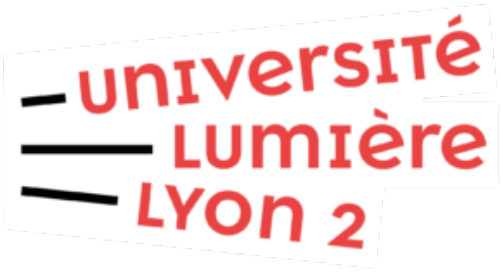

Fouille de Données Massives

Matière SISE

Adrien CASTEX, Célia MAURIN, Annabelle NARSAMA

lien projet github : https://github.com/adcastex/fouille_de_donnees.git

Janvier 2024

Table des matières

Résumé

La détection de fraude est un problème particulier de classification binaire : la proportion de fraude est sous-représentée en comparaison de la proportion de non-fraude. Nous proposons ainsi ici de créer différents modèles prédictifs à partir d'algorithmes capables de gérer le problème des classes déséquilibrées, afin de pouvoir établir le modèle le plus efficient pour la prédiction et la détection de fraude. Pour se faire, nous avons eu recours à SMOTEEN et Tomek Link pour ré-échantillonner les données, avant de lancer nos algorithmes d'analyses de données : arbres de décisions, forêts aléatoires, réseaux de neurones, modèles ensemblistes, régression logistique et k-means. À l'aide de la $F1$-mesure, nous avons évalué les performances de nos modèles qui restent très limitées. Notre meilleur score est au environs de 0.06. Le problème des classes très fortement déséquilibrées demeure un réel challenge dans l'apprentissage automatique.

Mots-clé : Apprentissage automatique, Apprentissage sensible aux coûts, Détection de fraude, Données déséquilibrées, Techniques de ré-échantillonnage

1 Introduction

Les données déséquilibrées sont un problème que nous pouvons rencontrer dans de nombreux domaines d'application, tels que le milieu médical (e.g., les individus malades ne représentent qu'un infime pourcentage parmi la population saine), le milieu industriel (e.g., une anomalie dans le processus de production d'une marchandise), ou encore le milieu bancaire (e.g., le nombre de fraudes est bien inférieur au nombre de non fraudes) (voir [1] pour plus de domaines d'application). Dans ces domaines d'application, les analyses statistiques doivent tenir compte de la sous-représentation de la classe d'intérêt (i.e., classe minoritaire) afin de pouvoir construire des modèles prédictifs capables de classifier correctement les exemples de la classe minoritaire.

De fait, les algorithmes d'apprentissage automatique classiques ne sont pas adéquats pour les jeux de données déséquilibrées. Ils se basent sur la probabilité maximale d'appartenir à une classe. De fait, si la classe d'intérêt est sous-représentée par rapport à la classe majoritaire, l'algorithme aura tendance à classer tous les exemples dans la classe majoritaire — et donc, à mal classifier les exemples de la classe minoritaire. De plus, ces algorithmes font bien souvent appel à l'accuracy pour évaluer les performances du modèle construit. Or, cette métrique perd tout son sens dans le contexte d'un jeu de données déséquilibrées [2] : elle ne prend pas en compte le déséquilibre des données et aura tendance à se focaliser sur les bonnes prédictions de la classe majoritaire. En effet, elle se base sur le nombre de bonnes prédictions par rapport au nombre total de données, indépendamment des classes et de leurs poids. Il convient par conséquent d'utiliser des métriques qui nous permettent d'évaluer la capacité du modèle à classer correctement les exemples de la classe minoritaire.

Dans leur article, Remy et al. (2021)[2] présentent les principales approches *data-level* de résolution du problème de déséquilibre des classes en apprentissage automatique. Elles décrivent en effet différentes méthodes de ré-échantillonnage des données déséquilibrées pour in fine pouvoir les analyser. Elles présentent des méthodes de sous-échantillonnage telles que Edited Nearest Neighbour (ENN), de sur-échantillonnage comme SMOTE, ainsi que des approches mixtes qui combinent techniques de sur et sous-échantillonnage (e.g., SMOTEEN).

Au niveau algorithmique, plusieurs méthodes sont également possibles pour la gestion du problème de classes déséquilibrées. Ritschard et al. (2017) [3] se concentrent, par exemple, sur la méthode des arbres de classification pour des données déséquilibrées. L'article porte principalement sur deux aspects : définir des critères de construction de l'arbre qui exploitent efficacement la nature déséquilibrée des données, et la pertinence de la conclusion à associer aux feuilles de l'arbre.

Dans sa thèse, Thomas (2009) [4] présente la méthode des forêts aléatoires comme une possibilité pour la gestion des données déséquilibrées. Il propose en effet de modifier trois étapes dans le processus d'apprentissage par forêts aléatoires.

La thèse de Desrousseaux (2022) [5] sur la détection de criminalité financière porte également sur une méthode d'analyse de données dans un contexte de données déséquilibrées. L'objectif de cette thèse est d'automatiser l'identification d'activités suspectes en réduisant le besoin d'intervention humaine, ainsi qu'en limitant le nombre de fausses alertes, à l'aide de réseaux de neurones.

Dans ce contexte, nous nous proposons de comparer différents modèles prédictifs inspirés des méthodes précédemment citées sur un jeu de données déséquilibrées (présenté en **Section 2.1**). Pour construire ces modèles, nous adopterons une approche hybride en utilisant des techniques de ré-échantillonnage

des données avec SMOTEEN, Tomek Link et Condensed Nearest Neighbor (CNN). Puis, nous lancerons des analyses de données nous permettant de gérer le déséquilibre des classes dans notre jeu de données. Toutes ces méthodes de ré-échantillonnage et d'analyses sont respectivement présentées dans les sections **Section 3.1** et **Section 3.2**. Pour finir, nous comparerons les différents modèles construits afin d'établir le plus pertinent pour notre cas à l'aide de la $F1$-mesure (**Section 4.2**).

2 Problématique

In fine l'objectif de ce projet est d'entraîner des modèles d'apprentissage automatique permettant la détection et la prédiction de fraude. Nous avons en effet à notre disposition un jeu de données de transactions par chèque frauduleuses et non frauduleuses. Les données sont présentées dans la section suivante, ainsi que quelques statistiques sommaires de ces données.

2.1 Notations

Soit un ensemble $S_k = (X, Y) = ((x_1, y_1), \dots (x_m, y_m))$ de m exemples d'entraînement où $x_i \in \mathbb{R}^d$ et $y_i \in \{0, 1\}^m$ sont les étiquettes correspondantes, et k le nombre d'ensembles — dans notre cas, $k = 2$.

La notation x_i^j est utilisée pour désigner la j^{ieme} valeur de l'exemple i. L'étiquette 0 désigne la classe majoritaire ou négative (i.e., non-fraude), et l'étiquette 1 désigne la classe minoritaire ou positive (i.e., fraude). Ainsi, S_+ désigne l'ensemble d'entraînement de m_+ exemples positifs, et S_- désigne l'ensemble d'entraînement de m_- exemples négatifs, avec m_+ bien inférieurs à m_-.

2.2 Présentation des Données

Les données sur lesquelles nous travaillons sont des données réelles. Elles sont issues d'une enseigne de la grande distribution, ainsi que de certains organismes bancaires (FNCI et Banque de France). Chaque ligne représente une transaction effectuée par chèque dans un magasin de l'enseigne quelque part en France. Les données ne sont pas brutes et plusieurs variables sont déjà des variables créées (i.e. elles sont issues du *feature engineering*). Nous disposons ainsi d'un ensemble de 23 variables qui ont la signification suivante :

- **ZIBZIN** : identifiant relatif à la personne (i.e., identifiant bancaire, relatif au chéquier en cours d'utilisation) ;
- **IDAvisAutorisAtionCheque** : identifiant de la transaction en cours ;
- **Montant** : montant de la transaction ;
- **DateTransaction** : date de la transaction ;
- **CodeDecision** : une variable qui peut prendre ici 4 valeurs
  - 0 : la transaction a été acceptée par le magasin,
  - 1 : la transaction, et donc le client, fait partie d'une liste blanche (i.e., bons payeurs),
  - 2 : le client fait partie d'une liste noire, son historique indique que c'est un mauvais payeur (des impayés en cours ou des incidents bancaires en cours), sa transaction est alors automatiquement refusée,
  - 3 : le client a été arrêté par le système par le passé pour une raison plus ou moins fondée ;
- **VérifianceCPT1** : nombre de transactions effectuées par le même identifiant bancaire au cours du même jour ;
- **VérifianceCPT2** : nombre de transactions effectuées par le même identifiant bancaire au cours des trois derniers jours ;
- **VérifianceCPT3** : nombre de transactions effectuées par le même identifiant bancaire au cours des sept derniers jours ;
- **D2CB** : durée de connaissance du client (par son identifiant bancaire), en jours — pour des contraintes légales, cette durée de connaissance ne peut excéder deux ans ;
- **ScoringFP1** : score d'anormalité du panier relatif à une première famille de produits (e.g., denrées alimentaires) ;
- **ScoringFP2** : score d'anormalité du panier relatif à une deuxième famille de produits (e.g., électronique) ;
- **ScoringFP3** : score d'anormalité du panier relatif à une troisième famille de produits (e.g., autres) ;
- **TauxImpNb_RB** : taux impayés enregistrés selon la région où a lieu la transaction ;
- **TauxImpNB_CPM** : taux d'impayés relatif au magasin où a lieu la transaction ;

— **EcartNumCheq** : différence entre les numéros de chèques ;
— **NbrMagasin3J** : nombre de magasins différents fréquentés les 3 derniers jours ;
— **DiffDateTr1** : écart (en jours) à la précédente transaction ;
— **DiffDateTr2** : écart (en jours) à l'avant dernière transaction ;
— **DiffDateTr3** : écart (en jours) à l'antépénultième transaction ;
— **CA3TRetMtt** : montant des dernières transactions + montant de la transaction en cours ;
— **CA3TR** : montant des trois dernières transactions ;
— **Heure** : heure de la transaction ;
— **FlagImpaye** : acception (0) ou refus de la transaction (1).

À noter que la variable "CodeDecision" n'est pas une variable à utiliser pour faire de la prédiction, car cette information est acquise post-transaction. On peut en revanche s'en servir lors de la phase d'apprentissage pour analyser les données par exemple. En outre, nous disposons d'un jeu de données comprenant 10 mois de transactions : du 1^{er} Février 2017 au 30 Novembre 2017.

2.3 Analyse et Préparation des Données

Après avoir chargé toutes les données du dataset, nous les avons inspectées (i.e., dimensions du dataset, type et nom de chaque variable). Le jeu de données est ainsi composé de 4,646,774 observations et de 23 variables.

Nous avons ensuite mis de côté les variables "CodeDecision" et "ZIBZIN", la première ne devant pas être utilisée pour la prédiction, et la seconde n'étant portante d'aucune information. La ligne n°1956361 a également été supprimée, car elle comportait le nom des variables.

Une majorité des variables restantes n'avaient pas le bon format. Les variables suivantes ont été transformées en $float$ en remplaçant les virgules par des points décimaux et en forçant ensuite leur typage :
— Montant
— ScoringFP1
— ScoringFP2
— ScoringFP3
— TauxImpNb_RB
— TauxImpNB_CPM
— DiffDateTr1
— DiffDateTr2
— DiffDateTr3
— CA3TRetMtt
— CA3TR.

Les suivantes ont simplement été forcées en $float$:
— IDAvisAutorisationCheque
— VerifianceCPT1
— VerifianceCPT2
— VerifianceCPT3
— EcartNumCheq
— NbrMagasin3J
— D2CB.

La variable 'DateTransaction' a été convertie au format date, et séparée en plusieurs variables : année, mois, jour, heure, minute, et seconde. Et la variable 'FlagImpaye' a été forcée en booléen, puisqu'elle ne peut prendre que deux valeurs : 0 si le chèque est accepté (non-fraude), ou 1 si le chèque est refusé (fraude).

Nous avons également établi la proportion de chaque classe dans le jeu de données. Ainsi, sur les 4,646,773 exemples, 62,622 sont positifs, soit 1.35% du jeu de données total (Figure 1).

Le degré de déséquilibre des données (appelé *Imbalance Ratio* en Anglais) est le suivant :

$$Imbalance\,Ratio = \frac{Exemples\,Majoritaires}{Exemples\,Minoritaires} = \frac{4,646,773}{62,622} = 73.20$$

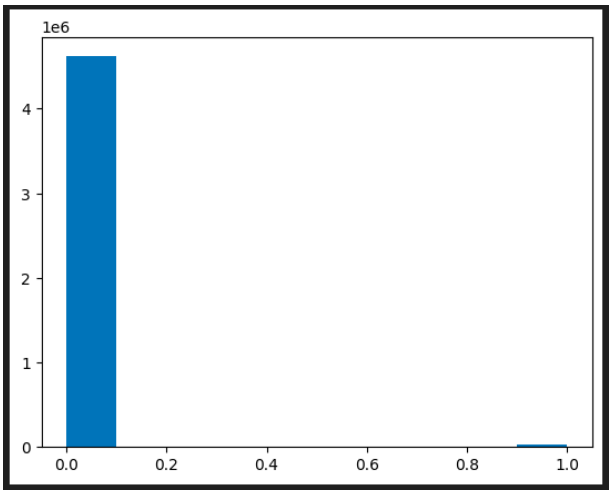

FIGURE 1 – Répartition des classes (non-fraude vs. fraude)

Ce qui signifie qu'il y a 73 fois plus d'exemples négatifs (i.e., appartenant à la classe majoritaire) que d'exemples positifs (i.e., appartenant à la classe minoritaire) dans notre jeu de données.

Après avoir inspecté et nettoyé les données, nous les avons séparées en échantillons d'apprentissage et de test, afin que toutes les analyses ultérieures ne soient pas biaisées par une dépendance dans les données entre ensembles d'entraînement et de test.

Pour se faire, nous avons d'abord trié les données selon la date de transaction ('DateTransaction'), puis nous avons supprimé cette variable déjà remplacée par six variables temporelles (présentées plus haut). Il était préalablement convenu que l'échantillon d'apprentissage soit constitué des données allant du 1^{er} Février au 31 Août 2017, et que le reste constitue l'échantillon test, soit 30% du jeu de données. Il a ensuite simplement fallu définir la variable "FlagImpaye" comme label, et le reste des variables comme prédicteurs.

3 Méthode

Nous rappelons que l'objectif principal de ce projet est d'entraîner des modèles capables de détecter les transactions frauduleuses et non frauduleuses faites par chèque. Par ailleurs, le fort déséquilibre des classes et la grosse volumétrie de nos données nous ont amené à préférer certaines méthodes de pré-traitement et de modèles d'apprentissage automatique plutôt que d'autres.

Nous avons en effet eu recours à SMOTEEN et Tomek Link pour ré-échantillonner les données de notre ensemble d'entraînement, afin de réduire le déséquilibre des classes dans nos données. Ces méthodes sont présentées dans la **section 3.1**. Nous avons ensuite lancé différents modèles d'entraînement : arbre de décision, forêts aléatoires, réseaux de neurones, modèles ensemblistes, et k-means. Ces modèles sont présentés dans la **section 3.2**.

3.1 Techniques de Ré-échantillonnage

Étant donné le fort déséquilibre des classes dans nos données entre la fraude et la non-fraude — rappelons qu'il y a 73 fois plus d'exemples négatifs (non-fraude) que d'exemples positifs (fraude) —, nous avons eu recours à plusieurs techniques de ré-échantillonnage afin de comparer les performances de chacune.

3.1.1 SMOTEEN

Pour des raisons liées à la rapidité des algorithmes à exécuter, nous avons choisi de lancer un SMOTEEN sur nos données. SMOTEEN est en effet une technique de ré-échantillonnage mixte disponible à partir de la librairie *imblearn* de Python. Elle combine l'approche de sur-échantillonnage de SMOTE (Synthetic Minority Over-sampling Technique) avec la technique de nettoyage de *Edited Nearest Neighbor* (ENN).

Classe	Exemples
Non-fraude	3,794,420
Fraude	21,925

TABLE 1 – Répartition des classes après SMOTEEN.

En effet, SMOTE vise à augmenter de manière synthétique les exemples de la classe minoritaire. Pour chaque exemple positif, SMOTE génère des exemples synthétiques dans son voisinage. Cela permet d'augmenter la taille de la classe minoritaire. À l'inverse, ENN consiste à supprimer les exemples de la classe majoritaire qui sont mal classés par leur voisinage. Cela permet d'éliminer les exemples de la classe majoritaire qui sont susceptibles d'introduire du bruit dans le modèle.

Algorithm 1 *SMOTE*

Require: Échantillon d'apprentissage S de taille $m = p + n$, k = nombre de voisins, R taux d'exemples positifs à générer
Ensure: Échantillonne S'
  Initialisation
  Poser $New = R * p$: nombre d'exemples à générer Syn, sélectionner aléatoirement New exemples parmi p et noter $setind$ leur indice
  for $i \in setind$ **do**
    Chercher les k-plus proche voisins positifs de p_i, sélectionner aléatoirement l'un des plus proches voisins
    rNN_i **for** `attributs` $attr$ de p_i **do**
    Choisir un nombre $\alpha \in [0, 1]$
    Poser $newattr = \alpha p_i[attr] + (1 - \alpha) rNN_i[attr]$
    Poser $Syn[attr] = newattr$
    Poser $S' = S \cup Syn_i$
  end for
  Poser $S' = S$
  Return S'

Algorithm 2 *Edited Nearest Neighbor*

Require: L'ensemble d'entraînement S, k = nombre de plus proches voisins
Ensure: L'ensemble sous-échantillonné S'
  Initialisation
  while S est modifié **do**
    for chaque exemple de la classe majoritaire **do**
      Déterminer les k-plus proches voisins de l'exemple
      Déterminer la classe majoritaire dans le k-voisinage de l'exemple
      Supprimer l'exemple de S si la prédiction ne coïncide pas avec le vrai label
    end for
  end while
  Poser $S' = S$
  return S'

Cela peut aider à améliorer la qualité des exemples synthétiques générés. Contrairement à certaines méthodes de sur-échantillonnage qui génèrent simplement des copies exactes d'instances existantes, SMOTEEN crée des exemples synthétiques en prenant en compte la relation entre les exemples existants. Cela peut réduire le risque de sur-ajustement en évitant la création de points synthétiques inutiles.

Le degré de déséquilibre des données après SMOTEEN est le suivant (voir Table 1 pour la répartition des classes après SMOTEEN) :

$$Imbalance\ Ratio = \frac{3,794,420}{21,925} = 173.06$$

On constate donc que cette technique de ré-échantillonnage n'a pas permis de réduire le déséquilibre

des classes. Au contraire, il l'a accentué : il y a 173 fois plus d'exemples négatifs que d'exemples positifs désormais. Pour tenter de pallier ce problème, nous avons fait le choix d'exécuter d'autres techniques de ré-échantillonnage sur les données.

3.1.2 Tomek Link

Étant donné la grosse volumétrie de données dont nous disposons, il est également apparu évident d'user de techniques de sous-échantillonnage afin de réduire le nombre de données de la classe majoritaire, tout en tentant de ne pas perdre d'informations dans les données ré-échantillonnées.

Tomek Link est donc une technique de sous-échantillonnage. Autrement dit, c'est une approche de ré-échantillonnage qui retire des exemples de la classe négative.

Cet algorithme se concentre sur les paires de données 'borderline' : lorsque dans la paire de données qui sont les plus proches voisines l'une de l'autre, un exemple appartient à la classe négative et l'autre à la classe positive, on considère cette paire comme un Tomek Link, et l'exemple de la classe négative est supprimé car potentiellement mal classifié par le modèle prédictif. Ainsi, le Tomek Link permet de supprimer les données négatives redondantes ou ambiguës, et stabilise ainsi les frontières entre les deux classes (i.e., fraude vs. non-fraude) de notre jeu de données.

Algorithm 3 *Tomek Link*

Require: L'ensemble d'entraînement S
Ensure: L'ensemble sous-échantillonné S' (plus petit que S)
  Initialisation
  Séparer S en deux ensembles aléatoires S_1 et S_2
  while S_1 et S_2 sont modifiés **do**
    Retirer de S_1 tous les exemples mal classifiés à l'aide de S_2 d'1-plus proche voisin
    Retirer de S_2 tous les exemples mal classifiés à l'aide de S_1 d'1-plus proche besoin
  end while
  $S' = S_1 \cup S_2$
  return S'

Tomek Link est également disponible à partir de la librairie *imblearn* de Python. Nous l'avons paramétré comme suit :

```
from imblearn.under_sampling import TomekLinks

tomek = TomekLinks(sampling_strategy='majority', n_jobs=8)
```

Ces paramètres permettent de ne ré-échantillonner que la classe majoritaire, et de paralléliser le calcul en utilisant 8 processeurs.

Cependant, bien qu'ayant réglé l'algorithme pour qu'il ne ré-échantillonne que les exemples de la classe négative, certains exemples positifs disparaissent dans le processus — car probablement mal classés par leurs voisins. Ainsi, Tomek Link échoue à réduire le déséquilibre des classes dans nos données. Pour la suite des analyses, nous avons donc laissé tomber cette technique. Nous ne l'avons utilisée que pour les modèles k-means, de régression logistique et d'auto-encodeur. Les performances sont semblables à celles générées par SMOTEEN.

3.2 Modèles et Entraînement

Précédemment, nous avons vu que plusieurs méthodes étaient possibles pour traiter le problème de déséquilibre des classes au niveau des données. Nous allons maintenant voir que cela est également possible au niveau algorithmique.

Nous avons donc choisi de tester sept modèles d'apprentissage automatique. Ces modèles diffèrent les uns des autres par leurs processus sous-jacents. Les modèles choisis sont les suivants : arbre de décision, forêts aléatoires, réseaux de neurones (simples et auto-encodeur), modèles ensemblistes avec XGBoost et forêts aléatoires équilibrées, et k-means. Ces algorithmes sont tous décrits dans les sections suivantes.

3.2.1 Arbres de Décision

Pour ce qui est de l'arbre de décision, au vu de la rapidité d'apprentissage du modèle, nous avons choisi d'optimiser le modèle en utilisant une recherche sur grilles. Les paramètres de la recherche sont les suivants :

```
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
    }
```

Comme nous pouvons le voir, les paramètres optimisés sont le critère (`'criterion'`), la profondeur maximale, l'ensemble minimum pour la séparation, et l'ensemble minimum par feuille.

Nous nous sommes concentrés sur les paramètres permettant d'optimiser la $F1$-mesure.

3.2.2 Forêts Aléatoires

Concernant le modèle de forêt aléatoire (ou *Random Forest* en Anglais), les analyses étaient très longues — de l'ordre d'une journée. Nous avons donc choisi arbitrairement les paramètres du modèle — comme suit :

```
rf_classifier = RandomForestClassifier(random_state=42,
                                       n_estimators=200,
                                       max_depth=20,
                                       min_samples_leaf=8,
                                       min_samples_split=10)
```

3.2.3 Réseaux de Neurones

3.2.3.1 Réseaux de Neurones Simples

Pour le modèle de réseaux de neurones, nous avons choisi le modèle suivant :

```
inputs=Input((X_train.shape[1]))

d1 = Dense(16, activation='relu')(inputs)
d1 = Dropout(0.3)(d1)

outputs=Dense(1, activation='sigmoid', bias_initializer=output_bias)(d1)

model = Model(inputs=[inputs], outputs=[outputs])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

model.summary()
```

Comme nous pouvons le voir, il s'agit d'un réseau simple composé de deux couches entièrement connectées (*full connected*). La première couche est une couche à seize neurones et une fonction d'activation "relu". Cette couche est directement connectée à une couche de sortie à un neurone, avec comme fonction d'activation la fonction sigmoïde. De plus, pour la fonction de sortie, nous avons initialisé le biais afin d'essayer de contre-balancer le déséquilibre des classes. Pour l'entraînement du modèle, nous avons choisi comme optimiseur `'adam'`, et comme fonction de perte la `'binary_crossentropy'`.

3.2.3.2 Auto-encodeur

L'auto-encodeur est un réseau de neurones pour l'apprentissage automatique non supervisé, souvent utilisé dans la reconstruction des données d'entrée. En effet, l'auto-encodeur dispose d'une partie appelée 'encodeur' et d'une partie 'décodeur'. C'est un réseau de neurones qui repose sur une fonction perte de reconstruction (appelée *reconstruction loss* en Anglais).

La première partie de l'auto-encodeur vise à construire une représentation de dimensions réduites de l'entrée. Pour se faire, elle extrait les caractéristiques les plus importantes des données, ce qui donne 'l'espace latent' de l'auto-encodeur. La seconde partie de l'auto-encodeur, le 'décodeur', vise à reconstituer les données d'entrée à partir de la représentation compressée construite par l'encodeur.

Cette méthode paraissait intéressante à lancer pour nos données. En effet, l'auto-encodeur récupère les principales caractéristiques des données. Autrement dit, il peut récupérer les 24 prédicteurs de notre jeu de données afin de détecter les fraudes.

L'auto-encodeur que nous avons lancé est disponible à partir de de l'API Keras de la librairie TensorFlow de Python. Nous l'avons paramétré comme suit :

```
input_dim = X_train.shape[1]  # nombre de caractéristiques dans nos données

input_layer = Input(shape=(input_dim))
encoded = Dense(16, activation='relu')(input_layer)  # couche d'encodage
decoded = Dense(input_dim, activation='sigmoid')(encoded)  # couche de décodage

autoencoder = Model(input_layer, decoded)

autoencoder.compile(optimizer='adam',
loss='binary_crossentropy')

with tf.device('/device:GPU:0'): # utiliser le GPU pour les calculs
    history = autoencoder.fit(X_train, X_train, epochs=10, batch_size=128, shuffle=True)
```

de sorte que la couche d'entrée et de sortie possèdent le même nombre de neurones (i.e., 24) — un par caractéristique —, que la couche d'encodage soit constituée de 16 couches avec une fonction d'activion ReLU, que la couche de décodage possède une fonction d'activion sigmoïde, que l'optimisation se fasse par `'adam'`, et que la fonction perte de reconstruction soit adaptée pour de la classe binaire (cf. `'binary_crossentropy'`).

De plus, nous avons fait en sorte de distribuer le calcul sur le GPU de la machine afin de réduire le temps d'exécution. Nous défini le nombre d'epochs à 10, et la taille du batch à 128 pour une grosse volumétrie de données.

Néanmoins, nous étions également conscients que les auto-encodeurs peuvent être limités dans le cas des classes déséquilibrées, car les caractéristiques non familières peuvent être interprétées comme des anomalies par l'algorithme, et donc ne pas être reconstruites.

3.2.4 Modèle Ensembliste

Les méthodes ensemblistes visent à combiner les prédictions de plusieurs modèles individuels faibles pour améliorer la robustesse et la performance globale du modèle. En combinant les forces de plusieurs modèles k-Nearest Neighbor, une méthode ensembliste peut améliorer la généralisation du modèle, ce qui est crucial dans les problèmes de déséquilibre de classe où la classe minoritaire peut être sous-représentée dans l'ensemble de données d'entraînement.

Ainsi, nous avons décidés d'utiliser un ensemble de 3 modèles k-Nearest Neighbor le voici :

```
models = list()
models.append(('knn1', KNeighborsClassifier(n_neighbors=1)))
models.append(('knn3', KNeighborsClassifier(n_neighbors=3)))
models.append(('knn5', KNeighborsClassifier(n_neighbors=5)))

ensemble = VotingClassifier(estimators=models, voting='hard')
```

Cet ensemble de domdèles combine les prédictions de plusieurs modèles K-NN avec différents paramètres pour prendre une décision de classification basée sur la majorité des votes. Chaque modèle a un poids égal dans le vote ('hard' voting).

3.2.5 XGBoost

XGBoost utilise un ensemble de modèles faibles, généralement des arbres de décision, qui sont combinés pour former un modèle robuste. XGBoost est capable de gérer automatiquement les ensembles de données déséquilibrés. Il ajuste les poids des exemples des classes majoritaires et minoritaires, ce qui permet de donner plus d'importance aux exemples de la classe minoritaire pendant l'entraînement.

Au vu de la rapidité d'apprentissage du modèle, nous avons choisi d'optimiser le modèle en utilisant une recherche sur grilles. Les paramètres de la recherche sont les suivants :

```
model = XGBClassifier()

param_grid = {
    'learning_rate': [0.01, 0.1, 1],
    'gamma': [1, 3, 5],
    'alpha': [0.01, 0.1, 1],
    'colsample_bytree': [0.8, 0.9, 1]
}

grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=5, scoring='f1')
```

3.2.6 Forêts Aléatoires Équilibrées

En voyant l'efficacité de l'algorithme XGBoost, nous avons décidé de nous pencher sur un autre algorithme ensembliste basé sur le boosting.

Le "Boosting Équilibré" (ou *Balanced Boost* en Anglais) consiste à faire une modification de RUSBoost, où au lieu d'appliquer RUS, il considère le sous-échantillonnage de la classe majoritaire et le sur-échantillonnage de la classe minoritaire de telle sorte que le nouvel ensemble de données soit équilibré et a le même nombre d'instances des classes majoritaires et minoritaires. De plus, le processus d'échantillonnages est effectué suivant les poids de l'algorithme DataBoost.M2.

Comme pour l'algorithme XGBoost, la rapidité d'apprentissage du modèle permet d'optimiser le modèle en utilisant une recherche sur grilles. Les paramètres de la recherche sont les suivants :

```
param_grid = {
    'learning_rate': [0.01, 0.1, 1],
    'gamma': [None, 1, 3, 5],
    'alpha': [0.01, 0.1, 1],
    'colsample_bytree': [0.8, 0.9, 1]
}

brf_model = BalancedRandomForestClassifier(sampling_strategy="majority", replacement=True, random

grid_search = GridSearchCV(estimator=brf_model, param_grid=param_grid, scoring='f1', cv=5)
```

3.2.7 *K*-Means

K-means est un des algorithmes de clustering les plus couramment utilisés en apprentissage automatique non-supervisé. Il permet d'identifier et d'interpréter la structure de données non étiquetées.

Il repose en effet sur un principe de partitionnement des données en plusieurs clusters. L'objectif est de minimiser l'inertie intra-groupe (i.e., l'écart de chaque donnée par rapport à la moyenne du cluster — aussi appelée centroïde — auquel elle appartient). Pour cela, il est nécessaire de trouver le nombre optimal de clusters afin qu'à l'intérieur d'un cluster, les exemples soient suffisamment similaires entre eux, et que d'un cluster à un autre, les exemples soient suffisamment dissimilaires.

Ainsi, étant donné un ensemble de m exemples (x_1, x_2, ..., x_m), on veut diviser ces m exemples en k groupes $S = S_1, S_2, \ldots, S_k (k \leq n)$ en minimisant la distance entre les points à l'intérieur de chaque groupe :

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x_j} \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2 \quad (1)$$

avec μ_i, le centroïde des exemples dans l'ensemble S_i.

Nous savons, néanmoins, que k-means peut s'avérer limité dans un contexte de données déséquilibrées et de données de grandes dimensions, qui sont les deux cas dans lesquels nous nous trouvons. Il tend en effet à tirer tous les centroïdes vers la moyenne de la classe négative. Malgré cela, nous avons trouvé intéressant de lancer un k-means sur nos données, et ce, bien qu'elles soient étiquetées.

K-means est disponible à partir de la librairie *scikit-learn* de Python. Nous l'avons paramétré comme suit :

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2, random_state=1)
```

3.2.8 Régression Logistique

La régression est utilisée dans le cas de classifications binaires. Elle repose sur une fonction simoïde — logistique — pour transformer une combinaison linéaire des caractéristiques d'entrée en une probabilité. Cette fonction définie comme suit : $\sigma(z) = \frac{1}{1+e^{-1}}$, avec z, la combinaison linéaire des caractéristiques ($z = b_0 + b_1x_1 + b_2x_2 + \cdots + b_nx_n$), et $\sigma(z)$, la probabilité estimée que l'événement positif survienne, notée $P(Y = 1)$. Ainsi, l'équation du modèle de régression logistique est donnée par : $P(Y = 1) = \sigma(b_0 + b_1x_1 + b_2x_2 + \cdots + b_nx_n)$.

La régression logistique est de fait utilisée dans divers domaines. Elle est par exemple utilisée en sciences médicales pour la modélisation de la probabilité de présence d'une maladie. Par analogie, nous avons trouvé intéressant de lancer une régression logistique sur nos données.

Elle est disponible à partir de la librairie scikit-learn de Python. Aucune optmisation ou paramétrage paticuliers n'ont été effectués pour ce modèle.

4 Expériences

Dans cette section, nous évaluons les performances de chacun de nos modèles. Nous comparons les résultats de chaque modèle sur leurs capacités à détecter et prédire les fraudes de notre jeu de données.

4.1 Données

Notre jeu de données consiste en 10 mois de transactions par chèque. Les sept premiers mois ont été utilisés comme ensemble d'entraînement (3,252,741 transactions), et les trois mois restants comme ensemble de test (1,394,032 transactions). Les données sont constituées de 24 prédicteurs (après séparation de la variable 'DateTransaction') et étiquetées comme fraude ou non-fraude. La classe positive représente 1.56% des données avec 50,682 exemples. Le degré de déséquilibre de l'ensemble d'entraînement s'élève ainsi à $IR = 63.18$.

4.2 Résultats

Dans le contexte d'un déséquilibre des classes, nous ne pouvons pas nous servir de l'accuracy comme métrique d'évaluation de nos modèles. En effet comme précisé plus haut, l'accuracy donne autant de poids à chaque classe sans tenir compte de la proportion de chacune d'elles. L'accuracy fait ainsi fi de la sur/sous-représentation d'une classe par rapport à une autre.

Pour pallier ce problème, nous avons recours à la $F1$-mesure. C'est une métrique plus appropriée pour les problèmes de classes déséquilibrées : elle permet en effet de donner plus ou moins d'importance aux classes, et est un parfait compromis entre la précision (*precision*) et le rappel (*recall*).

La $F1$-mesure permet de fait d'évaluer la capacité d'un modèle à bien prédire les individus positifs, tant en termes de prédictions positives correctes (cf. précision), qu'en termes de positifs correctement prédits (cf. rappel). Elle est définie comme la moyenne harmonique de la précison et du rappel :

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

et peut également s'écrire :

$$F1 = \frac{Vrais\,Positifs}{Vrais\,Positifs + \frac{1}{2}(Faux\,Negatifs + Faux\,Positifs)}$$

La $F1$-mesure permet ainsi une comparaison entre les prédictions positives correctes (i.e., vrais positifs) et les erreurs faites par le modèle.

4.2.1 Comparaison de Modèles

Nous présentons nos $F1$-mesures dans les tables 2 et 3. Nous pouvons ainsi voir que les performances de nos modèles sont très limitées au vu du problème des classes déséquilibrées.

Modèle	$F1$**-Mesure**
Arbres de Décision	0.001
Forêts Aléatoires	0.0003
Réseaux de Neurones Simples	0.0
Auto-encodeur	0.017
XGBoost	0.006
Forêts Aléatoires Équilibrées	0.056
Modèle ensembliste	0.0
K-means	0.000
Régression Logistique	0.0

TABLE 2 – Les scores de la $F1$-mesure pour chaque modèle lancé à partir du SMOTEEN

Modèle	$F1$**-Mesure**
Auto-encodeur	0.017
K-means	0.000
Régression Logistique	0.0

TABLE 3 – Les scores de la $F1$-mesure pour chaque modèle lancé à partir du Tomek Link

Il est important de noter que la $F1$-mesure que nous avons utilisée renvoie 0.0 si notre modèle renvoie la même classe pour l'ensemble des prédictions. C'est par exemple le cas du réseau de neurones qui renvoie `'False'` pour toutes les prédictions.

4.2.2 Apprentissage Sensible aux Coûts

L'utilisation de la $F1$-mesure pour l'évaluation des modèles n'est peut-être pas le plus pertinent. En effet, dans ce genre d'entreprises, les dirigeants résonnent souvent en termes de gains et pertes. De plus, la $F1$-mesure ne prend pas en compte le poids du montant de la perte et/ou du gain dans l'évaluation. C'est pourquoi nous pouvons trouver une fonction prenant ce poids en considération pour entraîner et évaluer nos modèles.

Cette fonction est considéré comme ceci : elle dépend du montant de la transaction, noté m, et d'un poids p défini à 0.05.

Marges :
- Si une transaction est acceptée et n'est pas frauduleuse, alors l'enseigne génère une marge de m * p.
- Si une transaction est refusée mais qu'elle n'était pas frauduleuse, alors l'enseigne génère un marge de 0.7 * M * p.

— Si une transaction est refusée et qu'elle était frauduleuse, alors l'enseigne ne perds pas d'argent. Néanmoins pour notre fonction nous avons décider de l'ajouter comme une marge de m * p.

Pertes :

— Si une transaction est acceptée et qu'elle est frauduleuse, alors l'enseigne génère une perte dépendant de son montant m.
  — perte de 0 si m <= 20
  — perte de 0.2 * m si m <= 50
  — perte de 0.3 * m si m <= 100
  — perte de 0.5 * m si m <= 200
  — perte de 0.8 * m si m > 200

Avec cette fonction, nous avons à nouveau évalué nos modèles et nous avons obtenu les résultats suivants :

Modèle	**Gain**
Arbres de Décision	1943459.56
Forêts Aléatoires	1941884.88
XGBoost	1944092.83
Forêts Aléatoires Équilibrées	1993271.16
K-means	4343895.03
Régression Logistique	3659124.15

TABLE 4 – Résultat de la fonction de perte sensible au coût

Nous avons donc ensuite essayé de créer une fonction permettant d'optimiser ce gain en fonction des prédictions du modèle. La fonction est décrite dans l'algorithme 4.

Algorithm 4 fonction_perte

```
function FONCTION_PERTE(X_test, mod = None)
    poids ← 0
    pp ← mod.predict_proba(X_test)
    tab ← []
    pred ← 0
    for i ← 0 to len(X_test) − 1 do
        predict_prob0 ← pp[i, 0]
        predict_prob1 ← pp[i, 1]
        M ← X_test.iloc[i]['Montant']
        TN ← 0.05 × M × predict_prob0
        FP ← 0.70 × M × 0.05 × predict_prob1
        if M ≤ 20 then
            FN ← 0
        else if M ≤ 50 then
            FN ← 0.2 × M
        else if M ≤ 100 then
            FN ← 0.3 × M
        else if M ≤ 200 then
            FN ← 0.5 × M
        else
            FN ← 0.8 × M
        end if
        TP ← predict_prob1 × poids × M
        rt ← TN + FP − FN + TP
        if rt ≤ 0 then
            pred ← 1
        else
            pred ← 0
        end if
        tab.append([rt, pred])
    end for
    return tab
end function
```

Après avoir testé cette fonction sur nos modèles, il en ressort que la fonction ne maximise pas les gains. En revanche, elle minimise les pertes, puisqu'elle refuse un grand nombre de transactions en fonction des probabilités de sortie du modèle.

5 Conclusion

Les performances de nos modèles restent très limitées. Notre meilleure $F1$-mesure est aux environ de 0.06. Les modèles utilisés sont pourtant des modèles très performants habituellement pour certains. Par ailleurs, nos modèles auraient nécessité plus d'optimisation. Par exmeple, les forêts aléatoires peuvent s'avérer très performantes avec une optimisation de recherche sur grille. De même, les réseaux de neurones requièrent une fonction de perte customisée (*custom loss*) spécifique dans le cas de classes très fortement déséquilibrées.

Ainsi, le problème des classes déséquilibrées est un réel challenge pour l'apprentissage automatique. Si l'on s'accorde sur une approche hybride (i.e., techniques de ré-échantillonnage combinées à des algorithmes capables de prendre en compte le déséquilibre des classes, voire le poids de ces dernières), les méthodes dépendent toujours de la particularité du cas dans lequel nous nous trouvons.

De plus, la grosse volumétrie de données dont nous disposons peuvent limiter l'utilisation de certains algorithmes plutôt que d'autres (e.g., SVM non adéquats pour les données volumineuses). De même, si nous voulions parfaire l'optimisation de nos paramètres, il nous faudrait des machines plus performantes.

Références

[1] Haixiang, G., Yijing, L., Shang, J., Mingyun, G., Yuanyue, H., Bing, G. : "Learning from class-imbalanced data : review of methods and applications", Expert Systems With Applications, vol. 73, pp. 220–239 (2017)

[2] Remy, E., Verges, V., Dautreme, E., Martin-Cabanas, B. : Revue des principales approches de résolution du problème de déséquilibre des classes, Congrès Lambda Mu 22 "Les risques au cœur des transitions" (e-congrès) - 22e Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement, Institut pour la Maîtrise des Risques, Oct 2020, Le Havre (e-congrès), France (2021)

[3] Ritschard, G., Marcellin, S., Djamel A. Zighed, D.A. : Chapitre 7 : Arbre de décision pour données déséquilibrées : Sur la complémentarité de l'intensité d'implication et de l'entropie décentrée (2017)

[4] Thomas, J. : Apprentissage supervisé de données déséquilibrées par forêt aléatoire [Thèse] (2009)

[5] Desrousseaux, R. : Détection de criminalité financière par réseaux de neurones [Thèse] (2022)