INF7370 - TP1

Guillaume Carignan (CARG29099504), François Huppé-Marcoux (HUPF10049509)

3 mars 2025

1 Tâche 1 : Compréhension de l'algorithme GBOOST

1.1 GBoost : introduction

L'algorithme GBoost est une méthode d'apprentissage automatique utilisée pour améliorer la performance des modèles de prédiction, notamment les arbres de décision. Il appartient à la famille des algorithmes d'ensemble et repose sur l'idée d'entraîner plusieurs modèles de manière séquentielle afin de réduire progressivement l'erreur. Contrairement à une forêt aléatoire, qui entraîne des arbres en parallèle et agrège leurs résultats, GBoost ajuste chaque nouvel arbre en fonction des erreurs des arbres précédents pour corriger leurs prédictions.

Les arbres de décision seuls présentent plusieurs limites. Ils sont très sensibles aux données d'apprentissage et peuvent soit sous-ajuster, en ne capturant pas suffisamment les tendances, soit sur-ajuster, en s'adaptant trop aux spécificités des données et en perdant leur capacité de généralisation. De plus, leur performance est fortement influencée par la structure des données et par la qualité de l'étiquetage. Dans un contexte où l'on dispose d'un volume suffisant de données bien structurées et où l'objectif est d'atteindre une haute précision, GBoost permet d'améliorer un arbre de décision en l'intégrant dans un processus itératif de correction des erreurs.

1.2 Principe de fonctionnement

GBoost construit une série d'arbres de décision de manière séquentielle, où chaque nouvel arbre est entraîné pour minimiser l'erreur des prédictions accumulées. L'idée centrale est que chaque nouvel arbre vient corriger les erreurs du modèle précédent en se basant sur la direction qui permet de réduire la fonction de perte.

1.2.1 Données

L'algorithme prend en entrée des données X et produit en sortie une prédiction Y. Il existe deux types de prédiction : une prédiction de classe (un nombre représentant une catégorie) ou une prédiction par régression (un nombre continu représentant une quantité). L'algorithme GBoost est implémenté pour les deux types avec les bibliothèques `GradientBoostingClassifier` ou `GradientBoostingRegressor`.

Les données en entrée du modèle sont des caractéristiques (features) provenant d'un ensemble de données. Par exemple, pour la classification d'utilisateurs « polluants » ou « légitimes », on pourrait avoir : `nombre_de_tweets`, `similarité_tweets`, `fréquence_tweets`. La classe en sortie, dans ce cas, correspond à une classification : `0` pour un utilisateur légitime et `1` pour un utilisateur polluant. Par exemple, supposons cet ensemble de données :

nombre_de_tweets	**similarité_tweets**	**fréquence_tweets**	**classe**
1087	3.10	2	1
1200	0.1	1.25	0
1000	1.6	0.05	0
890	1.5	0.9	1
500	2.5	1.2	0
950	1.0	0.7	1

Table 1: Exemple d'ensemble de données.

Note : La `similarité_tweets` est une cote z de la similarité des tweets d'un utilisateur comparée aux autres utilisateurs. Une cote z faible signifie que les tweets de cet utilisateur se ressemblent peu d'un tweet à l'autre. Une cote élevée signifie que l'utilisateur tweete des messages très semblables.

1.2.2 Construction de l'arbre de décision

Dans la première passe de l'algorithme, on construit un premier arbre de décision.

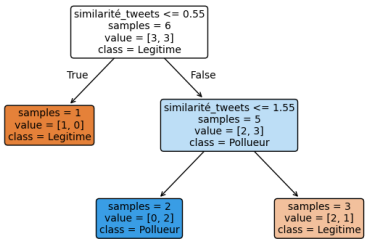

Figure 1: Premier arbre de décision (exemple).

Dans cet arbre, nous voyons qu'il y a une donnée mal classée. Dans un vrai scénario, le nombre de données mal classées serait beaucoup plus grand.

nombre_de_tweets	similarité_tweets	fréquence_tweets	classe	Prédiction
1087	**3.10**	**2**	**1**	**0**
1200	0.1	1.25	0	0
1000	1.6	0.05	0	0
890	1.5	0.9	1	1
500	2.5	1.2	0	0
950	1.0	0.7	1	1

Table 2: Exemple de prédiction erronée pour une instance.

1.2.3 Correction de l'erreur

Il existe deux équations pour calculer l'erreur : l'une pour la régression et l'autre pour la classification.
NOTE : Pour la régression, on utilise :

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{pour } i = 1, \ldots, n.$$

- n : Nombre total d'exemples dans le jeu de données.
- m : L'arbre de décision.
- i : L'exemple de donnée.
- $F(x_i)$: La prédiction $F()$ de l'arbre pour les valeurs x de la ligne i.
- y_i : La vraie classe de l'exemple i (0 ou 1).
- $\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}$: La pente du gradient (varie selon la fonction de perte).

Cette perte est conçue pour des valeurs continues, mais comme l'exemple utilise une classification, nous ne pouvons pas l'utiliser dans ce cas.

Pour calculer l'erreur pour une classification, on utilise la fonction de perte *log-loss* suivante (la perte utilisée par défaut par `sklearn`) :

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right].$$

- n : Nombre total d'exemples dans le jeu de données.
- y_i : La vraie classe de l'exemple i (0 ou 1).

- $\hat{y}_i$: La probabilité prédite pour que l'exemple i appartienne à la classe 1 (généralement sortie d'une fonction sigmoïde).

Pour cet exemple, par souci de simplicité des calculs, on simplifie la perte en :

$$r_{im} = y_i - \hat{p}_i$$

où :

- y_i : La vraie classe de l'exemple i (0 ou 1).
- $\hat{p}_i$: La probabilité prédite pour la classe 1.

NOTE : Cette perte simplifiée est utile pour illustrer l'algorithme, mais en pratique, la *log-loss* pénalise plus sévèrement les erreurs et mène à une convergence plus rapide.
Pour calculer l'erreur résiduelle de la première ligne de notre ensemble de données, il faut connaître $\hat{p}_1$, dont la formule est :

$$\hat{p}_i = \frac{\text{nombre d'observations de classe réel dans la feuille}}{\text{nombre total d'observations dans la feuille}}.$$

nombre_de_tweets	similarité_tweets	fréquence_tweets	classe	Prédiction
1087	**3.10**	**2**	**1**	**0**

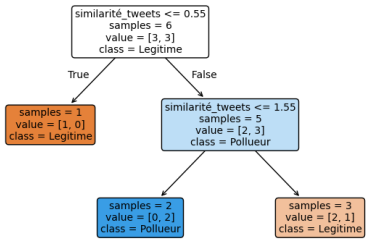

Figure 2: Chemin de décision pour la donnée mal classée.

En partant du sommet :

- `similarité_tweets <= 0.55 == Faux` → on va à gauche
- `similarité_tweets <= 1.55 == Faux` → on va à gauche

On tombe dans la feuille : `sample = 3, value = [2,1], class = legitime`. La `value` représente le nombre de données de la classe 0 (légitime) et 1 (pollueur). Ici, $[2, 1]$ signifie qu'il y a deux exemples légitimes et un exemple pollueur classé dans cette feuille. Donc, il y a un utilisateur pollueur mal classé dans une feuille légitime.

$$\hat{p}_1 = \frac{1}{3} = 0.33333$$

Pour calculer l'erreur pour la première ligne (i) du premier arbre (m), on fait :

$$r_{1,1} = y_1 - \hat{p}_1 = 1 - 0.33333 = 0.66667.$$

Cette valeur de perte est appelée **pseudo-erreur résiduelle**, car il s'agit d'une pseudo-erreur et non pas d'une dérivée.

Ensuite, on effectue cette opération pour toutes les lignes de l'ensemble de données, ce qui donne par exemple : `Code (erreurs_residuelles.py)`.

nombre_de_tweets	similarité_tweets	fréquence_tweets	classe	Prédiction	pseudo-erreur résiduelle
1087	**3.10**	**2**	**1**	**0**	**0.666667**
1200	0.1	1.25	0	0	0
1000	1.6	0.05	0	0	-0.333333
890	1.5	0.9	1	1	0
500	2.5	1.2	0	0	-0.333333
950	1.0	0.7	1	1	0

Table 3: Exemple de calcul des pseudo-erreurs résiduelles.

1.2.4 Deuxième arbre

Une fois que l'erreur a été calculée pour le premier arbre, la prochaine étape dans l'algorithme GBoost consiste à construire un nouvel arbre dont le but n'est pas de prédire la classe comme le premier arbre, mais plutôt de prédire la **pseudo-erreur résiduelle** du premier arbre.

NOTE : L'entraînement d'arbres consécutifs est aussi appelé *weak learner*, et l'addition de ces arbres pour classifier une donnée est appelée *strong learner*. Le terme *pseudo* provient du fait qu'on n'utilise pas la formule de calcul de la pente du gradient (première dérivée).

Le principe est de refaire les étapes de construction de l'arbre de décision avec les mêmes X en entrée, mais en utilisant les erreurs résiduelles comme Y.

1.2.5 Mise à jour du modèle

Une fois que le deuxième *weak learner* est entraîné, on applique une formule pour que chaque arbre de décision soit pris en compte dans le but de créer un ensemble de *weak learners* qui contribuent à la classification finale. Le principe est de diminuer l'erreur résiduelle à chaque arbre afin de converger vers une solution. On applique un poids à chacun des arbres de décision. Ce poids, aussi appelé *taux d'apprentissage* (*learning rate*), permet de converger lentement vers une solution. Un taux d'apprentissage trop élevé peut causer un surajustement au bruit dans les données.

L'ensemble du modèle est alors mis à jour en ajoutant une version pondérée du nouvel arbre à la prédiction précédente :

$$F_m(x) = F_{m-1}(x) + \nu \cdot h_m(x),$$

où $h_m(x)$ est la prédiction du nouvel arbre et ν est le taux d'apprentissage (learning rate) qui contrôle la contribution de chaque nouvel arbre.
Par exemple, si on prend un cas hypothétique où l'on définit $\nu = 0.1$ et que l'on a 100 arbres de décision qui prédisent tous la classe 1 :

$$F_{99}(x) = 0.1 \times 100 = 10.$$

Pour connaître la prédiction finale de l'ensemble des arbres, on applique une fonction sigmoïde :

$$p(x) = \frac{1}{1 + \exp(-10)} \approx 0.99995.$$

Enfin, on applique un seuil à 0.5 pour déterminer la classe. Dans ce cas-ci, $p(x) \approx 0.99995 \geq 0.5$, donc la classe prédite est `1`. Cela fait beaucoup de sens puisque les 100 arbres ont prédit la classe `1`.

La fonction sigmoïde a pour but de normaliser en pourcentage et de séparer les données de manière non linéaire.

1.2.6 Condition d'arrêt

Il existe plusieurs types de conditions d'arrêt.

Nombre d'itérations : La plus simple consiste à fixer un nombre d'itérations. Plus on ajoute d'arbres de décision, plus l'algorithme prendra du temps à s'exécuter. Dans un cas où l'on dispose de ressources limitées pour la classification, on peut choisir un nombre maximal d'arbres, même si l'on n'a pas nécessairement atteint la meilleure performance.

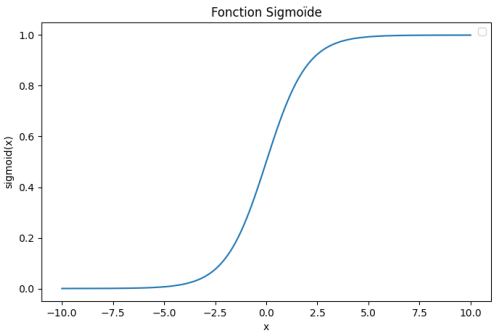

Figure 3: Fonction sigmoïde pour la transformation des scores en probabilités.

Amélioration de la perte : Au-delà d'un certain point, les gains obtenus en créant de nouveaux arbres de décision sont minimes. Le ratio temps d'exécution / gain en performance ne justifie plus de continuer.

Meilleure performance : On peut également attendre que la performance atteigne son maximum.

Les conditions d'arrêt et d'autres hyper-paramètres tels que le taux d'apprentissage, la profondeur des arbres, le nombre d'arbres, etc. peuvent être recherchés en faisant une validation croisée. La validation permet de tester différents hyper-paramètres sur différentes parties des données.

1.3 GBoost résumé

En somme, on construit plusieurs arbres de décision qui contribuent chacun à une fraction de la décision finale. Les arbres qui suivent le premier prédisent la pseudo-erreur résiduelle et tentent de la corriger avec un certain taux, réduisant ainsi progressivement l'erreur de l'arbre précédent. Ce processus itératif assure une convergence vers un minimum.

1.4 Forces et avantages

L'algorithme GBoost offre plusieurs avantages. Tout d'abord, il permet d'obtenir des performances élevées en ajustant finement les erreurs du modèle à chaque itération, ce qui améliore la précision par rapport à un simple arbre de décision ou à une forêt aléatoire. Ensuite, il est flexible, car il peut être appliqué à divers types de problèmes en ajustant la fonction de perte et les hyperparamètres, tels que la profondeur des arbres, le taux d'apprentissage et le nombre d'arbres.

Toutefois, cette approche présente des inconvénients. L'entraînement des arbres étant séquentiel, GBoost est plus lent que des méthodes parallélisables comme la forêt aléatoire. L'apprentissage dépend des itérations précédentes, ce qui limite la performance comparée à d'autres algorithmes.

GBoost peut également être sujet au surajustement si le nombre d'arbres est trop élevé ou si le modèle apprend trop fidèlement les données d'entraînement. Pour éviter cela, il est recommandé d'utiliser un taux d'apprentissage faible, associé à un nombre plus important d'arbres, ce qui permet une correction plus progressive des erreurs et une meilleure généralisation au coût d'un plus grand nombre d'itérations (donc plus de temps et de ressources pour classer une nouvelle donnée).

Les résultats de l'algorithme sont sensibles aux hyperparamètres. Le nombre d'itérations, le taux d'apprentissage et la taille des arbres sont autant de paramètres qui donneront des performances plus ou moins bonnes selon le type de données en entrée. Il peut être difficile de trouver une solution optimale dans l'espace d'hyperparamètres.

Des variantes comme **XGBoost** introduisent des techniques de régularisation supplémentaires pour améliorer la robustesse et la rapidité d'entraînement.

1.5 Conclusion

En comparaison avec d'autres méthodes d'ensemble, GBoost est particulièrement efficace lorsque les données d'entrée sont bien structurées et que la précision est un critère important. Son interprétabilité est également un avantage, car il est possible d'examiner l'importance des différentes variables et la contribution de chaque arbre aux prédictions finales. GBoost est un algorithme qui améliore la précision des modèles prédictifs en corrigeant itérativement les erreurs des arbres de décision, autant pour la classification que pour la régression, mais il nécessite un réglage minutieux pour éviter le surajustement et pour limiter le temps d'entraînement.

2 Préparation des données

La préparation des données s'effectue dans le fichier $features\_extraction.py$.

2.1 Caractéristiques implémentées

Plusieurs dataframes sont créés pour effectuer les calculs puis combinés en un seul dataframe pour les étapes suivantes. Les trois que nous avons choisi d'implémenter sont un pour les données qui sont propres aux utilisateurs, un pour les calculs relatifs aux tweets (proportions url) et un dernier pour les calculs de variance des followings.

2.1.1 Calculs sur le dataframe Utilisateur

La longueur du nom d'utilisateur, longueur de la description du profil, nombre de following et le nombre de followers sont directement collectés dans l'ensemble de données pour chaque type d'utilisateur (pollueur ou légitime) sans traitement supplémentaire nécessaire. Pour la durée de vie du compte, le calcul se fait avec la différence entre la date de création du compte et celle de collection qui sont disponibles dans l'ensemble de données. Le résultat de ce calcul est retournée directement en jours afin de pouvoir l'utiliser pour les prochains calculs. Le rapport « following/followers » est directement calculé à partir des données de l'ensemble.

2.1.2 Calculs sur le dataframe Tweets

Les proportions de mentions et d'URL dans les tweets sont calculés dans ce dataframe avec sensiblement les mêmes étapes. L'on fait le compte des chaînes "$http*$" et "@" dans les tweets par utilisateur. Par la suite l'on divise le compte par le nombre de tweets par utilisateur qui sont à notre disposition dans le dataframe (en gérant les erreurs de division par zéro en remplacant par NaN).
o Nombre moyen de tweets par jour o Temps moyen et maximal entre deux tweets consécutifs
o Deux caractéristiques supplémentaires : Proposez et implémentez deux nouvelles caractéristiques pertinentes.

2.2 Deux caractéristiques ajoutées

Pour les deux nouvelles caractéristiques, nous avons ajouté les métriques **variance du nombre d'utilisateurs suivis** et **similarité z-score**.

2.2.1 Variance du nombre d'utilisateurs suivis

Les fichiers `legitimate_users_followings.txt` et `content_polluters_followings.txt` contiennent les données des utilisateurs concernant le nombre de personnes qu'ils suivent à différents moments. Notre hypothèse est que les utilisateurs pollueurs s'abonnent et se désabonnent souvent, contrairement aux utilisateurs normaux. Calculer la variance du nombre d'abonnements a le potentiel de nous fournir des informations pertinentes quant aux types d'utilisateurs.

Formule de la variance pour le nombre de personnes suivies (σ^2) :

Si l'on considère T instants avec des valeurs $x_1, x_2, \ldots, x_T$, la variance est définie par :

$$\sigma^2 = \frac{1}{T} \sum_{t=1}^{T} (x_t - \mu)^2 ,$$

où :

- σ^2 est la variance du nombre de personnes suivies,

- x_t correspond à la valeur du nombre de personnes suivies au temps t,
- μ est la moyenne des valeurs du nombre de personnes suivies,
- T est le nombre total d'observations.

La métrique de la variance permet d'identifier quel utilisateur s'écarte le plus de sa moyenne de personnes suivies. Plus la variance est élevée, plus le nombre de personnes suivies par l'utilisateur change au fil du temps. À l'inverse, une variance faible indique que le nombre d'abonnements est stable. En vérifiant, avant l'entraînement des différents modèles, la colonne `variance_of_followings` pour les classes, nous observons en moyenne :

- Utilisateurs légitimes : **17 311**
- Utilisateurs pollueurs : **68 504**

Ces chiffres correspondent à notre hypothèse selon laquelle les utilisateurs pollueurs ont tendance à varier plus souvent leur nombre d'abonnements. La différence marquée entre les deux classes a de bonnes chances d'influencer les différents algorithmes d'apprentissage automatique.

2.2.2 Similarité des tweets

Distance de Levenshtein

Pour le critère de similarité des tweets, nous utilisons l'algorithme de la distance de Levenshtein issu de la programmation dynamique. Cette distance mesure de combien de caractères une chaîne diffère d'une autre. Par exemple, le mot `hello world` et le mot `hello` sont à une distance de 6 caractères (suppression des 6 derniers).

Exemple de l'algorithme

	S	a	t	u	r	d	a	y
0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6
u	2	1	1	2	2	3	4	5
n	3	2	2	2	3	3	4	5
d	4	3	3	3	3	4	3	4
a	5	4	3	4	4	4	4	3
y	6	5	4	4	5	5	5	4

NOTE : Ce tableau représente le calcul de la distance de Levenshtein entre les mots **"Saturday"** et **"Sunday"**. Chaque cellule (i, j) indique le coût minimal pour transformer les i premiers caractères du premier mot en les j premiers caractères du second. La première ligne et la première colonne montrent les opérations nécessaires pour convertir un mot vide en l'autre en ajoutant progressivement des caractères. Les cases intérieures sont remplies en prenant le minimum entre trois opérations possibles : la suppression (valeur au-dessus +1), l'insertion (valeur à gauche +1) et la substitution (valeur en diagonale +1 si les lettres sont différentes, sinon on garde la diagonale). Dans ce cas, la distance de Levenshtein entre **"Saturday"** et **"Sunday"** est de **3**, ce qui signifie qu'il faut trois opérations pour transformer un mot en l'autre (voir Levenshtein distance, Wikipedia).

Justification : Nous avons choisi cette métrique, car nous avons émis l'hypothèse que les utilisateurs pollueurs publient des messages semblables à chaque tweet. Par exemple, dans le fichier `content_polluters_tweets.txt`, on trouve ces données :

```
6301    5600313663  THE BURLESQUE BOOTCAMP SYDNEY - Open Date tickets now
    available from http://bbootcampsyd.eventbrite.com/ for Jan /...
    http://bit.ly/rCenZ    2009-11-10 15:46:05
6301    5600328557  THE BURLESQUE BOOTCAMP SYDNEY - Open Date tickets now
    available from http://bbootcampsyd.eventbrite.com/ for Jan /...
    http://bit.ly/1v5hvb    2009-11-10 15:46:40
```

La distance de Levenshtein pour ces deux tweets est de `12`. Contrairement à des messages d'un utilisateur légitime :

```
614     5912305459      at house party in Daybreak. Not as weird as I'd feared.
    ;)   2009-11-20 23:52:52
614     5908467165  Taxiing      at SLC Salt Lake City International
    http://gowal.la/s/b7V     2009-11-20 20:42:48
```

Ces messages ont une distance de `57`.

Optimisation : Le nombre de tweets est considérable, ce qui complique légèrement les calculs. Tout d'abord, nous nous concentrons uniquement sur la distance entre les tweets d'un même utilisateur. Chaque utilisateur peut posséder jusqu'à 200 tweets. La complexité en notation grand O est donc la suivante :

$$O(f) = u \cdot t^2 \cdot m \cdot n,$$

où :

- u représente le nombre d'utilisateurs,
- t correspond au nombre moyen de tweets par utilisateur,
- m est la longueur du premier tweet à comparer,
- n est la longueur du deuxième tweet à comparer.

Pour accélérer les calculs, il existe une bibliothèque en Python nommée `python-Levenshtein`, compilée en Cython (C en Python), qui s'exécute beaucoup plus rapidement qu'une implémentation maison. Nous avons testé notre propre implémentation et le temps d'exécution était de plusieurs heures. Avec la bibliothèque, ce temps descend approximativement à 10 minutes.

Autres langues et messages courts : Cependant, cet algorithme fournit une distance brute. Le problème survient lorsqu'on compare des messages de longueurs très différentes. Prenons, par exemple, ces données d'un utilisateur légitime :

```
'I Need A Hug'
im backkkk!!!!
```

L'algorithme seul donne une distance de `13`, ce qui est plus grand que la distance de l'utilisateur pollueur mentionnée plus haut. Le même problème survient avec des caractères issus d'autres langues. Par exemple, en mandarin, un caractère représente un mot, donc un tweet contient nécessairement moins de caractères qu'en français. À l'inverse, l'allemand est réputé pour ses mots très longs. Ainsi, la distance de Levenshtein seule n'est pas suffisante.

Pour corriger ce problème, nous normalisons la métrique avec la formule suivante. Soient m_1 et m_2 deux messages, et $d_L(m_1, m_2)$ leur distance de Levenshtein. On définit alors la distance normalisée $d_{\text{norm}}(m_1, m_2)$ par :

$$d_{\text{norm}}(m_1, m_2) = \frac{d_L(m_1, m_2)}{\max\{|m_1|, |m_2|\}},$$

où $|m_1|$ et $|m_2|$ représentent respectivement les longueurs de m_1 et m_2.

En d'autres mots, on divise la distance de Levenshtein par la longueur de la chaîne de caractères la plus longue afin de la normaliser. Ainsi, nous obtenons une distance normalisée de `0.93` pour les données :

```
THE BURLESQUE BOOTCAMP SYDNEY – Open Date tickets now available from
    http://bbootcampsyd.eventbrite.com/ for Jan /... http://bit.ly/rCenZ
THE BURLESQUE BOOTCAMP SYDNEY – Open Date tickets now available from
    http://bbootcampsyd.eventbrite.com/ for Jan /... http://bit.ly/1v5hvb
```

Et une distance normalisée de `0.09` pour les données :

```
'I Need A Hug'
im backkkk!!!!
```

On remarque qu'après la normalisation, la valeur de `0.93` pour l'utilisateur pollueur est bien plus grande que celle de l'utilisateur légitime, ce qui correspond à nos attentes.

Comparaison entre utilisateurs : Enfin, nous ajustons la métrique avec une dernière étape pour comparer les distances entre elles. Nous voulons obtenir une seule valeur de distance par utilisateur. Pour cela, nous utilisons la **moyenne** de toutes les distances de Levenshtein d'un utilisateur (toutes les paires de tweets, soit t^2). Ensuite, nous appliquons une **cote Z (z-score)** en supposant une distribution normale centrée à 2. Nous nous assurons que toutes les valeurs de la cote Z sont positives afin de travailler uniquement avec des valeurs supérieures à zéro (ce qui est pratique pour certains algorithmes).
Cette normalisation permet d'établir une distribution où :

- Les utilisateurs ayant des tweets très similaires (pollueurs) obtiennent des valeurs élevées.
- Les utilisateurs dont les tweets sont plus variés obtiennent une **faible similarité** et donc une **cote basse**.

Cette mesure nous permet d'identifier les utilisateurs qui spamment des messages similaires et de comparer le taux de similarité des messages par rapport aux autres utilisateurs. Nous espérons que cette métrique pourra discriminer efficacement les utilisateurs pollueurs et légitimes à l'aide de seuils définis par les algorithmes d'apprentissage automatique.

2.3 Valeurs manquantes

Il est à noter que pour chaque calcul comportant une possibilité d'erreur, par exemple une différence avec une seule valeur, une division par 0 ou NaN, le champ est remplacé par une valeur NaN qui sera dropped lors du nettoyage final. Nous avons décidé qu'il était plus pertinent de garder les données complètes et ainsi nettoyer les données potentiellement aberrantes avec *dropna*(). Suite au nettoyage des données nous retirons 3536 lignes soit 8.52% des données. Les 1499 premières sont celles dont les utilisateurs ont des tweets vides et les autres proviennent d'utilisateurs qui on 0 ou un seul tweet à leur actif et donc les calculs des distances de Levenshtein et leur Z-score n'étaient pas calculables. Vu que cela ne représente pas une trop grande proportion des données nous avons préféré les enlever.

3 Analyse comparative sur les données équilibrées

Au cours de nos expérimentations, nous avons à faire deux tests sur six algorithmes. Les tests sont **classes équilibrées** et **classes déséquilibrées**. Les algorithmes d'apprentissage machine sont **(1) Arbre de décision**, **(2) Bagging**, **(3) AdaBoost**, **(4) Boosting de gradient (GBoost)**, **(5) Forêts d'arbres aléatoires**, **(6) Classification bayésienne naïve**.

La partie "**3. Analyse comparative sur les données équilibrées**" porte sur le test avec des classes équilibrées où nous conservons toutes les données disponibles. Les classes ne seront pas exactement égales. Nous avons des données de `18 826` utilisateurs légitimes et `19 137` utilisateurs pollueurs. Le but de l'exercice de la partie "**4. Analyse comparative sur des classes déséquilibrées**" est de mesurer l'effet qu'un déséquilibre des classes peut avoir en ne prenant que 5 % des utilisateurs pollueurs.

3.1 Description des étapes

Pour réaliser ces expérimentations, nous devons :

1. **Charger les données :** lire les données du fichier `preprocessed_data.csv`, diviser les données en ensembles d'entraînement et de validation.
2. **Entraîner les modèles :** définir les différents types de modèles et leurs hyper-paramètres, utiliser les données selon le test (déséquilibré ou non) et procéder à l'entraînement.
3. **Évaluer et afficher les résultats :** utiliser les modèles entraînés pour prédire les classes de l'ensemble de test (données non présentes dans l'entraînement), mesurer les taux de vrais positifs (pollueur : classe 1, bien identifié), faux positifs (pollueur : classe 1, mal identifié), faux négatifs (légitime : classe 0, mal identifié), vrais négatifs (légitime : classe 0, bien identifié), mesurer le F-score (moyenne combinant la précision et le rappel) et mesurer la courbe ROC (l'aire sous la courbe). Les résultats seront présentés dans une matrice de confusion 2 x 2 et par des graphes pour la courbe ROC. La diagonale de cette matrice représente les prédictions correctes. L'objectif est d'avoir le plus de vrais positifs et de vrais négatifs (précision) en plus d'un bon taux de vrais positifs par rapport au total des positifs (rappel).

3.1.1 Implémentation

Pour implémenter les étapes énoncées précédemment, nous avons construit deux classes, `DataLoader` et `ModelTrainer`. La classe `DataLoader` se charge de lire les fichiers et de charger les données en mémoire. C'est elle qui s'occupe de diviser les données en ensembles d'entraînement et de validation.

La classe `ModelTrainer` est responsable de l'entraînement selon la balance des classes, puis d'évaluer les performances des modèles.

Cette façon d'organiser le code nous permet de facilement réaliser nos tests avec peu de lignes.
Lancer les tests avec différents ratios de classes

```
comparaison_algorithmes("preprocessed_data.csv")
comparaison_algorithmes("preprocessed_data.csv", imbalance_ratio=0.05)
```

Charger les données

```
data_loader = DataLoader(file_path, imbalance_ratio=imbalance_ratio)
X_train, X_test, y_train, y_test = data_loader.preprocess_data()
```

Définir des modèles

```
models = [
    ModelTrainer("DecisionTree", DecisionTreeClassifier(random_state=42),
        imbalance_ratio),
    ModelTrainer("Bagging",
        BaggingClassifier(estimator=DecisionTreeClassifier(), n_estimators=50,
        random_state=42), imbalance_ratio),
    ...
]
```

Entraîner et mesurer les modèles :

```
for model in models:
    model.train_model(X_train, y_train)
    model.evaluate_model(X_test, y_test)
    results.append(model.results)
```

Pour plus de détails, voir le fichier `comparison_all_algorithms.py`.

3.2 Résultats

Table 4: Résultats sur les données équilibrées

Modèle	TP Rate	FP Rate	F-Mesure	AUC
DecisionTree	0.91028	0.08939	0.91076	0.91044
Bagging	0.94989	0.06162	0.94469	0.98407
AdaBoost	0.93730	0.06797	0.93509	0.98138
GradientBoosting	**0.95121**	0.06744	0.94268	0.98431
RandomForest	0.94989	0.06030	**0.94531**	**0.98522**
NaiveBayes	0.13457	**0.02856**	0.23144	0.80698

3.3 Visualisation courbes ROC pour données équilibrées

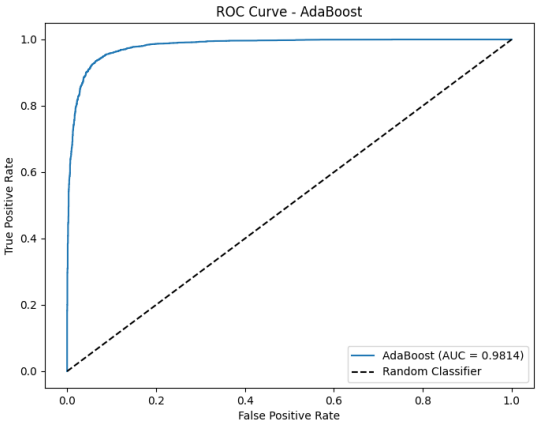

Figure 4: Courbe ROC AdaBoost

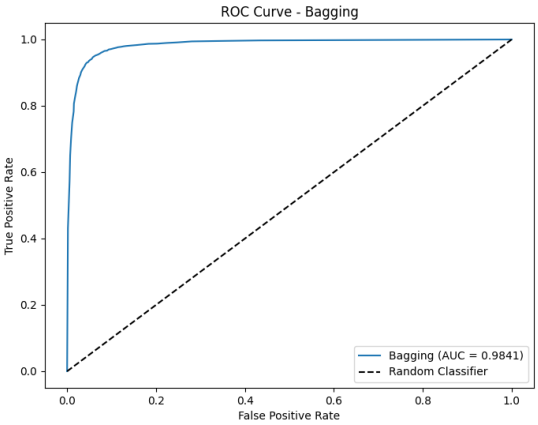

Figure 5: Courbe ROC Bagging

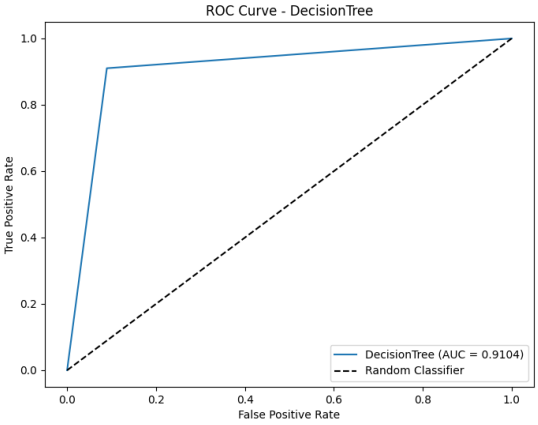

Figure 6: Courbe ROC DecisionTree

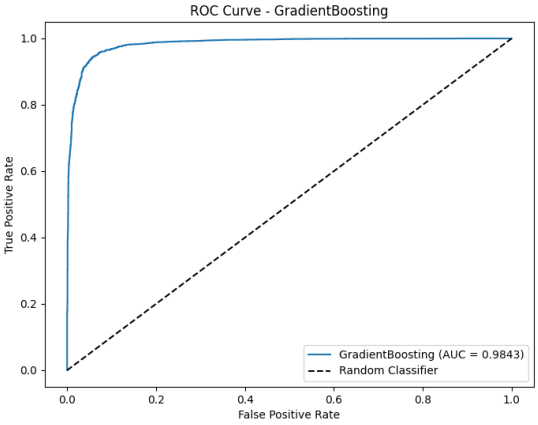

Figure 7: Courbe ROC GradientBoosting

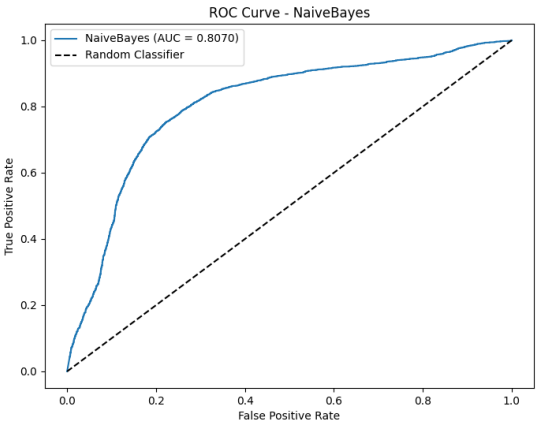

Figure 8: Courbe ROC NaiveBayes

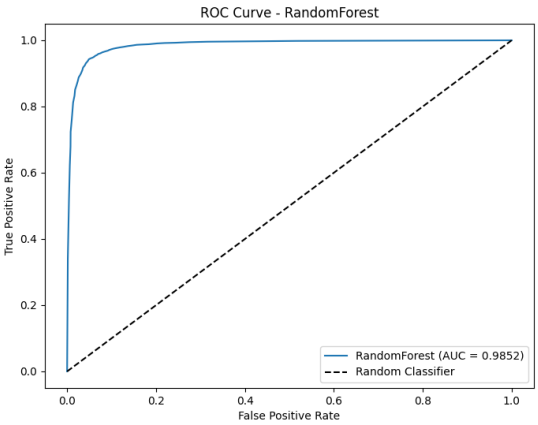

Figure 9: Courbe ROC RandomForest

Figure 10: Visualisation des courbes ROC par modèle pour données équilibrées

3.4 Visualisation des matrices de confusion pour données équilibrées

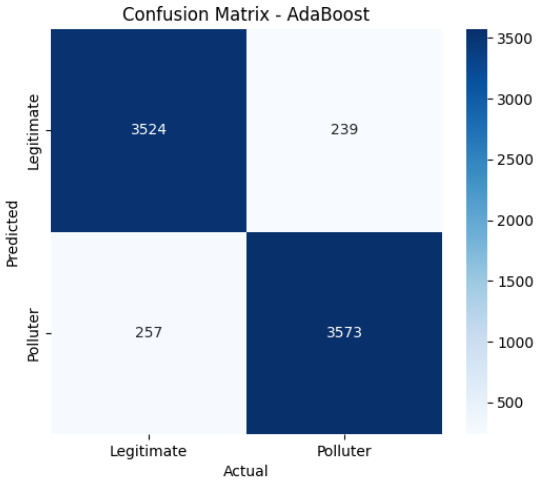

Figure 11: Matrice AdaBoost

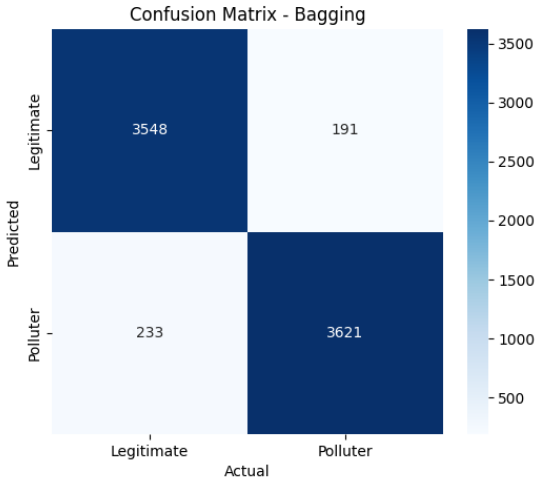

Figure 12: Matrice Bagging

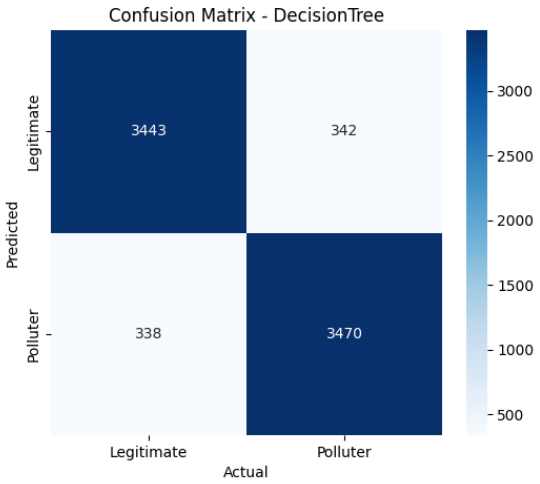

Figure 13: Matrice DecisionTree

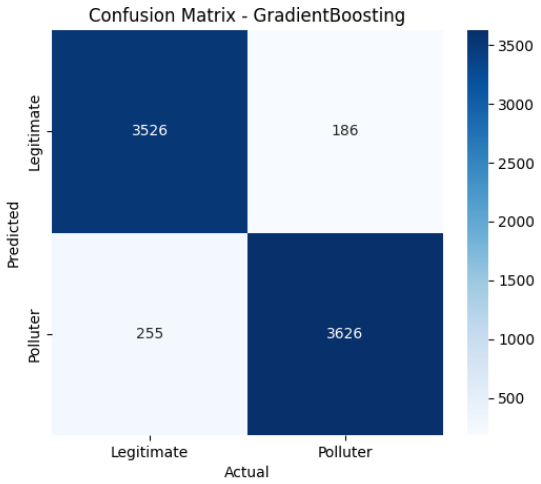

Figure 14: Matrice GradientBoosting

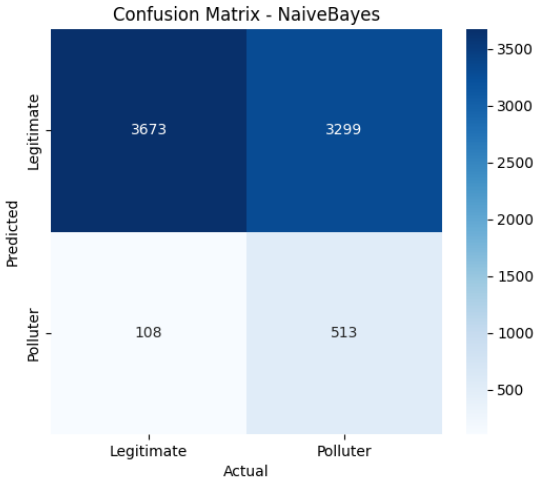

Figure 15: Matrice NaiveBayes

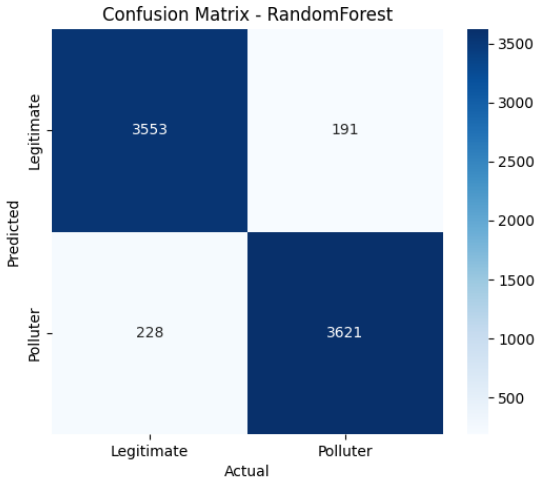

Figure 16: Matrice RandomForest

Figure 17: Visualisation des matrices de confusion par modèle pour données équilibrées

3.5 Analyse

3.5.1 DecisionTree

Pour l'arbre de décision, on peut remarquer que son taux de vrais positifs (TP Rate) est le plus bas des algorithmes qui performent bien (le Bayésien est non adapté comme on peut le constater). De plus, il a le taux de faux positifs le plus élevé, ce qui permet de souligner les forces, mais surtout les faiblesses de ce modèle. Il a pour avantage d'être simple et polyvalent, ce qui est un point de départ, mais en aucun cas cela ne compense ses faiblesses dans notre cas. Ce modèle a pour inconvénient de se surajuster et d'être sensible aux données plus complexes ou déséquilibrées. Il est cependant robuste aux données manquantes ce qui n'est pas notre cas. Certaines des caractéristiques, comme les tweets par jour et les proportions d'URL/mentions, sont des indicateurs clairs pour la classification ; d'autres, comme la longueur du nom d'utilisateur, viennent fausser les résultats finaux, car ce modèle se surajuste sur celles-ci. Il fournit de bonnes performances globalement, mais son taux de faux positifs est trop élevé pour nos besoins.

3.5.2 Bagging

Le modèle de Bagging fonctionne en aggrégant des arbres de décision avec des sous-ensembles de données et cela permet de réduire la variance. On peut constater que les performances sont très bonnes avec un TP Rate de 0.94989 et une F-Mesure (équilibre précision/rappel) de 0.94469 ce qui en fait le second modèle le plus performant. Ses avantages est qu'il est robuste face aux données désiquilibrées et que la variance des données est fortement réduite. Par exemple, des données qui sont très hétérogènes comme le ratio following_follower_ratio dans les deux types d'utilisateurs, une réduction de la variance permet d'obtenir moins de faux positifs. Les euls inconvénients est qu'il peut prendre plus de temps et de ressources pour l'entraînement, mais avec notre ensemble de données c'est négligeable.

3.5.3 AdaBoost

Le modèle AdaBoost a une approche basée sur les éléments qui sont mal classés et en réentrainant en attribuant des poids afin d'affiner la classification pour chaque itération. Pour ce qui est des résultats nous avons obtenu un TP rate de 0.9373 et un FP Rate de 0.06797 ce qui le place dans les modèles qui performent adéquatement. Les forces de ce modèle est qu'il améliore sa précision au fur et à mesure en pondérant les erreurs. Cela veut dire que les arbres qui prédisent de mauvais résultats ont un poids plus élevé (donc plus d'attention) et que cela permet d'afffiner les résultats en se concentrant sur les contre-exemples. Il est donc logique que ce modèle soit sensible aux données aberrantes qui recevront beaucoup d'attention et au bruit par la méme occasion comme nous l'observeront sur les données désiquilibrées.

3.5.4 GradientBoosting

Le modèle de GradientBoosting possède plus haut TP Rate (0.95121) de tous les modèles, mais performe assez semblablement pour le FP Rate (0.06744) et la F-Measure (0.94268) par rapport aux autres modèles. Comme détaillé plus haut, le GBoost apporte de bonne performances suite à la correction des erreurs de façon itérative mais c'est un modèle qui nécessite une plus grande attention pour ses hyperparamètres. Il a l'avantage de bien performer sur notre jeu de données qui est complexe et les relations qui sont non-linéaires comme avg_time_between_tweets ou following_follower_ratio sont bien prises en compte. un autre inconvénient est qu'il prompt à faire du surajustement s'il n'est pas bien réglé. Dans notre cas, nous voulions obtenir un résultat baseline sans optimisation des hyperparamètres afin de mettre en lumière ce genre d'inconvénients.

3.5.5 RandomForest

Le modèle de RandomForest ou Forêt Aléatoire combine plusieurs arbres de décision et comme le Bagging il permet de réduire la variance et augmenter la généralisation. Des caractéristiques redondantes comme le num_following et following_follower_ratio sont bien exploitées et celles qui sont moins importantes/discriminantes (comme le z_score_similarity) ont un impact plus faible et la robustesse est apparente dans les résultats. Avec un TP Rate de 0.94989 et le FP Rate le plus bas (dans les modèles performants car le NaiveBayes est aberrant) de 0.06030, il est le modèle à privilégier pour cet ensemble de données. Il possède aussi la plus haute Area Under the Curve (AUC) de 0.98522 ainsi que la plus haute F-Mesure de 0.94531. Cela en fait un modèle non seulement performant, mais qui réduit la variance en gardant une bonne précision et plus robuste face aux données bruitées.

3.5.6 NaiveBayes

Le modèle NaiveBayes assume une indépendance entre les caractéristiques ce qui est bien entendu irréaliste pour nos données. Des caractéristiques fortement corrélés sont présentent dans nos données car certaines sont même des bases pour en calculer d'autres (num_followers, num_following et following_follower_ratio par exemple). On peut remarquer avec la matrice de confusion que les utilisateurs légitimes sont bien classés (le FP rate est le plus bas de tous les modèles avec 0.02856) mais qu'il est incapable de classer les pollueurs comme tels et qu'ils sont classés comme légitimes. Il est apparent que le modèle ne peut pas interpréter les relations de variables plus complexes qui sont caractéristiques des utilisateurs pollueurs. Par contre il possède des avantages certains comme la simplicité d'implémentation et d'entraînement. Il est aussi très peu gourmand en ressources ce qui peut être utile avec des jeux de données plus massifs.

4 Analyse comparative sur des classes déséquilibrées

4.1 Description des étapes

Les étapes pour cette section sont bien décrites dans la section **"3.1 Description des étapes"**. Le seul changement dans le code se trouve dans ces lignes où l'on appelle la fonction de comparaison des algorithmes avec un paramètre de déséquilibre :

```
comparaison_algorithmes("preprocessed_data.csv")
comparaison_algorithmes("preprocessed_data.csv", imbalance_ratio=0.05)
```

4.2 Résultats

Table 5: Résultats sur les données déséquilibrées

Modèle	TP Rate	FP Rate	F-Mesure	AUC
DecisionTree	0.63746	0.01772	0.770328	0.80987
Bagging	**0.66710**	0.00634	**0.79730**	0.96710
AdaBoost	0.63221	0.01084	0.76959	0.97839
GradientBoosting	0.64375	**0.00449**	0.78115	**0.98031**
RandomForest	0.61542	0.00608	0.75910	0.97347
NaiveBayes	0.11831	0.02036	0.20783	0.81501

4.3 Visualisation courbes ROC pour données déséquilibrées

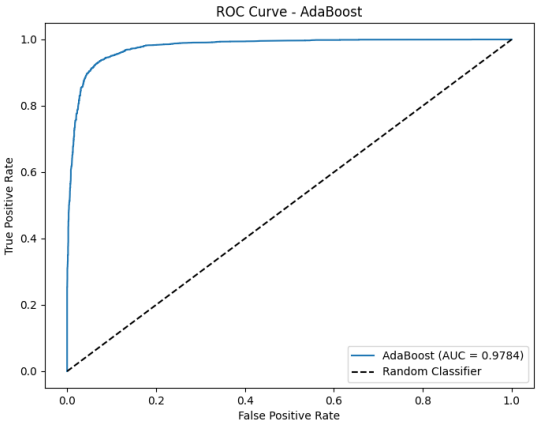

Figure 18: Courbe ROC AdaBoost

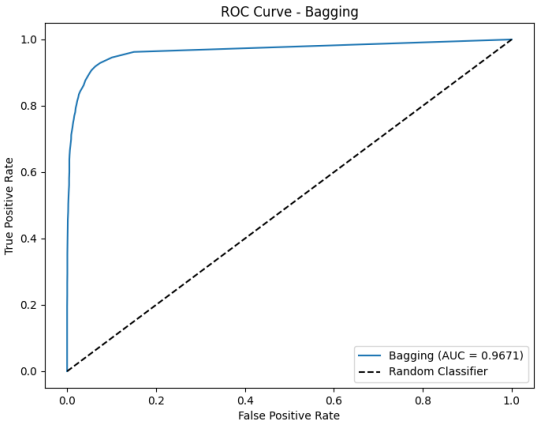

Figure 19: Courbe ROC Bagging

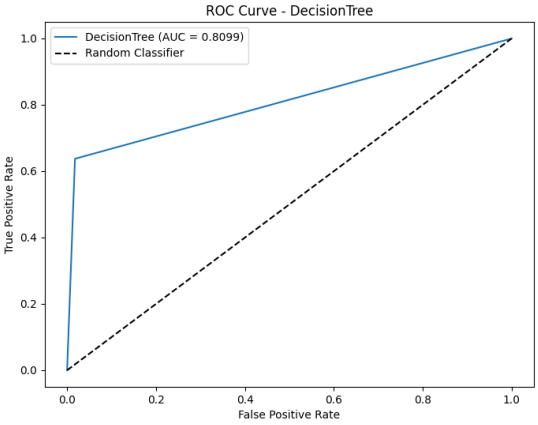

Figure 20: Courbe ROC DecisionTree

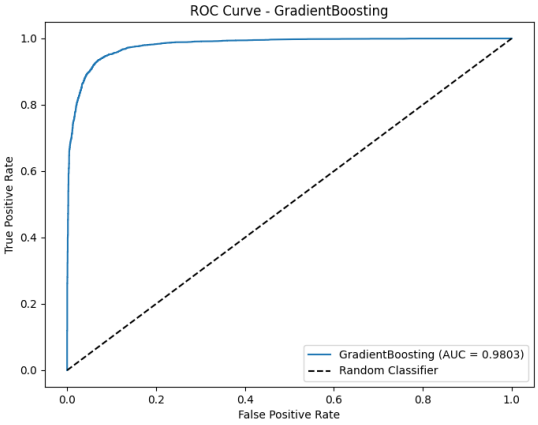

Figure 21: Courbe ROC GradientBoosting

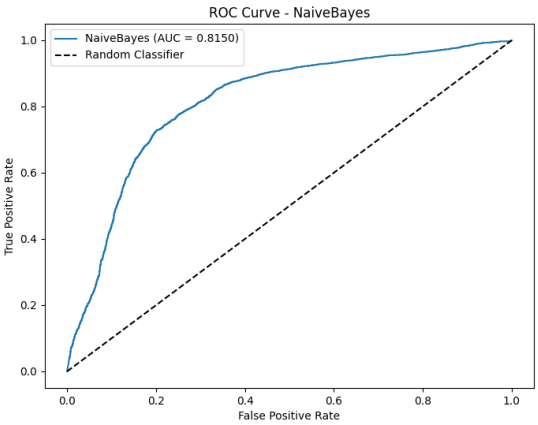

Figure 22: Courbe ROC NaiveBayes

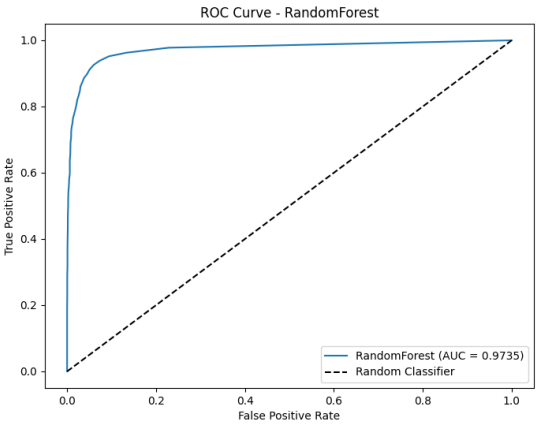

Figure 23: Courbe ROC RandomForest

Figure 24: Visualisation des courbes ROC par modèle pour données déséquilibrées

4.4 Visualisation des matrices de confusion pour données déséquilibrées

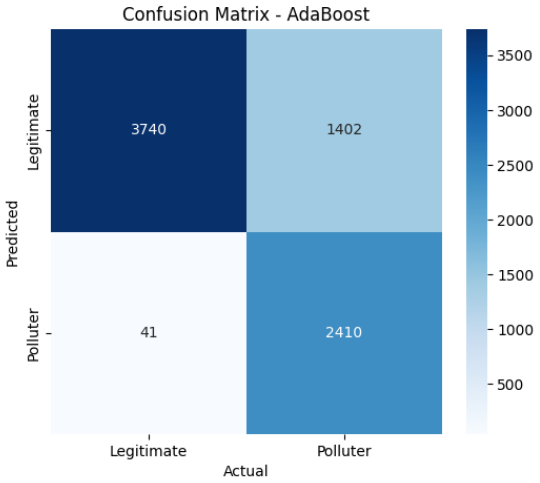

Figure 25: Matrice AdaBoost

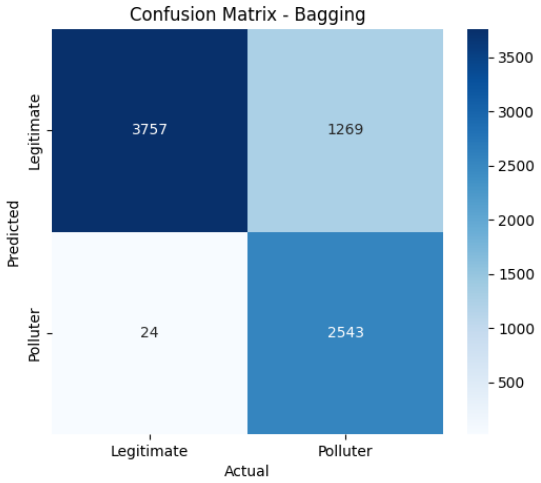

Figure 26: Matrice Bagging

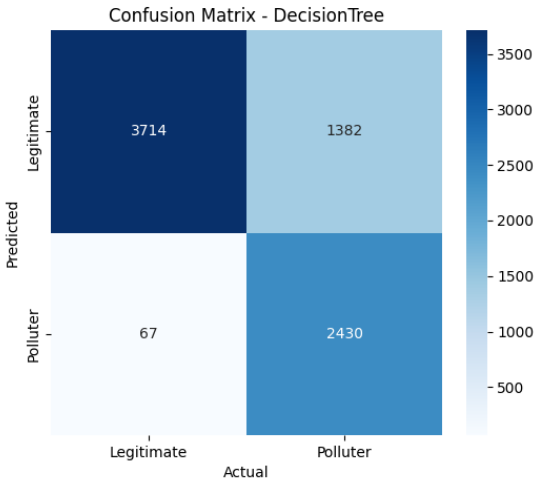

Figure 27: Matrice DecisionTree

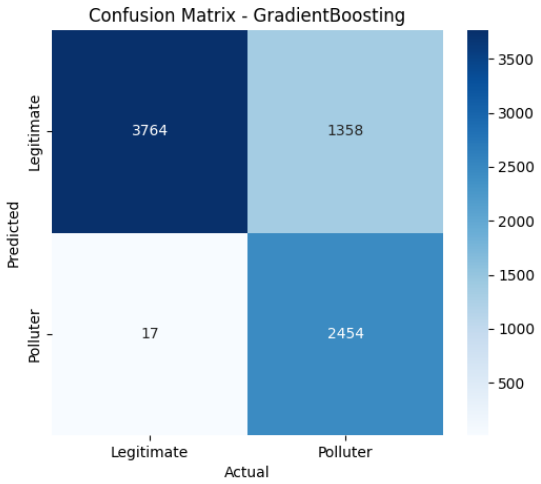

Figure 28: Matrice GradientBoosting

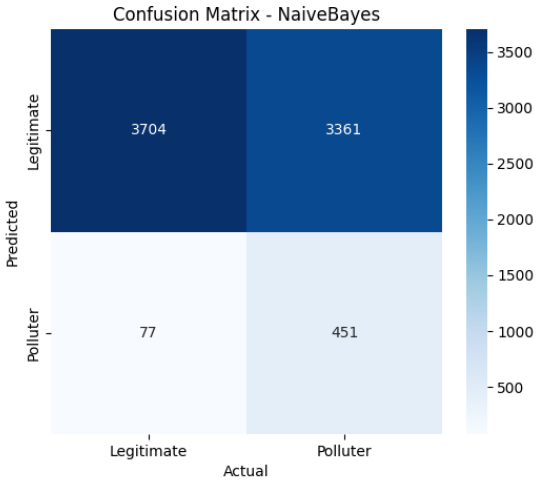

Figure 29: Matrice NaiveBayes

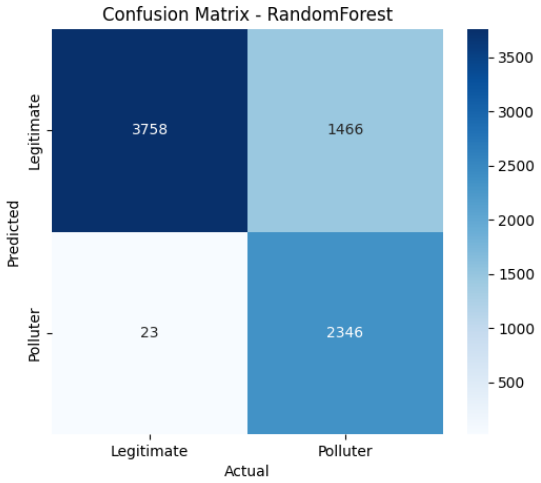

Figure 30: Matrice RandomForest

Figure 31: Visualisation des matrices de confusion par modèle pour données déséquilibrées

4.5 Analyse

4.5.1 DecisionTree

Le DecisionTree a (comme chaque modèle sur cet ensemble de données déséquilibrées) une perte de performance qui se ressent dans chaque métrique. Étant donné que le DecisionTree est sensible aux données déséquilibrées et au surapprentissage, le TP Rate passe de 0.910 à 0.637 et la matrice de confusion nous permet d'observer que la détection des pollueurs est l'aspect qui s'est le plus détérioré. Cela est par contre logique car la classe majoritaire (les utilisateurs légitimes) influe sur la capacité de détection de la classe minoritaire (les pollueurs).

4.5.2 Bagging

Le modèle de Bagging est celui qui offre les meilleurs résultats sur les données déséquilibrées avec un TP Rate de 0.667 et un bas FP Rate de 0.0063. Le Bagging, grâce à sa combinaison d'arbres et à la réduction de la variance est donc plus robuste aux données déséquilibrées et c'est pour cela qu'il garde une performance plus adéquate.

4.5.3 AdaBoost

AdaBoost améliore l'AUC mais ne parvient pas à maintenir le rappel sur la classe minoritaire (0.937 chute à 0.632), car il optimise principalement la précision globale. C'est un modèle qui est sensible au bruit et au déséquilibre. Si un exemple bruité de la classe pollueurs est mal classé, il va être surpondéré, ce qui entraînera un surapprentissage sur celui-ci au lieu d'améliorer la généralisation globale.

4.5.4 GradientBoosting

GradientBoosting maximise la séparation entre les classes, ce qui explique la faible perte d'AUC (0.9843 devient 0.9803), mais il reste sensible au rappel, particulièrement sur des données déséquilibrées. Il est à souligner que le taux de faux positifs est très faible et cela est sûrement dû au fait que le modèle ajuste ses pondérations en accordant plus d'importance aux exemples mal classés. Cela permet de mieux identifier les pollueurs sans trop pénaliser les utilisateurs légitimes. de plus, le pruning automatique fait en sorte que les arbres faibles créés sont de faible profondeur, ce qui limite la complexité et réduit les erreurs de classification sur la classe majoritaire.

4.5.5 RandomForest

RandomForest tout comme DecisionTree privilégie la précision globale ce qui vient dégrader le rappel (0.950 devient 0.615) à cause du déséquilibre. Une solution pourrait être de limiter la profondeur maximale de l'arbre. Cela pourrait améliorer le rappel en forçant le modèle à créer des divisions plus fines sur la classe minoritaire.

4.5.6 NaiveBayes

Comme pour les données équilibrées le NaiveBayes n'est pas adéquat pour des données qui sont corrélées encore moins si elles sont déséquilibrées. C'est un modèle qui privilégie les classes qui sont majoritaires donc la détection de la classe minoritaire est encore plus ardue et cela est reflété dans les résultats. Dans la matrice de confusion on peut clairement observer que les utilisateur pollueurs sont presques tous classés comme légitimes.

5 Conclusion

L'analyse comparative des différents modèles sur des données équilibrées et déséquilibrées montre que Bagging offre le meilleur compromis entre précision et rappel, avec des performances stables même sur des données déséquilibrées, tandis que GradientBoosting est meilleur en termes d'AUC et de faible taux de faux positifs. Par contre, NaiveBayes est à éviter car il a de très faibles performances, et RandomForest nécessite des ajustements pour améliorer le rappel face aux données déséquilibrées. Pour optimiser les performances, il faudrait utiliser des techniques de rééquilibrage des données comme SMOTE ou l'undersampling, d'ajuster les hyperparamètres, de combiner les modèles ou d'utiliser la validation croisée et d'effectuer une analyse approfondie des erreurs.