la classe minoritaire pour atteindre un ratio souhaité appelé *Sampling strategy*. À titre d'exemple, si on a 10 instances de fraude et 1000 instances légitimes, avec un *Sampling strategy* de 0.5 on va obtenir 500 instances de fraude tout en gardant les 1000 instances légitimes.
Avec un ratio plus élevé, on donne plus d'importance à la classe minoritaire, mais on risque de générer des instances qui ne représentent pas d'une façon fidèle cette classe.
On a utilisé 3 techniques d'Oversampling : **Random Oversampling** (documentation), **SMOTE** (documentation) et **ADASYN** (documentation).

3.2. Undersampling

L'Undersampling consiste à éliminer des instances de la classe majoritaire pour atteindre un ratio souhaité appelé *Sampling strategy*. À titre d'exemple, si on a 10 instances de fraude et 1000 instances légitimes, avec un *Sampling strategy* de 0.5 on va éliminer 980 instances légitimes (pour avoir 20 instances légitimes finalement) tout en gardant les 10 instances de fraude.
Avec un ratio plus élevé, on donne plus d'importance à la classe minoritaire, mais on risque de perdre des informations importantes qui concerne la classe majoritaire. On a utilisé 2 techniques d'Undrsampling : **Random Undersampling** (documentation) et **Near Miss** (documentation).

3.3. Exemple : effet de sampling pour XGBoost

On observe dans cette partie l'effet d'*Oversampling* et d'*Undersampling* sur *XGBoost*. On a fait les mêmes expériences sur les autres modèles, et on avait des résultats semblables.
On observe toujours la même allure des courbes : lorsqu'on augmente la *sampling strategy*, le *recall* augmente et la *précision* et le *f1 score* diminuent.
Pour une *sampling strategy* de 10% (0.1) : le f1 score de sampling est généralement proche du f1 score sans sampling : le premier peut être plus important (le cas de *Decision Tree*) ou moins important (la majorité des modèles).

Pour conclure sur l'utilité de ces techniques :
— Les techniques de sampling nous permettent parfois d'améliorer le *f1 score*.

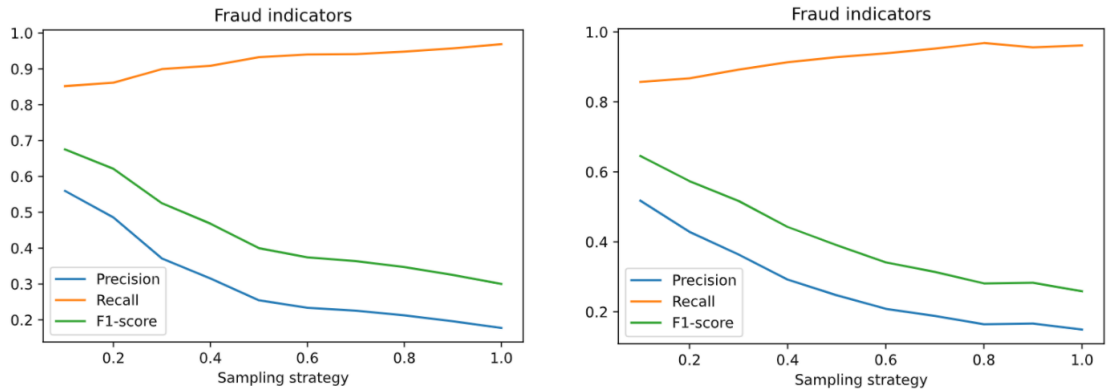

FIGURE 12 – Random Oversampling- Random Undersampling

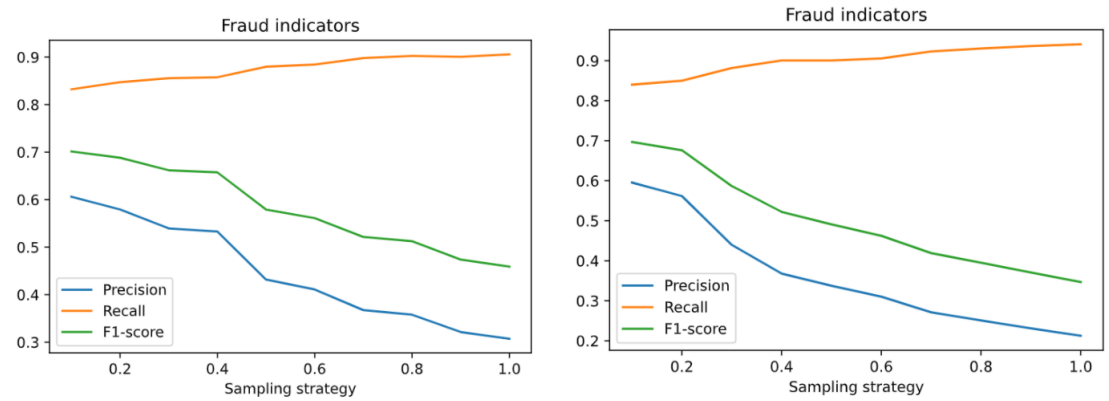

FIGURE 13 – SMOTE-ADASYN

— Les techniques de sampling nous permettent d'obtenir un *recall* plus important pour la classe 1. On peut exploiter ça dans les cas suivants :
  — Si on définit un seuil pour la *précision*, et puis notre but devient de maximiser le *recall* (au lieu de *f1 score*) : on peut varier la *sampling strategy* pour avoir une *précision* acceptable tout en maximisant le *recall*.
  — Créer plusieurs modèles avec plusieurs *sampling strategy* différentes : on aura des *recalls* différents et par conséquent des modèles différents et ça peut être utile si on applique le **bagging** [5].

4. Résultats

On présente dans ce paragraphe les performances de nos modèles. Pour cela, on a appliqué les 5 techniques de *sampling*, définis dans ce chapitre en 3.1. et 3.2., sur nos

5. On discute le bagging davantage dans 3.3.

modèles (sauf *LSTM*). Dans la figure 14, on observe les 10 meilleurs résultats, et dans la figure 15 [6] on observe les meilleurs résultats pour chaque modèle, tout en indiquant la méthode de *sampling* utilisée à chaque fois.

	No sampling	Over	Under	SMOTE	ADASYN	NearMiss	f1_score_1	recall_1	precision_1
LSTM	*						0.75	0.68	0.84
XGBoost	*						0.72	0.66	0.78
Random Forest				*			0.69	0.71	0.67
Random Forest	*						0.69	0.62	0.78
Random Forest					*		0.68	0.69	0.67
Random Forest		*					0.68	0.64	0.73
Neural Network	*						0.66	0.6	0.75
XGBoost				*			0.64	0.81	0.53
XGBoost					*		0.64	0.82	0.52
Decision Tree				*			0.61	0.82	0.48

FIGURE 14 – 10 meilleurs résultats

	No sampling	Over	Under	SMOTE	ADASYN	NearMiss	f1_score_1	recall_1	precision_1
LSTM	*						0.75	0.68	0.84
XGBoost	*						0.72	0.66	0.67
Random Forest				*			0.69	0.71	0.67
Neural Network	*						0.66	0.6	0.75
Decision Tree				*			0.61	0.82	0.48
KNN	*						0.44	0.41	0.47
Logistic Regression	*						0.08	0.65	0.04

FIGURE 15 – Meilleurs résultats pour chaque modèle

Conclusion : Dans cette partie, on a observé l'effet de différentes représentations, paramètres et méthodes de *Sampling*. Cela nous a donné un **Benchmark** initial, et par conséquent une vision globale sur les performances de chaque algorithme. Dans la partie suivante, on va optimiser les 4 meilleurs algorithmes : **LSTM**, **XGBoost**, **Random Forest** et **Neural Network**.

6. Le développement de Logistic Regression et Decision Tree a été fait par un collègue : ils subissent des traitements différents des autres algorithmes, pour cela on ne les évoque pas dans les paragraphes Modèles supervisés et Techniques utilisées.

Chapitre V.

Optimisation

Dans ce chapitre, on va se focaliser sur 4 algorithmes : : **LSTM**, **XGBoost**, **Random Forest** et **Neural Network**. Les traitements discutés ici concernent à la fois l'optimisation des performances (*f1 score*) et aussi le temps de réponse.

1. Optimisation des performances

Dans une première partie, on a optimisé les performances des algorithmes mentionnés auparavant en ajoutant des attributs qui n'existaient pas avant : *Feature Engineering*. Dans les paragraphes qui suivent, on va expliquer les attributs ajoutés, et on va montrer les résultats obtenus par ce traitement.

Il est important de souligner le fait que notre problème peut être considéré comme *Time Series* : puisque chaque transaction est effectuée à un temps bien défini : *unix_time*. Par conséquent, lorsqu'on ajoute un attribut pour la transaction effectuée à l'instant t, on a accès qu'à cette transaction et les transactions effectuées à t' avec $t' < t$: le non-respect de cette contrainte résulte dans un *Data Leakage* [1]. Les transactions dans nos *datasets* sont triées dans l'ordre chronologique, et on respecte toujours cette dernière contrainte.

Attributs contextuels : le point fort du *LSTM* est la présence du contexte : il regarde les variations dans le comportement de chaque utilisateur. Pour les autres algorithmes utilisés, les informations utilisées sont globales : ils regardent une transaction et produisent une prédiction sans rendre compte à l'utilisateur spécifique [2].

Une piste d'amélioration consistait donc de générer des informations liées à l'historique d'achat de chaque utilisateur.

1. C'est l'utilisation des informations qu'on a pas droit à obtenir dans le temps de prédiction
2. En théorie, un réseau de neurones peut simuler n'importe quelle fonction. Cependant, avec le nombre énorme d'utilisateurs et sans les colonnes d'identifications, c'est très peu probable qu'il puisse arriver à caractériser le comportement de chaque utilisateur

1.1. Dernier achat

Dans cette partie, on a utilisé 4 fonctions qu'on a implémentées pour générer 12 attributs supplémentaires. En premier lieu, on va présenter ces fonctions et puis on va analyser l'intérêt potentiel de ces attributs.

1.1 1 Génération des attributs

La première fonction porte la signature suivante :

```
def time_for_last_purchase(df, criteria=None,
time_col='unix_time', id_col='cc_num', inplace=False)
```

Elle est caractérisée par :

- **Output** : elle génère la différence en temps (*unix_time*) entre chaque transaction *tr* et une transaction *tr'* dans l'historique de l'utilisateur concerné. Le choix de *tr'* dépend du paramètre **criteria**.
- **df** : la base de données utilisée.
- **criteria** : si sa valeur est *None*, *tr'* est alors la dernière transaction avant *tr*. **Criteria** peut prendre soit la valeur *None* soit le nom d'une colonne de df. Si **Criteria**=*category* par exemple, *tr'* est la dernière transaction telle que $tr'(category) = tr(category)$.
- **time_col** : le nom de la colonne qui indique le temps.
- **id_col** : le nom de la colonne qui caractérise l'utilisateur (ou potentiellement un groupe d'utilisateur qui ont la même valeur de cet attribut).
- **inplace** : pour indiquer si les attributs sont ajoutés à **df** ou retourné sans affecter **df**.

La deuxième fonction est **delta_amt_purchase**, elle génère la différence du montant d'une entre chaque transaction *tr* et une transaction *tr'* dans l'historique de l'utilisateur concerné. Le choix de *tr'* dépend aussi du paramètre **criteria**.

La troisième fonction est **delta_dst_purchase**, elle génère de la même façon que les deux dernières fonctions la différence en distance (en utilisant *merch_lat* et *merch_long*).

La dernière fonction **delta_dst_amt** divise l'output de **delta_dst_purchase** par **delta_amt_purchase**.

1.1 2 Analyse des attributs générés

On utilise ces 4 fonctions pour générer 12 attributs, en donnant 3 valeurs pour le paramètre **criteria** : *None*, *category* et *merchant*.

Criteria = None :

— À partir de la matrice de corrélation présentée dans la figure 16 : on observe une corrélation non nulle pour les attributs *delta_amt* et *delta_time* avec *is_fraud*. On décide alors de garder ces deux attributs pour la phase d'optimisation.

— À partir des densités présentées dans la figure 17, on remarque une faible différence entre la distribution des fraudes et non fraudes pour les attributs *delta_dst* et *delta_dst_amt*. La corrélation avec *is_fraud* est faible (<0.01), on ne voit pas un grand intérêt alors de garder ces deux attributs pour la phase d'optimisation.

FIGURE 16 – Criteria=None : corrélation des attributs ajoutés avec is_fraud

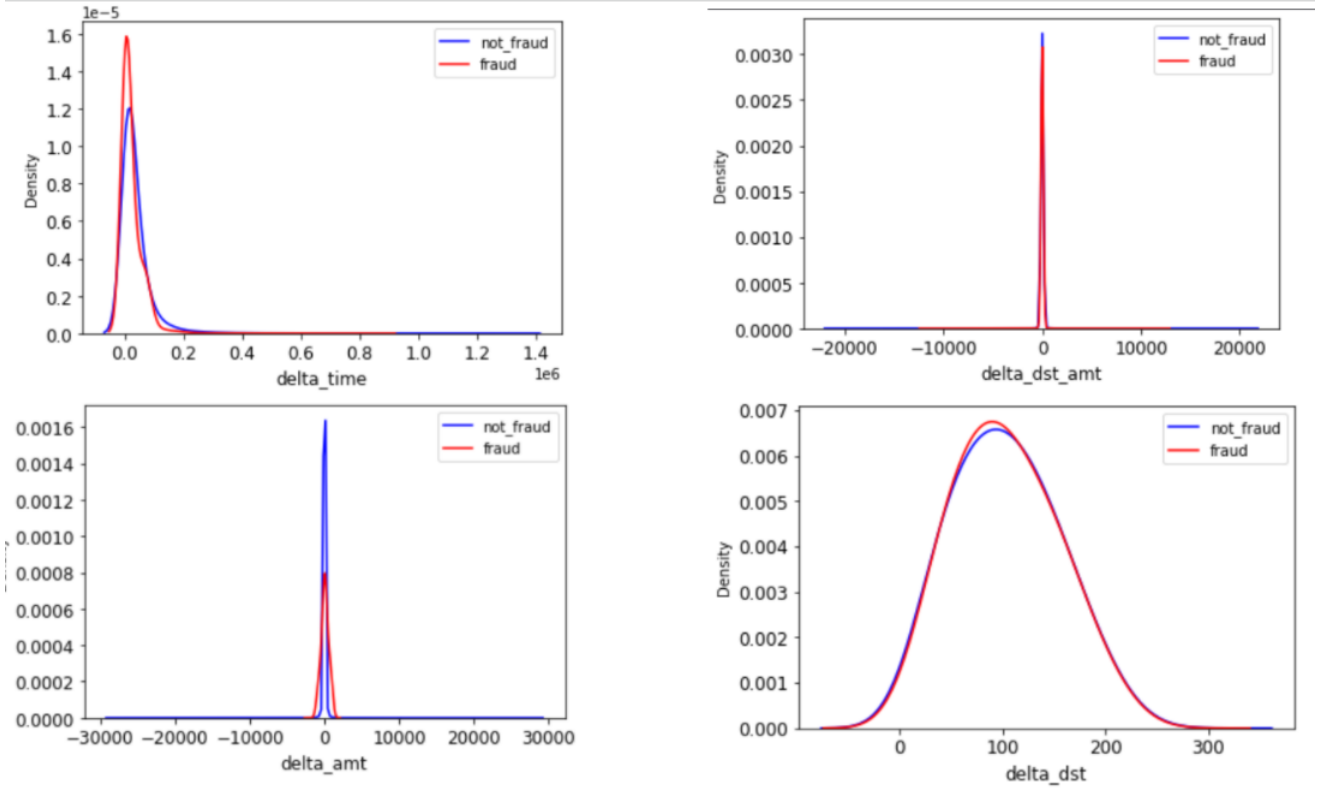

FIGURE 17 – Criteria=None : densités des attributs ajoutés

Criteria = category :

— A partir de la matrice de corrélation présentée dans la figure 18 : on observe une corrélation non nulle entre *delta_amt_category* et *is_fraud*. On décide alors de garder cet attribut pour la phase d'optimisation.

— À partir des densités affichées dans la figure 19, on décide de ne pas garder *delta_dst_category* (et aussi *delta_dst_amt_category*). En revanche, on décide de garder *delta_time_cateogry* à cause de la petite différence entre les deux distributions de fraude et de non fraude.

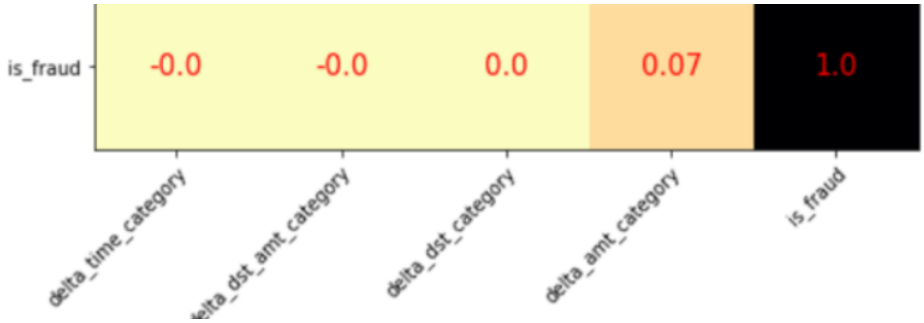

FIGURE 18 – Criteria=None : corrélation avec is_fraud

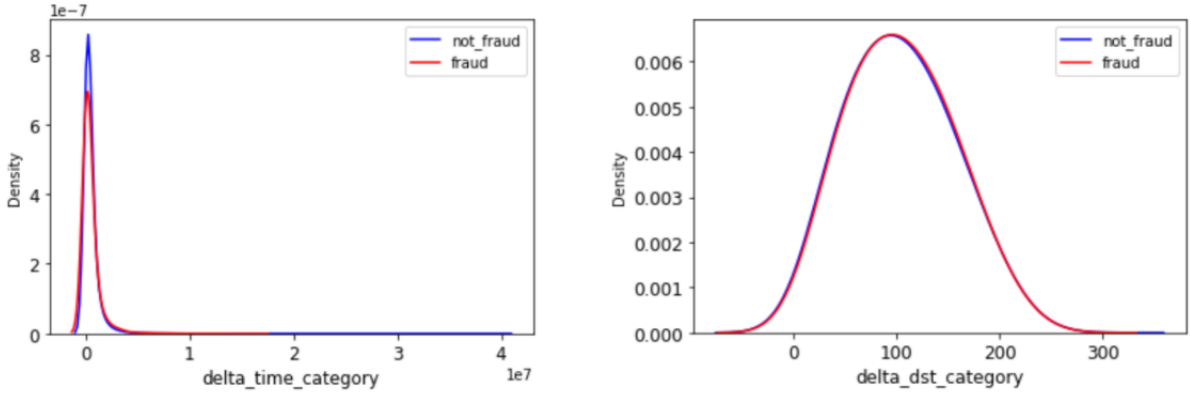

FIGURE 19 – Criteria=category : densités de delta_time_category et delta_dst_category

Criteria = merchant :

— On fait la même approche discutée dans **criteria=None**.

— On élimine *delta_dst_merchant* et *delta_dst_amt_merchant*. On garde *delta_amt_merchant* et *delta_time_merchant* pour la phase d'optimisation.

1.2. Moyenne des achats

1.2 1 Génération des attributs

La première fonction porte la signature suivante :

```
def avg_amt_purchase(df, window=60,criteria=None,
amt_col='amt', id_col='cc_num', inplace=False)
```

Elle est caractérisée par :

— **Output** : elle génère pour chaque transaction *tr*, la moyenne des dépenses des transactions *trs*. La valeur de *trs* dépend à la fois de **window** et **criteria**.

— **window** : nombre de transactions passées considérées (la transaction actuelle est incluse). S'il n'y a pas assez de transactions dans l'historique, le résultat sera *None*.

— **criteria**, **amt_col**, **id_col** et **inplace** ont le même comportement que dans *delta_amt_purchase*.

La deuxième fonction *delta_avg_amt_purchase* génère la différence du montant pour chaque transaction *tr*, avec la moyenne calculée avec *avg_amt_amt_purchase*.

1.2 2 Analyse des attributs générés

À l'aide des deux fonctions expliquées dans le paragraphe précédent, on a créé 16 nouveaux attributs :

— *avg_amt* (et *delta_avg_amt*) : on définit un **window** de 30 transactions, **id_col** = *cc_num* et **criteria** = *None*. On génère alors pour chaque transaction la moyenne des dépenses des 30 dernières transactions de cette personne (et la différence en montant de la transaction actuelle avec cette moyenne).

— *avg_amt_category* (et *delta_avg_amt_category*) : on définit un **window** de 4 transactions, **id_col** = *cc_num* et **criteria** = *cateogry*. On génère alors pour chaque transaction la moyenne des dépenses des 4 dernières transactions de cette personne pour la même *category* d'achat (et la différence en montant de la transaction actuelle avec cette moyenne).
On génère *avg_amt_merchant* (et *delta_avg_amt_merchant*) de la même façon (avec **criteria** = *merchant*)

— *avg_amt_state* (et *delta_avg_amt_state*) : on définit un **window** de 100 transactions, **id_col** = *state* et **criteria** = *None*. On génère alors pour chaque transaction la moyenne des dépenses des 100 dernières transactions pour toutes les personnes habitants dans le même *state* (et la différence en montant de la transaction actuelle avec cette moyenne).
On génère *avg_amt_city* (et *delta_avg_amt_city*) et *avg_amt_street* (et *delta_avg_amt_street*)

de la même façon (avec **id_col** = *city* et *street* respectivement)

— *avg_amt_job* (et *delta_avg_amt_job*) : on définit un **window** de 100 transactions, **id_col** = *job* et **criteria** = *None*. On génère alors pour chaque transaction la moyenne des dépenses des 100 dernières transactions pour toutes les personnes ayant dans le même *job* (et la différence en montant de la transaction actuelle avec cette moyenne).

— *avg_amt_job_category* (et *delta_avg_amt_job_category*) : on définit un **window** de 100 transactions, **id_col** = *job* et **criteria** = *category*. On génère alors pour chaque transaction la moyenne des dépenses des 100 dernières transactions pour toutes les personnes ayant dans le même *job* pour la même *category* (et la différence en montant de la transaction actuelle avec cette moyenne).

Tous ces attributs sont liés à l'attribut *amt* et d'une manière plus importante les attributs *delta_**. On visualise la corrélation de ces attributs avec *amt* et *is_fraud* dans la figure 20.

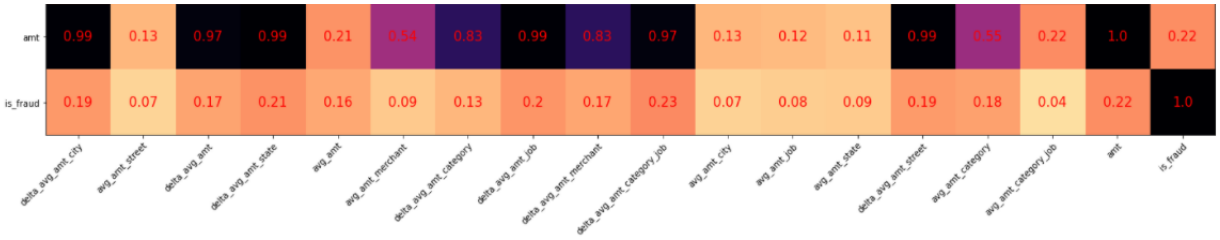

FIGURE 20 – Corrélation des attributs avec amt et is_fraud

On déduit à partir de la figure 20 le suivant :

— On élimine les attributs *delta_avg_amt_street*, *delta_avg_amt_job*, *delta_avg_amt_state*, *delta_avg_amt* et *delta_avg_amt_city* à cause de la trop forte corrélation avec *amt* (plus de 0.97).
— On garde *delta_avg_amt_category_job* malgré la forte corrélation avec *amt* car il a la plus forte corrélation avec *is_fraud* (0.23). En revanche, on élimine *avg_amt_category_job* car il a une faible corrélation avec *is_fraud* (0.04).

Finalement, on affiche les valeurs manquantes des 10 attributs qui restent dans la figure 21.

	percent_missing
avg_amt_merchant	77.772919
delta_avg_amt_merchant	77.772919
delta_avg_amt_category_job	43.765091
avg_amt_street	6.989647
avg_amt_city	6.427208
avg_amt_job	3.642547
avg_amt_category	2.992192
delta_avg_amt_category	2.992192
avg_amt	2.087879
avg_amt_state	0.382440

FIGURE 21 – Valeurs manquantes

On constate qu'on a beaucoup de valeurs manquantes pour *avg_amt_merchant* et *delta_avg_amt_merchant*. On affiche davantage leurs densités dans la figure 22.

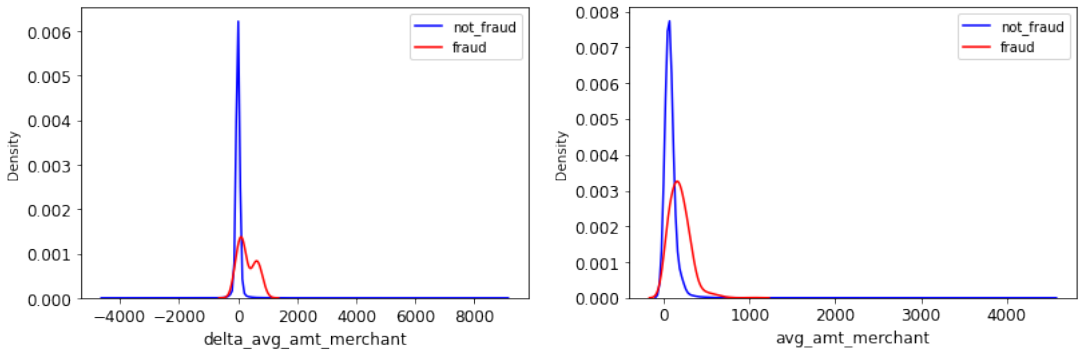

FIGURE 22 – Densités des attributs : delta_avg_amt_merchant et avg_amt_merchant

On observe la différence entre la distribution des données de fraude et de non fraude pour l'attribut *delta_avg_amt_merchant*, on décide alors de le garder. Pour *avg_amt_merchant*, il y a une différence, mais pas assez remarquable, on décide alors de l'éliminer.

1.3. Heure et jour

Les deux derniers attributs étaient proposés et implémentés par un collègue, ces attributs ne dépendent pas de l'historique de l'utilisateur : on fait l'extraction de l'heure (entre 0 et 23) et le jour de la semaine (entre 0 et 6) d'une transaction à partir de *trans_date_trans_time*. Dans figure 23 on observe les densités de ces attributs.

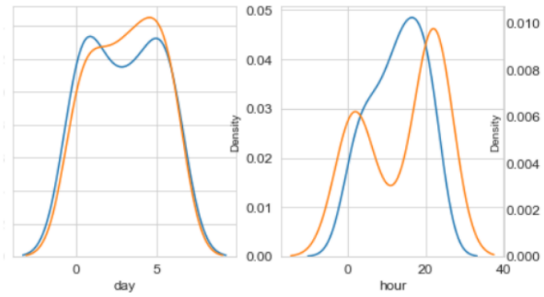

FIGURE 23 – Densités de attributs : day et hour

On remarque la différence entre la distribution de chacun de ces attributs, dans le cas de fraude et de non fraude. Cela montre l'intérêt de l'utilisation de ces deux attributs.

1.4. Résultats

Pour bien valider l'utilité de ces attributs, on a ajouté chaque ensemble d'attributs sur 3 étapes (dernier achat, moyenne des achats et jour/heure). On a utilisé le *Validation set* pour juger l'utilité de chaque ensemble d'attributs. On arrive aux conclusions suivantes :

— **XGBoost** : le *f1 score* s'améliore de 0.09 en ajoutant les 6 attributs liés au dernier achat. On gagne 0.04 de plus en ajoutant jour et heure. On gagne que 0.01 en ajoutant les attributs qui concernent la moyenne des achats. On a décidé alors de garder que les deux premières optimisations [3]. Finalement, le *f1 score* sur le *Test set* passe de **0.72** à **0.86**.

— **Random Forest** : pour ce modèle, on fait le même processus. On fait aussi une comparaison entre la **config_1_1** et **config_1_2**, et on a observé l'effet de **SMOTE** avec une *sampling strategy* de 0.1. En finale, on poursuit avec la **config_1_2**, **SMOTE** et l'ajout des 3 ensembles des attributs. Le *f1 score* passe sur le *Test set* de **0.69** à **0.88**.

— **Neural Network** : on fait le même processus effectué sur **XGBoost**. On finit par ajouter les 3 ensembles d'attributs. Le *f1 score* passe sur le *Test set* de **0.66** à **0.86**.

— **LSTM** : ce modèle passe de deux phases d'optimisation. Dans la première, on observe l'effet de **config_1_1** et **config_1_2**, la longueur des séquences et le nombre d'unités cachés. Dans la deuxième phase, on ajoute les ensembles d'attributs une par une, on finit par garder les indicateurs du dernier achat. Le *f1 score*

3. Les attributs qui sont liés aux moyennes des achats sont les plus compliqués à générer (en termes de temps d'exécution), on a décidé de les éliminer pour réduire ce coût temporel

passe sur le *Test set* de **0.75** à **0.83**.

Ces résultats montrent l'intérêt de l'utilisation d'une partie de ces attributs, on montre davantage l'importance des attributs ou *Feature Importance* pour **XGBoost** avec **SHAP** (documentation) dans la figure 24.

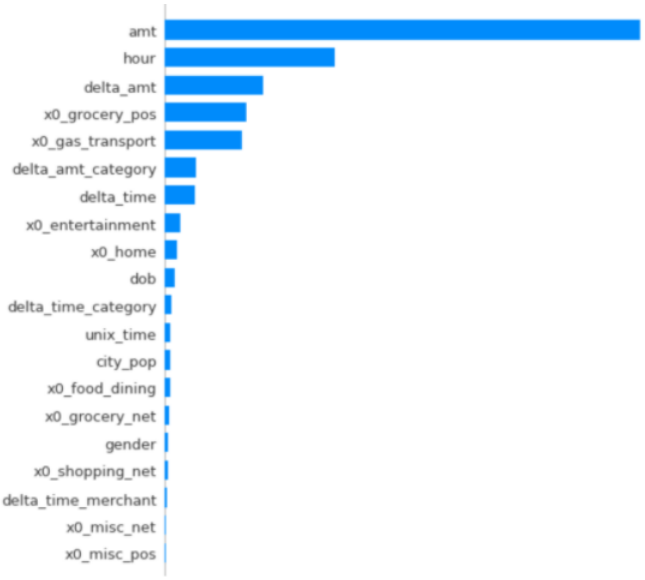

FIGURE 24 – Feature importance : XGBoost

Ce graphe montre l'importance des attributs ajoutés dans la prédiction : notamment *hour* et *delta_amt* étaient classés deuxième et troisième en importance.

Bien évidemment, on est arrivé à optimiser les performances de nos modèles d'une façon remarquable (démontré par l'amélioration de *f1 score* sur le *Test set*), mais est-ce qu'on a bien poussé nos modèles aux limites ? On discutera ce point dans la section 3.3. : *pistes d'améliorations*.

2. Mesure de temps de réponse

Dans cette partie, on a fait des expériences pour mesurer le temps moyen nécessaire pour faire une seule prédiction. On effectue une simulation : on passe les transactions une par une, on applique les étapes nécessaires de *Preprocessing*, *Feature Engineering* et prédiction sur chaque instance seule. Ceci est un premier prototype pour montrer la rapidité de chaque modèle. Il est important de noter que toutes les expériences sont faites sur la même machine : de 16Go de RAM, processeur I5 de 10ème génération (de 4 core) et sans l'utilisation d'une carte graphique.

Le but de cette partie est de faire une étude comparative entre les 3 modèles : **XGBoost**, **Random Forest** et **Neural Network**. On divise aussi le processus de prédiction en 4 étapes, pour avoir une idée globale sur les sources éventuelles de blocage ou ralentissement de la prédiction.

En dernier lieu, on ne fait pas ces expériences sur **LSTM** pour deux raisons : l'architecture du **LSTM** rend la simulation plus compliquée, et on ne voit pas un grand intérêt de faire face à cette complexité puisque ce modèle était le moins performant entre les versions optimisées.

2.1. XGBoost

Dans ce paragraphe, on va présenter deux versions utilisées pour simuler le temps nécessaire pour prédire la classe d'une transaction. On divise le temps d'exécution en 4 parties :

- **Preparation** : temps moyen nécessaire pour trouver les *transactions* nécessaires pour la génération des attributs supplémentaires.
- **Feature Engineering** : temps moyen nécessaire pour générer les nouvelles colonnes, ayant les transactions nécessaires (de l'historique) prêts.
- **Preprocessing** : temps moyen nécessaire pour appliquer les *Encodeurs* : *OneHotEncoder*, *LabelEncoder* et *StandardScaler*.
- **Prediction** : temps moyen nécessaire pour prédire la classe d'une instance : ayant appliqué toutes les transformations nécessaires.

2.1 1 Version 1

Pour cette version, la préparation des données se fait d'une manière simple : pour chaque transaction *tr* d'indice **n**, on parcourt toutes les transactions d'indices **i** avec $i < n$. On trouve les résultats suivants pour 1000 instances :

- **Preparation** : 251.30ms qui représentent 91.2% du temps total.
- **Feature Engineering** : 9.83ms qui représentent 3.6% du temps total.
- **Preprocessing** : 11.10ms qui représentent 4% du temps total.
- **Prediction** : 3.31ms qui représente 1.2% du temps total.
- **Total** : 275.55ms

On remarque que le temps pris dans la préparation est le plus important. Dans la version suivante, on se focalise sur l'optimisation de cette partie.

2.1 2 Version 2

Dans cette version, on initialise un dictionnaire (*map*) vide. Pour chaque transaction qui passe, on fait les traitements suivants :

- Si c'est la première fois qu'on rencontre ce *cc_num* : on crée un dictionnaire spécifique pour ce client, ou on met cette transaction comme dernière transaction,

on initialise la dèrniere transaction pour chaque *category* et *merchant* comme vide (sauf la catégorie et le commerçant de cette transaction).
— À chaque étape, on trouve les dernières transactions à partir de ce dictionnaire sans parcourir toutes les transactions.
— On met à jour l'historique à chaque fois, pour que l'historique de chaque client ne contienne que les informations nécessaires.

On refait la simulation avec 1000 itérations : le temps de préparation passe de **251.30ms** à **0.015ms**. Pour valider nos résultats sur un ensemble plus grand de données, on utilise le *Validation set* qui contient 259 335 transactions (en utilisant le *Training set* comme historique initial). On trouve les résultats suivants :
— **Preparation** : 0.02ms qui représente 0.1% du temps total.
— **Feature Engineering** : 9.76ms qui représentent 33.9% du temps total.
— **Preprocessing** : 15.17ms qui représentent 52.7% du temps total.
— **Prediction** : 3.83ms qui représentent 13.3% du temps total.
— **Total** : 28.78ms

2.2. Random Forest

Dans le cas de ce modèle, on utilise les mêmes attributs que dans *XGBoost*, et d'autres attributs qui indiquent la moyenne des achats. Par conséquent, on ne va pas obtenir les mêmes temps de préparation, génération des attributs et de preprocessing. En plus, on ne va pas obtenir le même temps de prédiction, car on a changé le modèle. Donc, on doit refaire toutes les étapes. On a refait la simulation (avec des changements pour la préparation des données) et on a eu les résultats suivants (avec 1000 instances du *Validation set*) :
— **Preparation** : 0.056ms qui représente 0.04% du temps total.
— **Feature Engineering** : 10.7ms qui représentent 7.5% du temps total.
— **Preprocessing** : 14ms qui représentent 9.8% du temps total.
— **Prediction** : 118.13ms qui représente 82.6% du temps total.
— **Total** : 143ms

On remarque que le temps de prédiction est le plus important. On a refait ces expériences sur toutes les transactions dans *Validation set*, on a mesuré le temps nécessaire pour toutes les étapes (autre que la prédiction) et on a trouvé des résultats similaires. On n'a pas refait les expériences pour le temps de prédiction, car ce temps n'est pas dépendant de l'historique considéré (contrairement au temps de préparation).

On peut conclure que pour réduire ce temps d'exécution, il est important de se focaliser sur le temps de prédiction : ce temps élevé est dû à l'utilisation de 140 arbres de décision. Une solution consiste à réduire la complexité du modèle.
Une deuxième solution consiste à paralléliser le processus de la prédiction : c'est faisable, car les arbres de décision travaillent d'une manière indépendante.

2.3. Neural Network

Ce dernier modèle partage avec *Random Forest* le même processus de préparation et de génération des attributs. Le processus de preprocessing diffère légèrement [4]. On décide alors de mesurer le temps de prédiction : on trouve un temps moyen de 25.92ms (en faisant l'expérience sur le *Validation set* tout entier). Cependant, ce temps peut être optimisé en utilisant des cartes graphiques.

Pour récapituler cette partie :
— C'est important de trouver une façon optimisée pour préparer les données.
— *XGBoost* était le plus rapide, grâce à sa rapidité dans la prédiction.
— *Random Forest* et *Neural Network* : ces deux modèles étaient plus lents, mais on peut les optimiser davantage.

3. Conclusion

3.1. Benchmark des performances

Après appliquer les optimisations mentionnées dans la section 1., on a mis à jour notre **Benchmark** : on le trouve dans la figure 25.

	No sampling	Over	Under	SMOTE	ADASYN	NearMiss	f1_score_1	recall_1	precision_1
Random Forest opt				*			0.88	0.86	0.89
Neural Network opt		*					0.86	0.83	0.88
XGBoost opt		*					0.86	0.78	0.96
LSTM opt		*					0.83	0.74	0.96
Decision Tree				*			0.61	0.82	0.48
KNN		*					0.44	0.41	0.47
Logistic Regression		*					0.08	0.65	0.04

FIGURE 25 – Benchmark : après optimisation

3.2. Déploiement des modèles

Les chiffres présentés dans les parties précédentes servent à une comparaison entre les modèles, car en réalité, le nombre réel de transactions est beaucoup plus important que dans notre base de données artificielle. En 2019 [22], les transactions par carte de crédit aux États-Unis ont dépassé 39 Milliards (contre à peu près 2 Millions sur 2 ans pour notre base de données). Cette différence dans l'échelle va avoir les conséquences suivantes :
— Une base de données plus grande : temps de préparation plus important pour chercher les transactions (perte de temps).

4. Les attributs *category* et *state* sont encodés avec le **OHE** au lieu du **LE**

— Des modèles plus complexes : temps de prédiction plus important (perte de temps).
— En réalité, on utilise des serveurs plus puissants et on parallélise les traitements (gain de temps).

Donc pour avoir des chiffres plus proches de la réalité, on doit trouver une base de données plus grande, ou augmenter le nombre de transactions dans notre base de données (par une technique de *Data Augmentation* qu'on définit, ou en utilisant le simulateur utilisé pour générer la base de donnée).

En plus de ça, le nombre de transactions aux États-Unis dépasse les 100 Millions par jour. Donc en réalité, on ne peut pas faire la prédiction pour une transaction puis on reçoit une autre, mais plutôt on doit traiter les transactions arrivant dans des temps très proches d'une manière simultanée. Il est donc important de concevoir un système capable de traiter plusieurs instances en même temps : d'où intervient une autre fois la parallélisation.

Finalement, on n'a pas précisé la borne supérieure imposée sur le temps de réponse pour être considéré comme *Real Time*. Selon [15], la réponse du système doit ne pas dépasser 200ms pour l'autorisation ou le blocage d'une transaction. Pour les réponses en temps réel, on doit avoir une précision très élevée. Une autre couche définit en [15] est le *Near Real Time* : dans cette couche, on favorise le *recall* (toujours avec un seuil minimal pour la précision), et le temps de réponse peut arriver à 1 minute. Dans cette couche, si une transaction est classée frauduleuse, elle est dirigée vers un expert pour vérifier et faire les démarches nécessaires. On peut juger alors un modèle comme adéquat pour une couche ou une autre (Real Time ou Near Real Time) après faire des expériences sur une base de donnée réelle.

3.3. Pistes d'améliorations

À ce stade-là, on a pensé à d'autres optimisations qu'on peut appliquer sur nos modèles. Cependant, on a jugé plus utile de passer à la partie *Théorie des Jeux*.
Ces optimisations concernent deux axes : le premier était le temps d'exécution, et en particulier le temps d'entraînement. En essayant d'optimiser les performances de nos modèles (*f1 score*), on doit toujours faire des expériences, ce qui peut prendre parfois beaucoup de temps. Donc une première étape consiste à paralléliser le processus de l'entraînement pour pouvoir faire plus d'expériences. Cela dit, on passe au deuxième axe : l'optimisation des performances.

On peut diviser cet axe en 4 :
— **Calibrage des paramètres** : on peut toujours calibrer nos paramètres après chaque optimisation en faisant un *Cross Validation*.

— **Feature Engineering** : les possibilités dans cette partie sont nombreuses. On a eu cependant quelques idées qu'on n'a pas essayées :
  — Trouver une façon pour mieux représenter *job* : en faisant un *Embedding* ou à la main : cette représentation peut dépendre du salaire, domaine ou les similarités dans les noms des *jobs*.
  — Varier le paramètre *window* dans la génération des attributs liés à la moyenne des achats.
  — Ajouter une colonne risque de fraude en utilisant *Kmeans*. On définit le risque à partir des pourcentages de fraude de chaque *cluster*. Pour le test, on associe chaque instance à un *cluster* (en utilisant la distance euclidienne) et cette instance va avoir le même risque que tous les points dans ce *cluster*. De cette façon, on n'a pas utilisé les *labels* du *Test set* et on évite le risque de *Data Leakage* car le *clustering* est fait uniquement avec les instances du *Training set*. Éventuellement, on peut utiliser aussi d'autres informations concernant chaque *cluster* : comme la taille et la moyenne des distances des points avec le centre.

— **Feature Selection** : on a implémenté trois méthodes pour la sélection des attributs. Ces méthodes sont : *Forward Feature Selection*, *Recursive Feature Elimination* et *Sequence Forward Floating Selection*. On a testé ces méthodes sur la base de données des transactions cryptées mentionnée en 1.. En utilisant ces techniques, an a eu des meilleures performances pour *Decision Tree*, mais pas pour *Random Forest* et *XGBoost*.

— **Bagging** : on a essayé d'agréger les résultats de nos 3 meilleurs modèles en faisant un vote. Les résultats sont améliorés de 0.01, ce qui n'est pas une optimisation remarquable. En contrepartie, en utilisant plusieurs modèles, on peut avoir des résultats stables et des modèles plus robustes. On peut générer ces modèles par plusieurs façons :
  — En utilisant des techniques de *sampling* avec des faibles *sampling strategy* (au dessous de 0.1).
  — En variant le *random seed* pour nos modèles.
  — En variant les paramètres de chaque modèle légèrement sans perturber le *f1 score*.

  On peut aussi faire varier le seuil de vote. Par exemple, un vote majoritaire, c'est voir si les votes de fraude dépassent un seuil de 50%, si c'est le cas alors c'est une fraude. On peut manipuler ce seuil pour favoriser un *recall* plus élevé (avec un seuil plus faible) ou pour avoir une *précision* plus importante (avec un seuil plus important).

Finalement, on a décidé d'implémenter des techniques de la Théorie des Jeux pour créer un modèle plus robuste contre les comportements dynamiques des fraudeurs. Dans le chapitre suivant, on va expliquer nos motivations pour l'utilisation de ces techniques de la Théorie des Jeux, l'approche utilisée, les expériences effectuées et les résultats.

Chapitre VI.

Théorie des jeux

1. Motivations

Lorsqu'on déploie un modèle d'apprentissage automatique, on fait l'hypothèse [14] que l'ensemble d'entraînement et les instances de la vie réelle suivent la même distribution.

Dans la réalité [14], l'environnement autour d'un problème change souvent, et on se trouve peu à peu avec des données d'une distribution qui diffère de celle qu'on a anticipée. Dans le cas de fraude à la carte crédit, le fraudeur s'adapte au système pour changer son comportement et échapper à la détection, ce qui cause une chute de performances. Pour cela, il est important de suivre les performances ou définir une stratégie de mise à jour des modèles de *Machine Learning*.

On ne va pas entrer dans les détails de ces techniques de mise à jour, notre but concerne plutôt la phase qui précède le déploiement : on veut créer un modèle robuste qui peut durer plus longtemps.
Dans notre cas, on est dans un environnement antagoniste ou un **Adversarial Environment**. On souhaite modéliser et anticiper le comportement du fraudeur pour couvrir le plus possible de vulnérabilités qu'il voudrait et pourrait exploiter avec ce qu'on appelle **Adversarial Learning** : on modélise un jeu compétitif entre le système et le fraudeur en définissant les informations, motivations et actions possibles pour chaque joueur.

Dans le paragraphe suivant, on donne un aperçu sur les types de jeux proposés dans la littérature, et on explique en détail l'approche qu'on a choisie.

2. État de l'art

2.1. Différentes approches

Les approches dans la littérature diffèrent sur plusieurs niveaux : le type de jeu proposé (simultané ou séquentiel), le type d'équilibre (qui dépend de la fonction d'utilité ainsi que le type du jeu) et les hypothèses sur l'adversaire.

Dans [14], les auteurs définissent un jeu séquentiel entre un Spammeur et le système de détection de Spam. À chaque mail, le Spammeur a le droit d'agir sur le mail si ce mail est Spam, il joue en premier et peut changer le contenu du mail pour camoufler ses intentions. Le Système joue en second : il a la possibilité de mettre à jour ses paramètres (sans savoir si ce mail est Spam ou pas). On appelle ce type de jeu un *Stackelberg Game*, les auteurs de [14] cherchent l'équilibre avec un algorithm génétique. Pour limiter les actions du *Spammeur*, il est pénalisé pour chaque changement qu'il fait, et il cherche à augmenter le *False Negative Rate*. Quant au Système, son utilité dépend du *True Negative Rate*, *True Positive Rate* et il est pénalisé pour chaque mise à jour de ses paramètres : c'est un jeu à 2 joueurs de somme non nulle.

L'approche de [19] consiste en premier temps à diviser une base de données de fraude bancaire sur 10 parties. Chaque partie est divisée en *Train, Validation* et *Test sets*. Les auteurs de [19] font la comparaison entre deux classifieurs. Le premier est entraîné sur le *Train set* de chaque partie i, et évalué sur le *Test set* de i. Le deuxième est entraîné sur les données d'entraînement augmentées : les éventuelles vulnérabilités détectées lors de la phase $i-1$ [1] sont passées à la *Train set* de la phase i (et augmentés avec SMOTE). Le deuxième Classifieur anticipe mieux les vulnérabilités qui peuvent être exploitées.

Il y a plusieurs approches qui traitent ce problème, on ne va entrer en détails, on propose [21] qui enquête ces méthodes d'une façon plus exhaustive.

2.2. Approche choisie

Parmi les différentes approches proposées dans la littérature, on a décidé de poursuivre avec les méthodes proposées par [26] [2].

Dans [26], les auteurs créent un *SVM* (Support Vector Machine) dans un *Adversarial Environment* : le Spam sur la messagerie électronique *e-mail*. Cet article fait la comparaison entre deux *SVMs* (SVM linéaire et One Class SVM) et des *AD-SVM* : ce sont des *SVMs* entraînés en anticipant certaines formes d'attaques.

Dans cet article, on trouve les éléments suivants :

1. à l'aide de la *Validation set*
2. Toutes les expressions mathématiques dans 2.2. sont pris de [26]

— Un ensemble de points $\{(x_i, y_i) \in (X, Y)\}_{i=1}^{n}$ avec x_i la $i^{ème}$ instance, y_i le label associé, $X \subset \mathbb{R}^d$ et $Y = \{-1, 1\}$.
— d est le nombre d'attributs, on note par x_i^j l'attribut j de l'instance x_i.
— Les instances de label 1 sont appelés les instances *malicieuses*, l'adversaire (ou le Spammeur) a le droit de déplacer ces instances. Le but du Spammer est d'échapper à la détection. On note δ_{ij} comme le déplacement (ou l'attaque) effectué sur l'attribut j de l'instance x_i [3].

L'article définit deux types d'attaques : *Free Range* et *Restrained*.

2.2 1 Mode Free Range

On suppose dans ce mode que le Spammeur à la connaissance des limites de chaque attribut. On note x^{max} (et resp. x^{min}) une liste de d valeurs : les plus grandes (resp. petites) valeurs qu'un attribut peut prendre. Pour éviter l'influence des anomalies, on peut prendre les quantiles 0.01 et 0.99 de chaque attribut pour définir x^{max} et x^{min}.

Le déplacement effectué par le Spammeur est limité par le coefficient $Cf \in [0, 1]$:

$$Cf(x_j^{min} - x_i^j) \leqslant \delta_{ij} \leqslant Cf(x_j^{max} - x_i^j)$$

Ce mode peut s'avérer plus général par rapport au mode *Restrained* qu'on va présenter dans la partie suivante, mais en contrepartie plus paranoïaque (il représente parfois des attaques plus sévères).

2.2 2 Mode Restrained

Pour ce mode, on suppose que le Spammer connaît l'emplacement d'un point $x^t \in X; y^t = -1$. On peut prendre x^t comme le centre des points légitimes (de label = -1). Le déplacement δ_{ij} dépend de deux coefficients C_δ et C_ξ :

$$0 \leqslant (x_j^t - x_i^j)\delta_{ij} \leqslant C_\xi \left(1 - C_\delta \frac{\mid x_j^t - x_i^j \mid}{\mid x_i^j \mid + \mid x_j^t \mid}\right)(x_j^t - x_i^j)^2$$

Dans ce mode de déplacement, le Spammeur peut rapprocher chaque point x_i vers x^t. C_ξ joue le même rôle que Cf dans *Free Range* pour limiter le déplacement, cependant C_ξ peut prendre des valeurs supérieures à 1. $C_\delta \in [0, 1]$, lorsqu'elle est plus grande, elle atténue plus le déplacement. Contrairement à C_ξ qui contrôle le déplacement d'une façon linéaire, C_δ agit d'une façon plus intense sur les points qui sont éloignés de x^t.

2.2 3 Informations et fonction d'utilité

On définit dans cette partie la fonction d'utilité du Spammeur, ainsi que le déroulement du jeu.

3. Effectuer un déplacement δ_{ij} est équivalent à : $x_i^j \leftarrow x_i^j + \delta_{ij}$

On considère un jeu entre le système de détection de fraude et le Spammeur. Le système de détection de fraude joue en premier et définit son *SVM*, le Spammer joue en second et effectue son attaque.

— Informations :
  — Système : emplacement initial de tous les points et les labels associés, ainsi que les déplacements autorisés pour le Spammeur (mode du jeu, coefficients associés et x^t ou x^{max} et x^{min})
  — Spammeur : emplacement des points malicieux et le *SVM* choisi par le système.
— Buts :
  — Système : minimiser le *hinge loss* du *SVM*.
  — Spammer : maximiser le loss du système, c'est la fonction d'utilité du Spammeur.

Sous les contraintes imposées par le mode du jeu choisi, le jeu consiste à résoudre le problème suivant :

$$\arg\min_{\omega,b}\left(\sum_{\{i|y_i=1\}}\max_{\delta_i}\lfloor 1-(\omega(x_i+\delta_i)+b)\rfloor_+ + \sum_{\{i|y_i=-1\}}\lfloor 1+(\omega.x_i+b)\rfloor_+ + \mu\|\omega\|^2\right)$$

Le système doit alors anticiper l'attaque le plus sévère pour chaque *SVM*, et choisir le *SVM* qui est le plus robuste contre les attaques qui sont permises sous les contraintes de déplacement imposées.

Pour valider cette approche, [26] ont fait des expériences sur deux bases de données fictives et deux bases de données de Spam. Les expériences consistaient à définir une sorte d'attaque (sur une partie de données qui n'est pas utilisé dans l'entraînement) [4]. Les auteurs de [26] ont effectués cette attaque avec des intensités différentes, et ont mesuré les performances (*accuracy*) avec plusieurs modèles (SVM, One Class SVM, AD-SVMs avec les deux modes Free Range et Restrained sous plusieurs coefficients) [5].

3. Objectifs

Dans les paragraphes précédents, on a parlé de l'intérêt général de l'utilisation de la Théorie des Jeux. Dans ce paragraphe, on va définir deux critères d'évaluation pour juger la contribution des techniques qu'on va tester. Pour cela, on divise notre base de données *fraudTrain.csv* en deux (*Training set* et *Validation set*) et on utilise notre base de données *fraudTest.csv* comme *Test set*.

4. Les auteurs de [26] divise chaque dataset en deux : training set et test set. 10% des données de test set sont choisies aléatoirement pour effectuer les expériences, ce processus est répété 10 fois puis la moyenne est prise.

5. On fait la même méthode d'évaluation avec fscore comme indicateur de performance dans 30.

3.1. Objectif 1 : robustesse contre les attaques

Cette méthode d'évaluation était évoquée dans [26]. Dans un premier temps, on crée plusieurs modèles, qui anticipent des attaques de sévérité et de types différents. Ces modèles sont comparés avec un **SVM normal** (entraîné sur la *Training set* sans modifications). Le critère de comparaison est le *f1 score*. On compare ces modèles et le **SVM normal** sur la *Validation set* d'origine, et aussi sur des versions modifiées de la *Validation set* (ces versions modifiées sont affectées par des attaques qu'on a modélisées, plus de détails dans 5.).

Pour **résumer** cet objectif : on va modéliser des attaques, et on cherche à trouver un modèle qui est à la fois performant en l'absence d'une attaque et robuste contre des attaques de sévérités différentes.

Cet objectif, si atteint, peut se refléter sur la détection des fraudes en pratique de deux façons :

— Le fait d'anticiper certaines attaques peut minimiser les dégâts qui sont liés à l'adaptabilité des fraudeurs.
— Le fait de couvrir plus de vulnérabilités qui peuvent être exploitées, peut produire un système qui dure plus longtemps.

3.2. Objectif 2 : stabilité

Lors du déploiement d'un modèle, on cherche de la stabilité. Par exemple, un modèle qui détecte 90% des fraudes tous les jours est préférable à un système qui détecte parfois 80% et parfois 100%. Avec le premier, on peut anticiper les dégâts éventuels et aussi on peut détecter la défaillance du système.

Pour les 3 modèles optimisés (**Neural Network**, **XGBoost** et **Random Forest**), on a affiché les performances mesurées sur chaque mois de la *Test set*. On présente les graphes des performances de **XGBoost** à titre d'exemple dans la figure 26. On observe dans la figure 26 la variation de la *précision*, *recall*, *fscore* et la proportion des fraudes à chaque mois. Ce dernier montre le déséquilibre entre les transactions frauduleuses et légitimes. On remarque que la *précision* et le *recall* ne varient pas de la même manière :

— La *précision* dépend à la fois des transactions légitimes (présentes dans FP : False Positives) et des transactions frauduleuses (présentes dans TP : True Positives). TP et FP peuvent être affectés si la proportion des fraudes a changé (par exemple entre le mois 0 et le mois 6).
— Le *recall* : ce terme comporte seulement FN et TP qui sont tous des fraudes, donc il ne dépend pas des transactions légitimes. Alors seul un changement de comportement des fraudeurs peut induire des changements dans la capacité du système de détecter les fraudes. Ce changement affecte aussi la précision, cependant un changement dans la proportion des fraudes sans changement dans les comportements des fraudeurs n'affecte pas le *recall*.

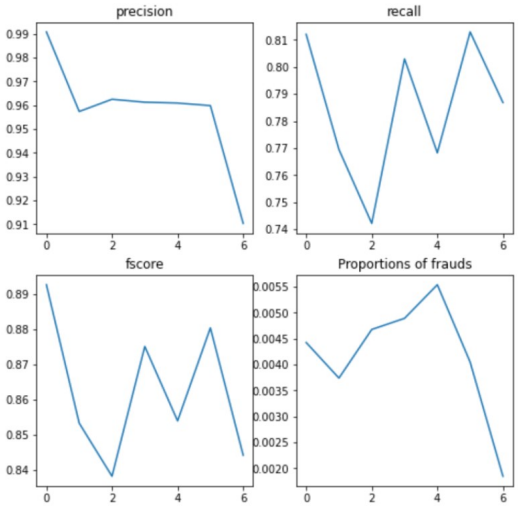

FIGURE 26 – XGBoost : variations des indicateurs sur Test set

On a défini donc l'objectif suivant : on cherche à détecter des fraudes qui sont différentes de ceux présents dans le *fraudTrain.csv* en anticipant le comportement adaptatif du fraudeur. On ne sait pas si le changement de la distribution des données frauduleuses (ou le comportement du fraudeur) dans le *Test set* est lié à un aspect adaptatif ou aléatoire. On suppose en premier lieu que c'est adaptatif, et on cherche à minimiser son effet en stabilisant le *recall*. Pour faire cela : on cherche à minimiser la variance du *recall* sur les 6 mois de la *Test set*.

4. Expériences : données fictives

Dans ce paragraphe, on va présenter les expériences menées sur des données fictives qu'on a créé dans les buts suivants :

— Les premières données créées sont inspirées de [26] : on utilise ces données pour valider notre compréhension [6] de l'article, et observer l'effet de la variation de plusieurs paramètres sur les résultats.

— On a créé d'autres données fictives, qui diffèrent en formes par rapport aux premières données, pour illustrer les difficultés qu'on peut rencontrer avec les données de fraudes. On a mené plusieurs expériences sur ces données, et on a observé les résultats de plusieurs approches. L'avantage de ces données et la simplicité de la représentation (données en 2D), et le temps d'exécution qui est moins important (par rapport aux données de fraude).

6. Ainsi des développements mathématiques et l'implémentation

4.1. Données séparables linéairement

Dans cette partie, on va traiter le mode *Free range* mentionné en 2.2. [7].

4.1 1 Expressions mathématiques

On pose x l'ensemble des instances d'entraînement (elle comporte n instances de dimension d) et y l'ensemble des labels associés (de valeurs dans{-1,1}). Soit x_i une instance de x, on note par x_i^j la valeur de l'attribut j pour le point x_i.

D'après [26], trouver le meilleur *SVM* sous l'hypothèse de présence d'une attaque de type *Free range* de coefficient C_f consiste à résoudre ce problème :

$$\arg\min_{\omega,b} \sum_i \max_{\delta_i} \lfloor 1 - y_i(\omega.x_i + b) - \frac{1}{2}(1+y_i)\omega.\delta_i \rfloor_+ + \mu\|\omega\|^2$$

$$\text{avec} \begin{cases} \delta_i \leqslant C_f(x^{max} - x_i) \\ \delta_i \geqslant C_f(x^{min} - x_i) \end{cases}$$

Dans [26], on trouve une deuxième expression équivalente à ce problème avec plusieurs contraintes, mais on ne trouve pas comment résoudre ce problème. On fait alors les calculs nécessaires pour transformer ce problème en un problème d'optimisation quadratique (en utilisant les conditions de **K.K.T**). En posant $\mu = \frac{1}{2}$, on arrive finalement à définir le problème dual suivant :

$$\min_\alpha \frac{1}{2}\alpha^T P\alpha + q^T\alpha$$

$$\text{avec} \begin{cases} G\alpha \leqslant h \\ A\alpha = b \end{cases}$$

Les expressions de ces variables, ainsi que les démonstrations mathématiques sont dans l'annexe 1..

Posons d=2, on résout le problème dual à l'aide du module **CVXOPT** dans Python (documentation). On trouve ainsi ω à partir de l'expression de α :

$$\omega = \sum_{i=1}^{n} \alpha_i.y_i.x_i + \frac{1}{2}\sum_{i=1}^{n}(1+y_i)(\alpha_{i+n}.x_i^0, \alpha_{i+2n}.x_i^1)$$

Le premier terme est le même qu'un **SVM normal**, le deuxième terme représente la réaction du *SVM* contre le déplacement (ou l'attaque) prévu. Le deuxième terme contient $(1+y_i)$, donc si $y_i = -1$ alors $(1+y_i) = 0$ et on anticipe l'absence de déplacement (car ces instances représentent les instances légitimes, et l'attaquer a le droit de déplacer les points x_i tels que $y_i = 1$).

Soit x^* un point tel que $\epsilon < \alpha^* < C - \epsilon$ et $\alpha^{*+n} = alpha^{*+2n} < \epsilon$; $\epsilon = 1^{-7}$:

$$b = y^* - w.x^*$$

7. Les mêmes expériences sont menées avec le mode *Restrained* en variant C_δ.

4.1 2 Visualisation

Dans ce paragraphe, on présente un exemple sur le comportement d'un *SVM* avec x un ensemble de points en 2 dimensions, qui comprend deux classes (-1 et 1). Les instances de x sont séparables linéairement, on implémente le problème d'optimisation quadratique définie en 4.1 1 et on montre parmi les résultats deux *SVMs* : le premier présent dans la figure 27 est un *SVM* appliqué sur les données d'origine ($Cf = 0$), le deuxième présent dans la figure 28 est un *SVM* appliqué sur les données x avec $Cf = 0.3$ (l'adversaire peut déplacer les points pour compliquer la tâche de classification).

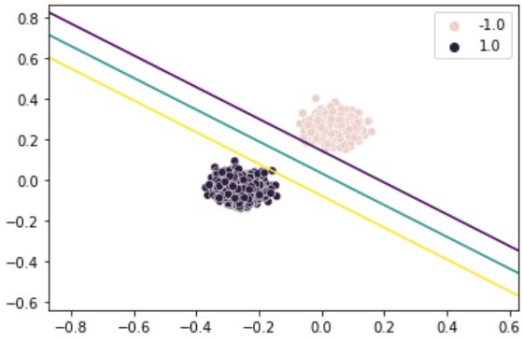

FIGURE 27 – SVM sur les données x : sans perturbation

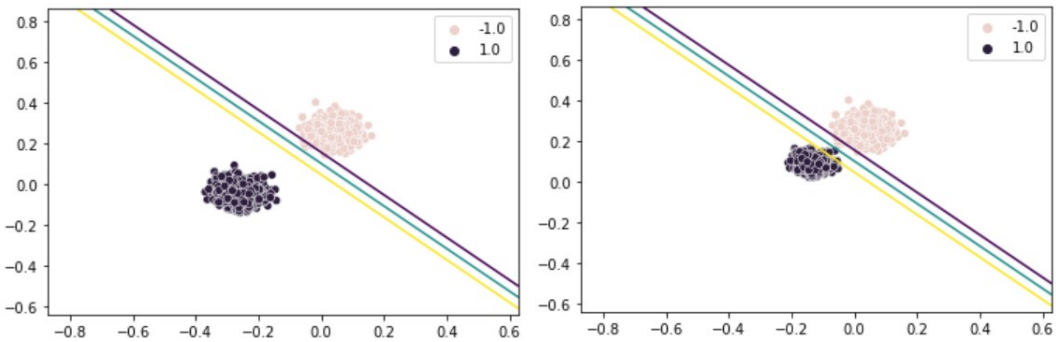

FIGURE 28 – SVM sur les données x : Cf=0.3

Dans la figure 28, on trouve à gauche la réponse du système et la représentation des données sans perturbation. À droite, on observe les déplacements que l'adversaire peut effectuer sur les data.
À partir les deux figures ci-dessus on observe deux choses :

— La marge est plus faible dans la figure 28 par rapport à la figure 27. En général, une marge plus large sert à définir un *SVM* plus robuste, donc le *SVM* présenté dans la figure 28 risque de mal généraliser en le testant sur d'autres données de distributions différentes.

— En observant le déplacement effectué par l'adversaire, on trouve que le *SVM* qui anticipe une attaque d'intensité $Cf = 0.3$ est plus adapté en cas d'attaque que le *SVM* présenté dans la figure 27. On conclut donc que le *SVM* de la figure 28 sera plus robuste contre des perturbations qui rassemblent l'attaque modélisée par le mode *Free Range* (avec $Cf = 0.3$).

En définissant un modèle d'attaque qui est indépendant de la perturbation anticipée par le mode *Free Range* (ou *Restrained*) et le plus proche de réalité, on peut comparer les performances des *SVMs* qui anticipent des déplacements avec les performances d'un *SVM normal*.

Dans cette partie, ainsi que dans le paragraphe 4.2., on adopte une attaque définie dans [26]. Dans la section 5., on définit une attaque qui rassemble à ce dernier. Des attaques plus compliquées sont proposés dans la section 6.3..

L'attaque mentionnée est effectuée sur une partie de données (*Validation set*) de même distribution que les données d'origine. Le déplacement effectué pour chaque point x_i tel que $y_i = 1$, comme mentionné dans [26], est le suivant :

$$x_{ij} = x_{ij} + \delta_{ij} \text{ avec } \delta_{ij} = f_{attack}(x_j^t - x_{ij})$$

f_{attack} est un réel entre 0 et 1, x^t est un point qui apporte une information sur la distribution des points légitimes : on prend x^t comme le centre des points légitimes ($y = -1$). Ce déplacement permet de rapprocher les points malicieux vers le centre des points légitimes. On montre dans la figure 29 une attaque (ou déplacement) effectué sur le *Validation set* avec une intensité $f_{attack} = 0.3$.

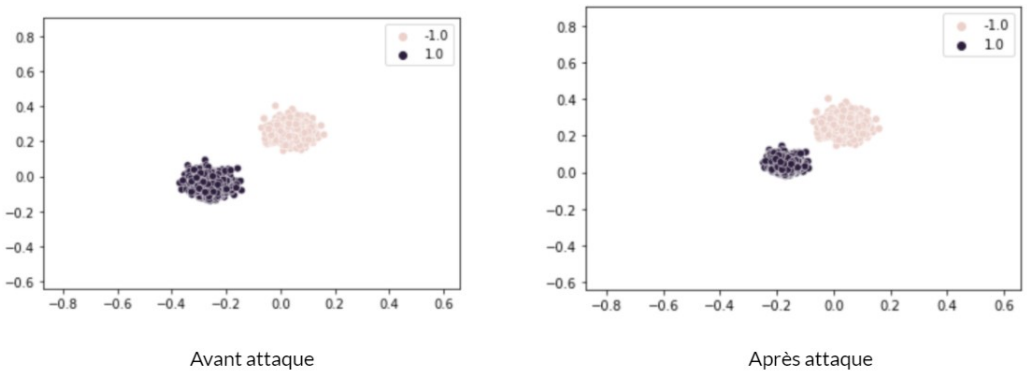

FIGURE 29 – Attaque sur Validation set : f=0.3

4.1 3 Résultats

On entraîne plusieurs *SVMs* : un *SVM* qui n'anticipe pas de déplacement et 5 *SVMs* qui anticipent des attaques de sévérités différentes [8]. On évalue ces modèles en mesurant l'*accuracy* sur le *Validation set* d'origine ($f_{attack} = 0$) et des versions modifiées de la *Validation set* (en effectuant des attaques d'intensités différentes). Les résultats sont illustrés dans la figure 30.

	f_attack=0	f_attack=0.3	f_attack=0.5	f_attack=0.7	f_attack=0.9
AD-svm Free Range, Cf = 0.1	1.00	1.00	0.922	0.500	0.50
AD-svm Free Range, Cf = 0.3	1.00	1.00	1.000	0.536	0.50
AD-svm Free Range, Cf = 0.5	0.99	0.99	0.990	0.990	0.49
AD-svm Free Range, Cf = 0.7	0.68	0.68	0.680	0.680	0.68
AD-svm Free Range, Cf = 0.9	0.50	0.50	0.500	0.500	0.50
SVM	1.00	1.00	0.808	0.500	0.50

FIGURE 30 – Tableau de comparaison : données séparables linéairement

D'après le tableau ci-dessus, on remarque le suivant :

— Si on anticipe une attaque, qui est similaire à celui représenté par le déplacement δ_{ij} et on ne pose pas une marge pour la valeur de f_{attack} : on trouve qu'un *SVM* avec $Cf = 0.3$ a une meilleure *accuracy* par rapport à un *SVM normal* pour des attaques qui varient entre 0 (pas de déplacement) et 0.9 (attaque sévère) [9].

— Si on suppose qu'on est susceptible de se trouver face à une attaque qui est modéré (f_{attack} entre 0 et 0.5), on trouve que la configuration $Cf = 0.3$ a les mêmes performances avec un *SVM normal* et fait mieux dans le cas d'une attaque d'intensité $f_{attack} = 0.5$.

— Si on suppose qu'on est susceptible de se trouver face à une attaque qui est sévère (f_{attack} entre 0.7 et 0.9), on trouve que les configurations $Cf = 0.5$ et $Cf = 0.7$ sont les plus adaptées.

4.2. Données non-séparables linéairement

Dans ce paragraphe, on montre d'une façon simplifiée la problématique liée à la classification des données qui ne sont pas séparables linéairement et notre approche pour faire face à ce problème.

Le problème quadratique présenté dans la partie 4.1 1 ne reste plus valable dans le cas des données qui ne sont pas séparables linéairement. Ceci est dû à l'introduction de la non-linéarité (kernel non linéaire pour SVM, en particulier **RBF**) dans la recherche de l'expression du SVM. En résumé, on ne travaille plus avec un problème convexe avec un optimum global qui est plus simple à atteindre, mais avec un problème non linéaire qui

8. On fait la même approche pour le mode *Restrained*

9. L'accuracy du SVM qui anticipe l'attaque est strictement supérieure à celle d'un SVM normal pour des attaques d'intensités f_{attack} 0.5 et 0.7, et la même pour les autres intensités.

contient des optimums locaux ou on risque de se bloquer lors de la recherche de notre optimum global. **À partir de là, les équations présentées dans [26] ne marchent plus, on ne peut plus travailler avec la méthode de résolution proposée par l'article. Pour cela, on développe une approche qu'on va détailler dans la partie suivante**.

4.2 1 Méthodes de résolution

Le problème d'optimisation qu'on cherche à résoudre en introduisant le kernel **RBF** ϕ [11] est le suivant :

$$\arg\min_{\omega,b} \sum_i \max_{\delta_i} [1 - y_i(\omega.\phi(x_i + \delta_i) + b)]_+$$

$$\text{avec } \begin{cases} \phi(x1).\phi(x2) = K(x1, x2) = e^{-\gamma\|x1-x2\|^2} \forall x1, x2 \in \mathbb{R}^d; d \in \mathbb{N}* \\ \delta_i = 0 \text{ si } y_i = -1 \end{cases}$$

On cherche alors à minimiser cette expression en trouvant le meilleur SVM (ω et b) qu'on appellera *loss* (et en prenant en compte toutes les valeurs de δ_i possibles). Quant à l'adversaire, il cherche à maximiser cette expression en trouvant le meilleur déplacement δ (sachant ω et b choisis). Pour résoudre ce problème, on a adopté l'approche suivante :

- On initialise $\delta = 0_{n\times d}$.
- On répète les étapes suivantes jusqu'à convergence ou pour un nombre maximum d'itérations :
  - On minimise l'expression du loss en variant ω et b : en pratique, on applique le dernier déplacement δ sur x pour générer x', ensuite on entraîne un SVM sur x' : pour la phase de minimisation, on peut atteindre un minimum global.
  - On maximise l'expression du loss en variant δ_i avec les dernières valeurs de ω et b. Pour cela, on a testé 3 méthodes, avec les 3 on risque toujours de tomber dans un maximum local : donc on a fait une comparaison entre ces trois méthodes. On va les présenter dans la partie 4.2 2. On fixe ainsi la valeur de δ pour l'itération suivante.
- À la fin, on choisit le SVM avec le plus petit *loss* après la phase de maximisation : c'est le SVM qui minimise l'effet le plus sévère (ou maximum) des attaques.

Ce processus est inspiré du *Gradient Descent Ascent* ou **GDA** utilisé pour résoudre des problèmes de type min-max [10]. Cependant, on n'utilise pas la descente de gradient dans la phase de minimisation, et on finit par utiliser une méthode autre que la montée de gradient pour la maximisation. Notre approche ne partage finalement avec le GDA que le fait de minimiser et maximiser à chaque itération.

Il est important de noter que ce processus itératif ne permet pas toujours de trouver l'optimum global, on valide ainsi l'utilité de cette approche à partir des résultats de l'évaluation.

10. Dans [17], les auteurs ont utilisé le GDA pour créer un GAN : Generative Adversarial Network

4.2 2 Méthodes de maximisation

Pour la phase de maximisation, on a défini 3 approches :

— Une approche simple : essayer 3 valeurs pour chaque attribut, 9 valeurs dans ce cas (3^d avec $d = 2$). Pour chaque attribut, on associe deux valeurs extrêmes (déplacement maximal vers x^{min} ou x^{max}) ou la valeur nulle. On choisit parmi ces 9 valeurs le déplacement qui ajoute le plus dans l'expression du *loss*.

— Descente de Gradient : on cherche à maximiser le terme $-y_i.\omega.\phi(x_i + \delta_i)$. L'adversaire peut varier δ_i que lorsque $y_i = 1$. Le problème est équivalent à minimiser $\omega.\phi(x_i + \delta_i)$. Ce terme, ainsi que son dérivé par rapport à δ_i, peuvent être calculés en dimension finie. Les détails de calcul sont dans l'annexe 2. .

— Algorithme génétique : on cherche à minimiser la même expression définie pour la descente de gradient. Pour cela, on génère une population de δ_i pour chaque instance x_i tel que $y_i = 1$. À l'aide d'un algorithme génétique, on choisit le meilleur descendant pour un nombre défini de générations.

Ces méthodes de maximisation ne garantissent pas de trouver un optimum global, cela augmente le risque de trouver un optimum local lors de l'application de notre méthode de résolution.

4.2 3 Résultats

On a utilisé 3 datasets pour étudier cette approche itérative. On va présenter à titre d'exemple les résultats obtenus à partir des données présentés dans la figure 31.

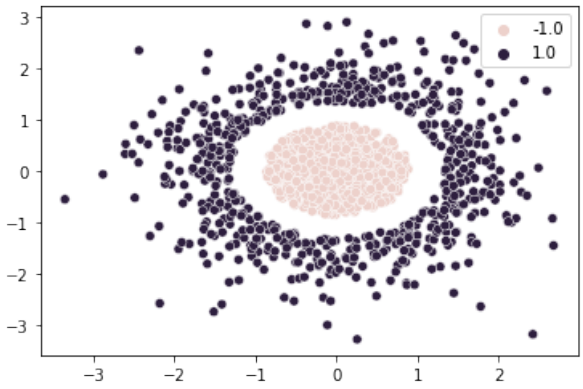

FIGURE 31 – Données non séparables linéairement

En appliquant cette approche avec les 3 méthodes de maximisation sur cette base de donnée, on est arrivé à générer 3 tableaux de résultats pour chaque type d'attaque. On présente ici à titre d'exemple le tableau associé à la méthode simple (avec le mode *Restrained*).

	f_attack=0	f_attack=0.3	f_attack=0.5	f_attack=0.7	f_attack=0.9
AD-svm restrained, Ce=1, Cd = 0.9	1.000000	**0.905983**	**0.649573**	**0.535613**	0.532764
AD-svm restrained, Ce=1, Cd = 0.7	1.000000	**1.000000**	**0.746439**	**0.541311**	0.532764
AD-svm restrained, Ce=1, Cd = 0.5	0.800570	0.800570	**0.777778**	0.401709	0.333333
AD-svm restrained, Ce=1, Cd = 0.3	0.823362	0.777778	0.529915	**0.717949**	0.358974
AD-svm restrained, Ce=1, Cd = 0.1	1.000000	0.851852	0.606838	0.532764	0.532764
SVM	1.000000	0.851852	0.606838	0.532764	0.532764

FIGURE 32 – Tableau de comparaison : données non séparables linéairement

À partir de la figure 32, on constate qu'avec une configuration ($C_\xi = 1, C_\delta = 0.9$) ou ($C_\xi = 1, C_\delta = 0.7$), la performance est toujours aussi ou plus bonne qu'un *SVM normal*. On arrive ainsi à valider l'utilité de notre approche itérative sur les données fictives.

5. Expériences : données de fraude

Dans cette partie, on a entraîné un *SVM* sur nos données artificielles de fraude. On présente les résultats dans le paragraphe suivant.

5.1. SVM

On a essayé d'entraîner un *SVM* avec un **kernel** linéaire pour la classification des fraudes, mais on n'a pas trouvé de bons résultats. Les performances des *SVMs* avec des **kernels** différents est dans l'annexe 1..

Cependant, on arrive à créer un *SVM* (**kernel** RBF) avec un *f1 score* de 0.91 pour *Validation set* et *0.88* pour la *Test set* (meilleures performances : même score que le *Random Forest*). En deuxième lieu, on a calculé la variance du *recall* de nos meilleurs modèles sur les 6 mois de la *Test set*. On trouve les résultats suivants : *XGBoost* : 5.98, *Random Forest* : 3.79, *Neural Network* : 3.6 et *SVM* : 2.29.

Notre objectif comme défini dans 3. est double : avoir un module robuste contre des attaques possibles (on valide ça par des tableaux comme dans la figure 30), et aussi minimiser la variance du *recall* sur la *Test set*.

5.2. Hypothèses et simplifications

L'approche décrite dans [26] consiste à varier tous les attributs d'une façon indépendante. Dans notre cas, on va simplifier cette approche pour créer un premier prototype. Les attributs utilisés pour entraîner le SVM sont :
(amt, delta_avg_amt, avg_amt, delta_amt, hour, delta_avg_amt_category, avg_amt_category, dob, delta_amt_category, delta_avg_amt_category_job, delta_time, gender, x0_entertainment, x0_food_dining, x0_gas_transport, x0_grocery_net, x0_grocery_pos, x0_health_fitness, x0_home, x0_kids_pets, x0_misc_net, x0_misc_pos, x0_personal_care, x0_shopping_net,

x0_shopping_pos, x0_travel)

On fait alors les choix suivants :

— On effectue la variation sur *amt* et *hour* : ces deux ont une contribution importante pour plusieurs modèles (c'est clair dans l'**EDA** et le **Feature importance** de *XGBoost*). Le fraudeur peut manipuler ces attributs et une attaque qui touche à ces 2 attributs peut être alors très sévère.

— Les attributs *dob, gender* et les attributs qui commencent par avg (qui concernent seulement la moyenne des achats). Cependant, ces attributs avg seront affectés dans le futur (pas la transaction concernée) par des changements dans *amt* et *hour* : on choisit de négliger cet effet collatéral, car c'est plus compliqué à modéliser et les changements sont négligeables.

— Les attributs qui commencent par *delta_* sont liés à *amt* et *hour*, ils changent en parallèle. On choisit aussi de négliger ce changement pour simplifier les traitements.

— Les attributs qui représentent la catégorie (qui commencent par *x0_*) : *category* est un variable catégorique, donc on doit définir une relation d'ordre (ou utiliser un *LE*) pour faire la variation de cet attribut. Dans ce prototype, on choisit de ne pas modifier sa valeur, même si le fraudeur en réalité a la possibilité de la manipuler.

5.3. Méthode de recherche

Pour générer les *SVMs* résilient contre les attaques, on réduit le processus itératif à une seule itération (le temps de calcul est plus important pour cette base de données). On utilise pour chaque configuration le *SVM* entraîné sur les données de fraude d'origine, on génère la meilleure attaque δ en respectant les contraintes de déplacement, puis finalement on entraîne un *SVM* (de mêmes paramètres que le SVM initial) sur les données modifiées. Les résultats de cette approche sont présentés dans la partie suivante.

Pour trouver la meilleure attaque, on teste 3 méthodes de maximisation (évoquées dans la partie 4.2 2). En appliquant ces méthodes sur plusieurs configurations (en variant Cf pour *Free Range* et C_δ pour *Restrained*) [11], on constate le suivant :

— La méthode simple est la plus efficace : moins de temps de calcul et plus de maximisation

— Les algorithmes génétiques sont les plus lents

— La descente de gradient est moins performante (en termes de maximisation).

On continue alors de travailler avec la méthode simple.

11. Les détails de comparaisons sont dans l'annexe 2..

5.3 1 Version 1 : données pures

Dans cette première approche, on calcule la meilleure attaque, on l'effectue sur notre *Training set*, et on entraîne notre *SVM* de nouveau. Avant de faire des expériences qui peuvent montrer si les *SVMs* générés satisfassent ou non nos objectifs, on a défini un critère éliminatoire : les *SVMs* générés doivent avoir un *f1 score* aussi bien qu'un *SVM normal* sur la *Validation set* (avec tolérance de 0.01). On montre dans la figure 33 les performances des *SVMs* qui anticipent une attaque *Free Range* contre un *SVM normal*.

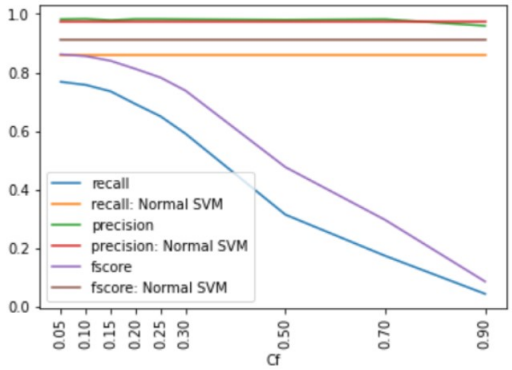

FIGURE 33 – Évolution des performances : SVM en mode Free range vs SVM normal

Le *f1 score* d'un *SVM* qui anticipe des attaques *Free Range* est toujours inférieur à un *SVM normal* (différence de 0.04 pour Cf=0.05). On remarque que lorsqu'on augmente l'intensité de l'attaque anticipée, les performances sur la *Validation set* détériorent : cela n'est pas le comportement attendu, on cherche un *SVM* qui est valide pour plusieurs intensités d'attaque (et dans l'absence d'une attaque).
On fait les mêmes expériences pour le mode *Restrained*, on trouve qu'avec $C_\delta = 0.95$ un *f1 score* qui est proche à un *SVM normal*, mais en contrepartie la même allure présente dans la figure 33.

On constate deux choses :

— Apporter des changements sur les valeurs de *amt* et *hour* (des transactions frauduleuses) peut induire des changements importants dans les performances d'un modèle : cela valide notre hypothèse de l'importance de ces deux attributs.

— La substitution de données par des données générées avec une seule itération n'est pas pratique pour satisfaire nos buts : soit parce qu'on tombe dans un optimum local, ou car on perd des informations utiles en éliminant les données d'origine.

Dans la partie suivante, on teste une autre façon pour entraîner nos *SVMs*.

5.3 2 Version 2 : données hybrides

Le problème rencontré dans la partie précédente peut être dû à la perte des informations utiles présentes dans les instances de fraude d'origine. Donc, on a décidé de

changer une étape dans le processus d'entraînement. Au lieu d'appliquer les attaques sur les données de fraude d'origine, on fait deux copies de ces données, on applique les changements sur une copie et on laisse l'autre intacte. On fait l'entraînement après sur les transactions légitimes et les deux copies de données de fraudes (les transactions légitimes ne sont pas dupliquées, car elles ne sont pas changées lors d'une attaque). On appelle ces données : les **données hybrides**.

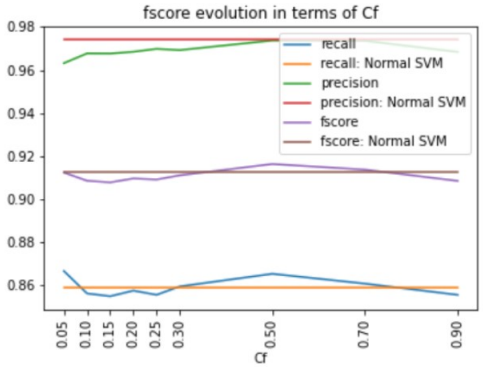

FIGURE 34 – Évolution des performances : SVM en mode Free range (données hybrides) vs SVM normal

La figure 34 montre la stabilité de *f1 score* par rapport à la première approche. On décide alors de faire les expériences sur les *SVMs* générés avec les données hybrides.

6. Résultats et conclusion

6.1. Objectif 1 : robustesse contre les attaques

Cet objectif concerne la création des *SVMs* robustes contre des attaques éventuelles. Pour cela, on applique une attaque de cette forme :

$$x_{ij} = x_{ij} + \delta_{ij} \text{ avec } \delta_{ij} = f_{attack}(x_j^t - x_{ij})$$

x_i sont les instances avec $y_i = 1$, j définit l'attribut changé : *amt* ou *hour*. x_j^t contient la moyenne des valeurs des transactions légitimes de la *Validation set* pour l'attribut j. On trouve ainsi deux tableaux, pour le mode *Free Range* et pour le mode *Restrained*. On montre le graphe qui présente les résultats du mode *Free Range* à titre d'exemple dans la figure 35.

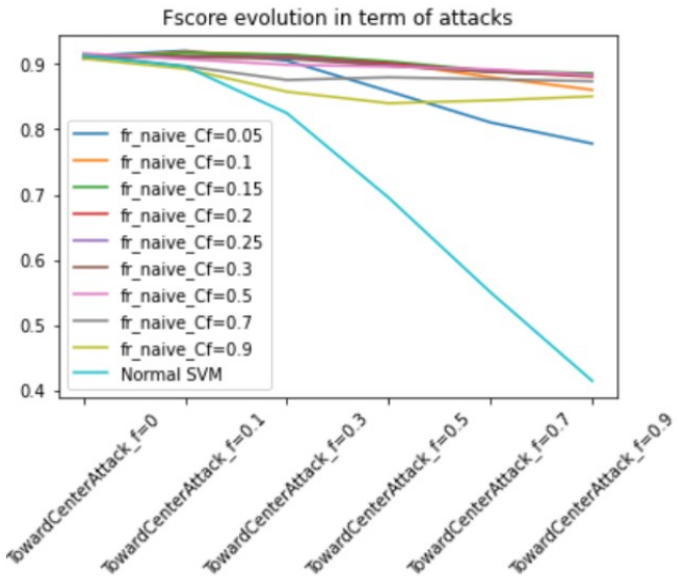

FIGURE 35 – Évolution des performances : variation de l'intensité des attaques

On observe que le *f1 score* de l'*SVM normal* se détériore lorsqu'on augmente l'intensité de l'attaque. Les autres *SVMs* qui anticipent des attaques sont relativement stables. Ils ont des performances similaires à un *SVM normal* et performent toujours mieux dans le cas d'attaque. On a obtenu des résultats similaires pour le mode *Restrained*.

L'utilisation d'un *SVM* qui anticipe des attaques donne alors des performances similaires dans l'absence d'attaque, et peut potentiellement être utile dans le cas d'une attaque qui est similaire à la forme de l'attaque définie. On constate qu'on est arrivé à satisfaire le premier objectif.

6.2. Objectif 2 : stabilité

On a comparé la stabilité des modèles générés avec les deux modes attaques en mesurant la variance du *recall*, sur des données non modifiées (pas d'attaque). Pour la phase de développement, on a mesuré cette variance sur les mois de la *Validation set*. On trouve pour le mode *Free Range* les résultats dans la figure 36. On trouve des variances qui sont plus petites (par rapport à un *SVM normal*) pour la majorité des valeurs de Cf. On ne trouve pas cependant une allure monotone. On trouve des résultats similaires avec le mode *Restrained*.

On fait les mêmes expériences sur le *Test set*, pour valider nos résultats. Par contre, on trouve que les modèles qui anticipent des attaques ont des variances plus importantes (et donc moins stables) par rapport à un *SVM normal*. On observe ça dans la figure 37.

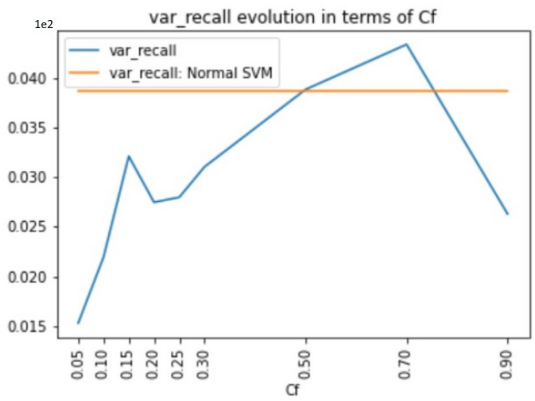

FIGURE 36 – Évolution de la variance du recall : Validation set

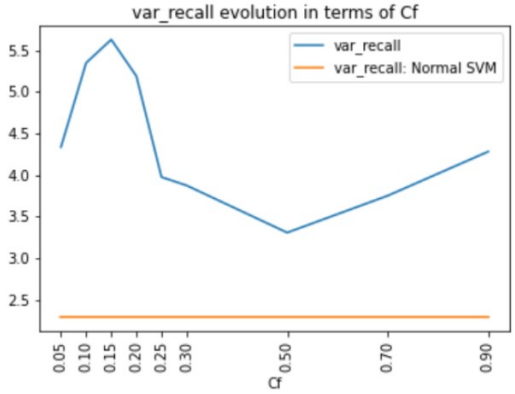

FIGURE 37 – Évolution de la variance du recall : Test set

On constate alors qu'on n'arrive pas à stabiliser le *recall* avec l'introduction de la notion de l'attaque de cette forme. On associe ce résultat aux possibilités suivantes :

- La définition de l'attaque :
  - Varier deux attributs seulement.
  - Ne pas varier les attributs qui sont liés en parallèle.
  - Ne pas trouver un optimum global : seule itération, processus itératif ou méthode de maximisation.
- Les données :
  - Les données ne sont pas réelles : le changement de la distribution dans le *Test set* peut être aléatoire.
  - Les données n'exploitent pas les vulnérabilités du *SVM* car elles ne sont pas une réponse du déploiement de ce modèle.
- La définition de l'objectif :
  - L'article [26] n'évoque pas la possibilité de stabiliser le *recall* avec cette approche.
  - Cet objectif est plus adéquat dans le cas de défaillance du *recall* : courbe décroissante.

Dans le paragraphe suivant, on discute davantage les étapes qui suivent pour approfondir l'étude faite avec la Théorie des Jeux, et améliorer et solidifier les résultats obtenus.

6.3. Prochaines étapes

En se basant sur les résultats présentés dans les deux paragraphes précédents, on définit les prochaines étapes de ce projet.

Pour le premier objectif : on a montré que lorsqu'on modélise une attaque (en chan-

geant *amt* et *hour*), les performances d'un *SVM normal* sont toujours proches ou inférieures à celles des *SVMs* qui anticipent les attaques (même si on n'a pas d'attaques, on trouve des résultats proches).

- On peut effectuer plus des changements sur les attributs :
  - Les attributs : changer les attributs qui sont liés en parallèle (comme *delta_amt* en changeant *amt*).
  - Définir une relation d'ordre pour *category* (selon le taux de fraude par catégorie par exemple) et donner la possibilité au fraudeur de la changer durant une attaque.
- Modéliser plusieurs formes d'attaques effectuées sur la *Validation set*, pour voir si les *SVMs* qui anticipent des attaques performent toujours mieux sur une variété d'attaques :
  - Ces attaques peuvent englober un terme de pénalité pour le fraudeur : lié à *amt* et *category* par exemple.
  - On peut aussi définir d'autres formes d'attaques : en définissant x_i^t pour chaque instance d'une façon différente (milieu des k-plus-proches voisins : moyenne locale au lieu d'une moyenne globale), d'une façon probabiliste (en choisissant x_i^t pour chaque instance de fraude avec une probabilité inversement proportionnelle à la distance de chaque point des fraudes).

Lorsqu'on utilise plusieurs types d'attaques, avec des hypothèses plausibles qui peuvent refléter des aspects de la réalité, on renforce la crédibilité de notre modèle.

Pour le deuxième objectif, il peut s'avérer utile de suivre les prochaines pistes :

- Faire les mêmes changements qui concernent les attributs (mentionnés dans les prochaines étapes liées à l'objectif 1).
- Essayer l'approche itérative mentionnée dans la partie 4.2 1, sans se limiter à une seule itération.
- Générer plusieurs modèles à partir des données générées par des différentes valeurs de *sampling strategy*, et par des coefficients *Cf* petites (entre 0 et 0.1), et des coefficients C_δ proches de 1 (entre 0.9 et 1) : effectuer le **bagging** avec un vote majoritaire entre ces modèles.
- Tester cette approche sur des données réelles.

Chapitre VII.

Conclusion

Au début de ce projet, on a défini 4 problématiques à parcourir. Dans un premier temps, on a présenté un **Benchmark** initial qui contenait 7 modèles de *Machine Learning*. Finalement, on est arrivé à **optimiser** 4 modèles : *Random Forest*, *XGBoost*, *Neural Network* et *SVM*. Les performances de ces derniers ont dépassé d'une façon remarquable celles des premiers modèles. Dans ces parties, on a traité à la fois le problème des données **déséquilibrées** ainsi que le comportement **imitatif** du fraudeur.
Par rapport au **temps** de réponse, notre démarche a été de **comparer** les temps de réponse de 3 algorithmes *Random Forest*, *XGBoost* et *Neural Network* par une simulation. La dernière problématique était le comportement **dynamique** du fraudeur. Pour cela, on a implémenté une approche basée sur la **Théorie des Jeux**. Avec le premier prototype qu'on a développé, on a réussi à réaliser un objectif sur deux pour cette partie.

« A conclusion is simply the place where you got tired of thinking »

(Martin H. Fischer)

Ce stage n'est qu'une ouverture dans la détection automatisée de fraude bancaire. On a participé à la mise en place des briques fondamentales pour un système robuste de prédiction de fraude bancaire et il reste plusieurs problématiques à adresser. Les enjeux futures imaginés pour poursuivre ces travaux sont : améliorer l'approche *Théorie des Jeux* et explorer les techniques issues de la *Théorie des Graphes*. Un point important à ne pas oublier est la performance du système. On pourrait imaginer un recours aux calculs massifs via du **GPU** ou du calcul parallèle via des **Frameworks** de type **Spark**.

Sur le plan personnel, ce stage était une chance pour appliquer ce que j'ai appris au cours de mes études : soit des mathématiques, du développement Python et du Machine Learning. J'ai eu la chance de pratiquer davantage plusieurs modules Python : notamment Sci-Kit Learn, Keras, Pandas, Matplotlib, NumPy, Seaborn ... Cela m'a permis de me familiariser plus avec ces bibliothèques et de découvrir plusieurs fonctionnalités. La partie *Théorie des Jeux* était la plus compliquée, j'ai lu plusieurs articles avec des diverses approches pour comprendre le plus possible les problématiques impliquées. Finalement, je suis familiarisé avec plusieurs modèles d'apprentissage automatique et les problématiques autour de chacun de ces modèles.

Bibliographie

[1] https://www.societe.com/societe/acensi-finance-488984014.html.

[2] Acensi. https://www.acensi.fr/page/accueil.

[3] Credit card fraud statistics, 2021. https://shiftprocessing.com/credit-card-fraud-statistics/.

[4] Documentation labelencoder. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html.

[5] Documentation onehotencoder. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html.

[6] Documentation standardscaler. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[7] https ://www.tutorialspoint.com/scikit_learn/scikit_learn_knn_learning.htm.

[8] Understanding xgboost algorithm. https://www.mygreatlearning.com/blog/xgboost-algorithm.

[9] Xgboost python package. https://xgboost.readthedocs.io/en/latest/python/python_api.html.

[10] Andrew Moore Dan Pelleg. Accelerating exact k-means algorithms with geometric reasoning. page 277, 08 1999.

[11] Chih-Chung Chang et Chih-Jen Lin. Libsvm : A library for support vector machines. pages 4, 34, 2001.

[12] Geoff Gordon et Ryan Tibshirani. Karush-kuhn-tucker conditions. https://www.cs.cmu.edu/~ggordon/10725-F12/slides/16-kkt.pdf.

[13] Wei Liu et Sanjay Chawla. A game theoretical model for adversarial learning. 2009.

[14] Wei Liu et Sanjay Chawla. A game theoretical model for adversarial learning. 2009.

[15] Gabriele Gianini. Managing a pool of rules for credit card fraud detection by a game theory based approach. page 552, 2020.

[16] Brandon Harris. Fake credit card transaction data generator. https://github.com/namebrandon/Sparkov_Data_Generation.

[17] Mehdi Mirza Bing Xu David Warde-Farley Sherjil Ozair Aaron Courville Yoshua Bengio Ian J. Goodfellow, Jean Pouget-Abadie. Generative adversarial nets.

[18] Chetan Kumar. A top-down approach to classify enzyme functional classes and sub-classes using random forest. page 2, 2012.

[19] Nathan Fogal Stephen Adams Donald E. Brown Mary Frances Zeager, Aksheetha Sridhar and Peter A. Beling. Adversarial learning in credit card fraud detection. 2017.

[20] Derek Monner. A generalized lstm-like training algorithm for second-order recurrent neural networks. page 3, 2012.

[21] Joseph B. Collins Prithviraj Dasgupta. A survey of game theoretic approaches for adversarial machine learning in cybersecurity tasks. 2019.

[22] Erica Sandberg. The average number of credit card transactions per day year. `https://www.cardrates.com/advice/number-of-credit-card-transactions-per-day-year`.

[23] Terence Shin. A beginner-friendly explanation of how neural networks work. `https://towardsdatascience.com/a-beginner-friendly-explanation-of-how-neural-networks-work-55064db60df4`.

[24] Manosij Ghosha Sanjoy Kumar Saha Sumit Misraa, Soumyadeep Thakura. An autoencoder based model for detecting fraudulent credit card transaction. page 255, 2020.

[25] Sutapat Thiprungsri. Cluster analysis for anomaly detection in accounting data : An audit approach. pages 71, 72, 2011.

[26] B. Kantarcioglu et B. Xi Y. Zhou, M. Murat. Adversarial support vector machine learning. 2012.

Annexes

Annexe A

Démonstrations mathématiques

1. Forme dual du problème d'optimisation sous contraintes

On part du problème initial, en posant $d = 2$ et $\mu = \frac{1}{2}$:

$$\min_{\omega,b,\xi_i,t_i,u_i,v_i} f$$

$$\text{avec} \begin{cases} f = \frac{1}{2}\|\omega\|^2 + C\sum_i \xi_i \\ \xi_i \geqslant 0 \\ \xi_i \geqslant 1 - y_i(\omega.x_i + b) + t_i \\ t_i \geqslant \sum_j C_f(v_{ij}(x_j^{max} - x_i^j) - u_{ij}(x_j^{min} - x_i^j)) \\ u_i - v_i = \frac{1}{2}(1 + y_i)\omega \\ u_i \geqslant 0 \\ v_i \geqslant 0 \end{cases}$$

On passe au problème dual en utilisant les conditions de **K.K.T** [12] :

$$\max g = \max \frac{1}{2}\|\omega\|^2 + C\sum_i \xi_i - \sum_i \alpha_i.\xi_i + \sum_i \beta_i((1 - y_i(\omega.x_i + b) + t_i) - \xi_i)$$

$$+ \sum_i \gamma_i(\sum_j C_f(v_{ij}(x_j^{max} - x_i^j) - u_{ij}(x_j^{min} - x_i^j)) - t_i)$$

$$+ \sum_i \rho_i(\frac{1}{2}(1 + y_i)\omega - (u_i - v_i)) +$$

$$- \sum_i \Gamma_i.u_i - \sum_i \lambda_i.v_i$$

$$\forall i; 1 \leqslant i \leqslant n : \begin{cases} \textit{Conditions du premier ordre :} \\ \textbf{1.}\ \frac{\partial g}{\partial \omega} = 0 \Leftrightarrow \omega^T = \sum_i \beta_i.y_i.x_i + \frac{1}{2}\sum_i \rho_i^T(1+y_i) \\ \textbf{2.}\ \frac{\partial g}{\partial b} = 0 \Leftrightarrow \sum_i \beta_i.y_i = 0 \\ \textbf{3.}\ \frac{\partial g}{\partial \xi_i} = 0 \Leftrightarrow \beta_i = C - \alpha_i \\ \textbf{4.}\ \frac{\partial g}{\partial t_i} = 0 \Leftrightarrow \gamma_i = \beta_i \\ \textbf{5.}\ \frac{\partial g}{\partial u_i} = 0 \Leftrightarrow \rho_i + Cf.\gamma_i(x^{min} - x_i) = -\Gamma_i \\ \textbf{6.}\ \frac{\partial g}{\partial v_i} = 0 \Leftrightarrow Cf.\gamma_i(x_i - x^{max}) - \rho_i = -\lambda_i \\ \textit{Conditions de relâchement supplémentaires :} \\ \textbf{7.}\ \alpha_i, \beta_i, \gamma_i \geqslant 0 \ \textit{et}\ \Gamma_i, \lambda_i \in \mathbb{R}+^2 \\ \textbf{8.}\ \rho_i \in \mathbb{R}^2 \\ \textbf{9.}\ \alpha_i.\xi_i = 0 \\ \textbf{10.}\ \beta_i((1 - y_i(\omega.x_i + b) + t_i) - \xi_i) = 0 \\ \textbf{11.}\ \gamma_i(C_f(v_i(x^{max} - x_i) - u_i(x^{min} - x_i)) - t_i) = 0 \\ \textbf{12.}\ \rho_i(\frac{1}{2}(1+y_i)\omega - (u_i - v_i)) = 0 \\ \textbf{13.}\ \Gamma_i.u_i = 0 \\ \textbf{14.}\ \lambda_i.v_i = 0 \end{cases}$$

Le problème dual devient :

$$\max g = \max \frac{1}{2}\|\omega\|^2 - \sum_i \beta_i.y_i.\omega.x_i) - \sum_i \rho_i \frac{1}{2}(1+y_i)\omega$$

$$= \max \frac{1}{2}\|\omega\|^2 - \omega(\sum_i \beta_i.y_i.x_i) + \frac{1}{2}\sum_i \rho_i^T(1+y_i))$$

$$= \max \frac{1}{2}\|\omega\|^2 - \|\omega\|^2$$

$$= \min \frac{1}{2}\|\omega\|^2$$

$$= \min_\alpha \frac{1}{2}\alpha^T P\alpha + q^T\alpha$$

$$\text{avec} \begin{cases} \textbf{3.}, \textbf{4.}, \textbf{5.}, \textbf{6.}, \textbf{7.}\ \textit{et}\ \textbf{8.} \Rightarrow G\alpha \leqslant h \\ \textbf{2.} \Rightarrow A\alpha = b \end{cases}$$

— $\alpha = (\alpha_i)_{1 \leqslant i \leqslant 3n}$ avec $\begin{cases} \alpha_i = \beta_i; 1 \leqslant i \leqslant n \\ \alpha_i = \rho_i^1; n+1 \leqslant i \leqslant 2n \\ \alpha_i = \rho_i^2; 2n+1 \leqslant i \leqslant 3n \end{cases}$

$$P = \begin{pmatrix} P_{11} & P_{12} & P_{13} \\ P_{12}^T & P_{22} & 0_{n\times n} \\ P_{13}^T & 0_{n\times n} & P_{22} \end{pmatrix} \quad \text{avec} \begin{cases} P_{11} = y_i.x_i.x_j^T.y_j & \text{pour } 1 \leqslant i,j \leqslant n \\ P_{12} = -\frac{1}{2}.y_i.x_i^0(1+y_j) & \text{pour } 1 \leqslant i,j \leqslant n \\ P_{13} = -\frac{1}{2}.y_i.x_i^1(1+y_j) & \text{pour } 1 \leqslant i,j \leqslant n \\ P_{22} = 1\frac{1}{2}(1+y_i)(1+y_j) & \text{pour } 1 \leqslant i,j \leqslant n \end{cases}$$

$$q = (q_i)_{1\leqslant i\leqslant 3n} \text{ avec } \begin{cases} q_i = -1 & \text{si } 1 \leqslant i \leqslant n \\ q_i = 0 & \text{sinon} \end{cases}$$

$$G = \begin{pmatrix} -\mathbb{1}_{n\times n} & 0_{n\times n} & 0_{n\times n} \\ G_{21} & \mathbb{1}_{n\times n} & 0_{n\times n} \\ G_{31} & 0_{n\times n} & \mathbb{1}_{n\times n} \\ G_{41} & -\mathbb{1}_{n\times n} & 0_{n\times n} \\ G_{51} & 0_{n\times n} & -\mathbb{1}_{n\times n} \\ \mathbb{1}_{n\times n} & 0_{n\times n} & 0_{n\times n} \end{pmatrix} \text{ avec } \begin{cases} G_{21} = \mathbb{1}_{i=j}.Cf(x_0^{min} - x_i^0) & \text{pour } 1 \leqslant i,j \leqslant n \\ G_{31} = \mathbb{1}_{i=j}.Cf(x_1^{min} - x_i^1) & \text{pour } 1 \leqslant i,j \leqslant n \\ G_{41} = \mathbb{1}_{i=j}.Cf(x_i^0 - x_0^{max}) & \text{pour } 1 \leqslant i,j \leqslant n \\ G_{51} = \mathbb{1}_{i=j}.Cf(x_i^1 - x_1^{max}) & \text{pour } 1 \leqslant i,j \leqslant n \end{cases}$$

$$h = (h_i)_{1\leqslant i\leqslant 6n}^T \text{ avec } \begin{cases} h_i = 0 & \text{si } 1 \leqslant i \leqslant 5n \\ h_i = C & \text{sinon} \end{cases}$$

$$A = (a_i)_{1\leqslant i\leqslant 3n} \text{ avec } \begin{cases} a_i = y_i & \text{si } 1 \leqslant i \leqslant n \\ a_i = 0 & \text{sinon} \end{cases}$$

$$b = 0$$

On résout le problème dual , et on trouve ainsi l'expression de ω et b à partir de α :

$$\omega = \sum_{i=1}^{n} \alpha_i.y_i.x_i + \frac{1}{2}\sum_{i=1}^{n}(1+y_i)(\alpha_{i+n}.x_i^0, \alpha_{i+2n}.x_i^1)$$

On pose $Cf > 0$, $\beta_i \neq 0$, $\beta_i \neq C$ et $\rho_i = 0$:

$$\begin{cases} \beta_i \neq 0 \ et \ \beta_i \neq C \Rightarrow \alpha_i \neq 0 \\ \alpha_i \neq 0 \ et \ \textbf{9.} \ \Rightarrow \xi_i = 0 \ \textbf{(a)} \\ \rho_i = 0 \ et \ \textbf{5.} \ \Rightarrow \Gamma_i > 0 \ ou \ x^{min} = x_i \\ \textbf{13.} \ et \ (\Gamma_i > 0 \ ou \ x^{min} = x_i) \Rightarrow u_i = 0 \ ou \ x^{min} = x_i \Rightarrow u_i(x^{min} - x_i) = 0 \\ \rho_i = 0 \ et \ \textbf{6.} \ \Rightarrow \Gamma_i > 0 \ ou \ x^{max} = x_i \\ \textbf{14.} \ et \ (\Gamma_i > 0 \ ou \ x^{max} = x_i) \Rightarrow u_i = 0 \ ou \ x^{max} = x_i \Rightarrow v_i(x^{max} - x_i) = 0 \\ v_i(x^{max} - x_i) = 0, u_i(x^{min} - x_i) = 0, Cf > 0, \beta_i > 0 \ et \ \textbf{11.} \ \Rightarrow t_i = 0 \ \textbf{(b)} \\ \beta_i \neq 0, \textbf{(a)}, \ \textbf{(b)} \ et \ \textbf{10.} \Rightarrow 1 - y_i(\omega.x_i + b) = 0 \end{cases}$$

Donc pour trouver b, il suffit de trouver $i \leqslant n$ tel que $\alpha_i \neq 0$, $\alpha_i \neq C$, $\alpha_{i+n} = \alpha_{i+2n} = 0$:

$$b = y_i - \omega.x_i$$

2. Descente de gradient

On rappelle l'expression de notre problème d'optimisation :

$$\arg\min_{\omega,b} \sum_i \max_{\delta_i} \lfloor 1 - y_i(\omega.\phi(x_i + \delta_i) + b) \rfloor_+$$

avec $\begin{cases} \phi(x).\phi(y) = K(x,y) = e^{-\gamma\|x-y\|^2} \\ \delta_i = 0 \text{ si } y_i = -1 \end{cases}$

L'adversaire cherche alors à maximiser le terme lié à δ_i si $y_i = 1$. Alors pour chaque i tel que $y_i = 1$, l'adversaire cherche à trouver :

$$\arg\max_{\delta_i} \lfloor 1 - 1 \times (\omega.\phi(x_i + \delta_i) + b) \rfloor_+ =$$

$$\arg\max_{\delta_i} (1 - (\omega.\phi(x_i + \delta_i) + b)) =$$

$$\arg\min_{\delta_i} (\omega.\phi(x_i + \delta_i) + b) =$$

$$\arg\min_{\delta_i} (\omega.\phi(x_i + \delta_i))$$

Or on a [11] :

$$\omega = \sum_{\{j|x'_j \in SV\}} \alpha_j.y_j.\phi(x'_j)$$

avec $\begin{cases} SV = \{x'_j \in x_{train}; \alpha_j > 0\} \\ x_{train} : \text{ensemble de points d'entraînement pour le SVM} \end{cases}$

L'adversaire cherche alors à minimiser pour chaque i tel que $y_i = 1$ [11] :

$$\sum_{\{j|x'_j \in SV\}} \alpha_j.y_j.\phi(x'_j).\phi(x_i + \delta_i) =$$

$$\sum_{\{j|x'_j \in SV\}} \alpha_j.y_j.K(x'_j, x_i + \delta_i)$$

On dérive K par rapport à δ_i :

$$\frac{\partial}{\partial \delta_i} K(x'_j, x_i + \delta_i) =$$

$$\frac{\partial}{\partial \delta_i} e^{-\gamma(x'_j.x'_j + (x_i+\delta_i)(x_i+\delta_i) - 2x'_j(x_i+\delta_i))} =$$

$$-\gamma(2(x_i + \delta_i) - 2x'_j).K(x'_j, x_i + \delta_i)$$

Finalement, avec un *learning rate* qu'on notera **lr**, on met à jour l'expression de tous les δ_i tels que $y_i = 1$ de la façon suivante :

$$\delta_i \leftarrow \delta_i - lr. \sum_{\{j|x'_j \in SV\}} 2\gamma(x'_j - x_i - \delta_i) \left(\alpha_j.y_j.K(x'_j, x_i + \delta_i)\right)$$

Annexe B

Comparaisons supplémentaires

1. Comparaison entre les kernels

Dans la figure ci-dessous, on montre les performances, en terme de *f1 score* de la classe 1, des différents *SVMs* avec 4 types de **kernels**. Les performances sont mesurés sur le *Validation set* des données de fraude artificielles [1].

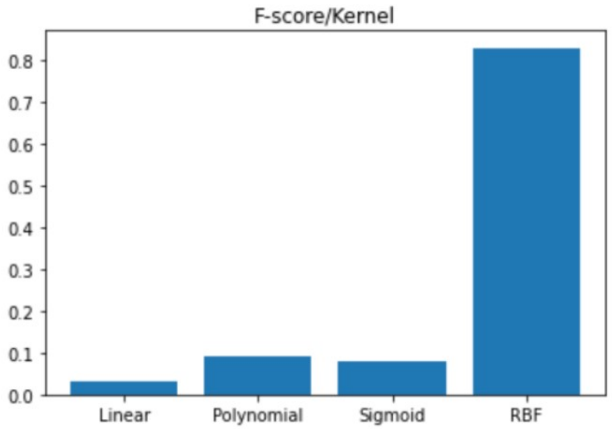

FIGURE 38 – F1 score : SVMs avec des différents kernels

2. Comparaison entre les méthodes de maximisation

Pour comparer les méthodes de maximisation : méthode simple, algorithme génétique et la descente de gradient, on applique chaque méthode sur 4 attaques différentes et on mesure le loss ajouté. Les 4 attaques comportent :

1. Les performances de SVM avec RBF sont mesurés avant la phase d'optimisation.

— Deux attaques modérées : Free Range avec $Cf = 0.1$ et Restrained avec $C_\delta = 0.9$ et $C_\xi = 1$.

— Deux attaques agressives : Free Range avec $Cf = 0.9$ et Restrained avec $C_\delta = 0.1$ et $C_\xi = 1$.

Dans la figure 39, on montre le loss ajouté par chaque méthode de maximisation en respectant les contraintes associées à chaque attaque :

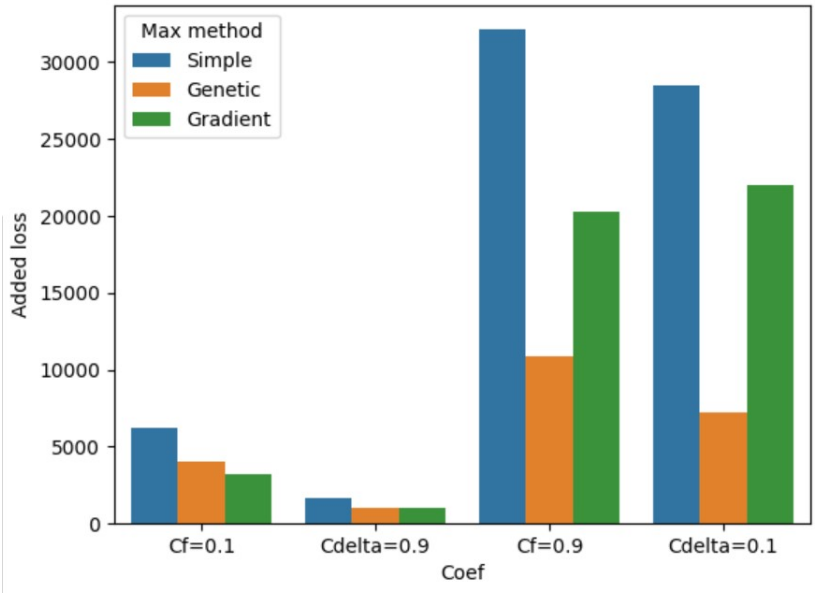

FIGURE 39 – Comparaison entre les méthodes de maximisation : loss ajouté

3. Comparaison entre les variances du recall : NN, RF et XGBoost

Dans ce paragraphe, on présente les résultats de l'application de notre approche hybride sur d'autres modèles de *Machine Learning*.

On a entraîné 3 modèles : notamment Neural Network, Random Forest et XGBoost, sur les mêmes données d'entraînement des SVMs Hybrides. On présente dans ce paragraphe les effets de cette approche sur la variance du recall dans la *Validation Set* et la *Test set*.

Neural Network : Dans les deux figures suivantes, on trouve parfois des coefficients ($Cf = 0.1$ et $C_\delta = 0.75$) qui produisent une variance moins importante que celle d'un

Neural Network normal sur les données de la *Validation Set*. Cependant, on ne trouve pas les mêmes résultats sur le *Test Set*. Avec des allures qui ne sont pas monotones, et qui ne montrent pas des similitudes apparentes entre les résultats de la *Validation set* et de la *Test set*, on ne peut pas conclure l'efficacité de ces méthodes pour diminuer la variance.

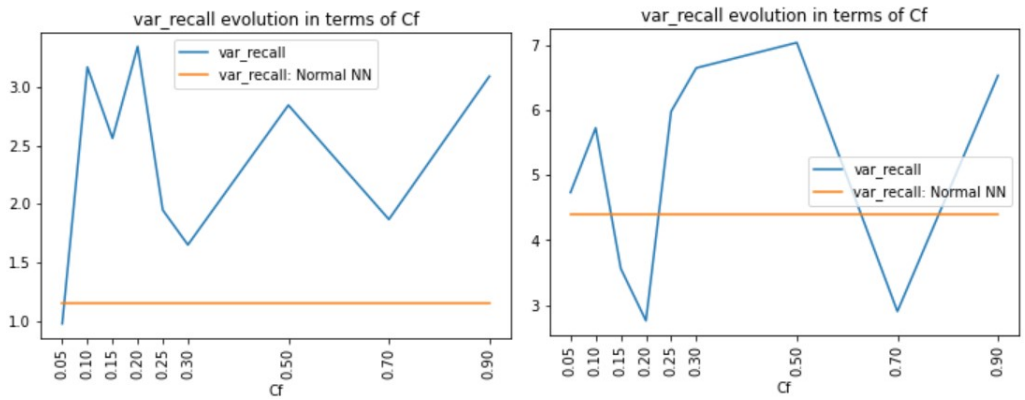

FIGURE 40 – Variance du recall de NN : Free Range (Validation-Test)

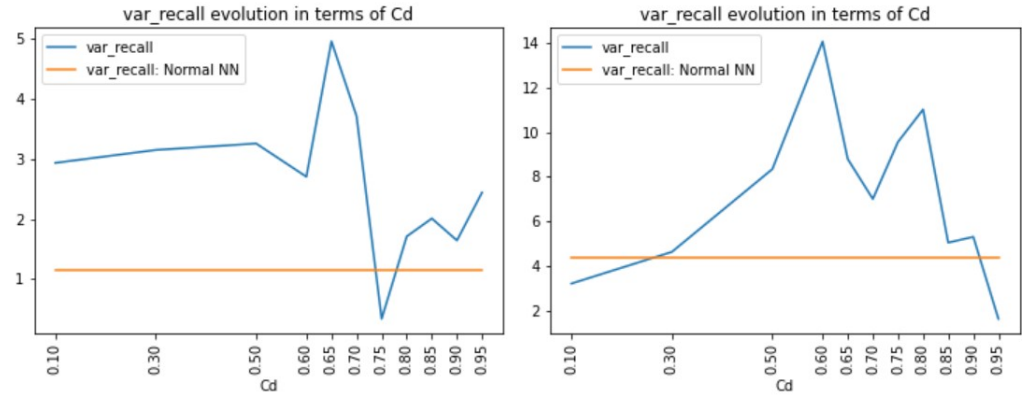

FIGURE 41 – Variance du recall de NN : Restrained (Validation-Test)

Random Forest : Dans les deux figures suivantes, les variances des modèles entraînés sur des données hybrides sont toujours plus importantes que le *Random Forest* normal [2].

2. Un OverSampling avec SMOTE est appliqué sur les données hybrides avant l'entraînement, comme pour le *Random Forest* normal

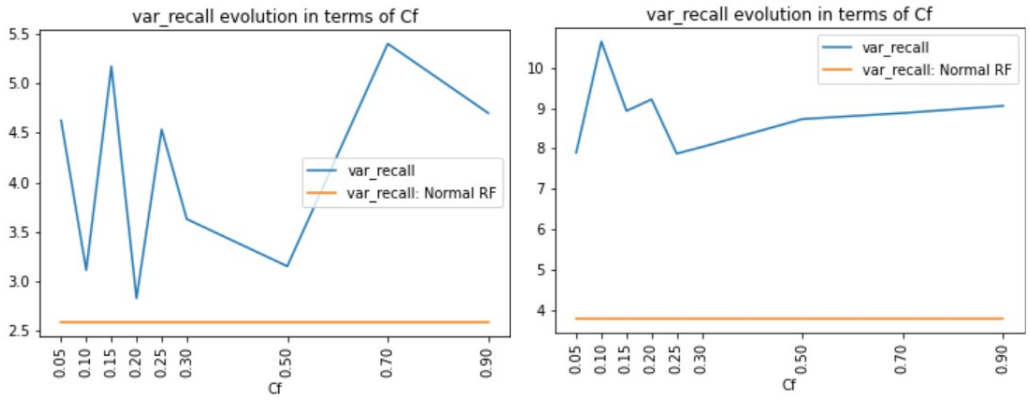

FIGURE 42 – Variance du recall de RF : Free Range (Validation-Test)

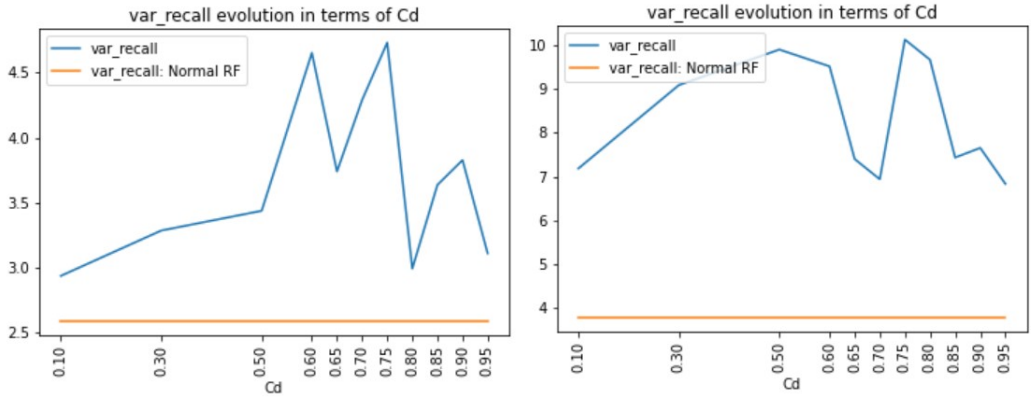

FIGURE 43 – Variance du recall de RF : Restrained (Validation-Test)

XGBoost : Dans les deux figures suivantes, on trouve des variances qui sont moins importantes pour les données de la *Validation set* . Cependant, ce n'est pas valable pour les données de la *Test set*, on trouve que la variance du recall est plus importante que celle du *XGBoost* normal.

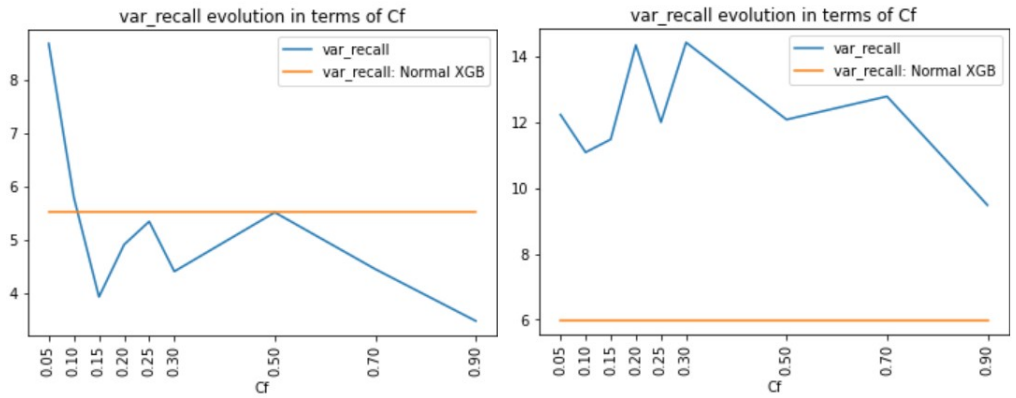

FIGURE 44 – Variance du recall de XGBoost : Free Range (Validation-Test)

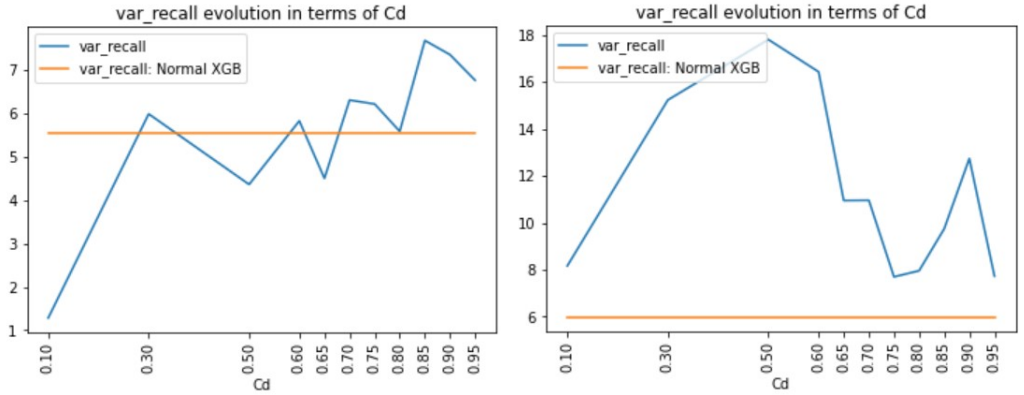

FIGURE 45 – Variance du recall de XGBoost : Restrained (Validation-Test)

Dans une prochaine étape, on peut générer des données qui sont mieux adaptées pour chaque modèle :

— Neural Network : on peut utiliser la fonction de loss **Binary Cross Entropy** pour générer des attaques lors de la phase de maximisation.
— Random Forest et XGBoost : on peut minimiser la *précision*, le *recall* ou le *f1 score* dans la phase de maximisation.