UNIVERSITÉ DU QUÉBEC À MONTREAL

TP1 PRÉSENTÉ À

MOHAMED BOUGUESSA

DANS LE CADRE DU COURS

INF7370 – Apprentissage machine

Réalisé par :

LARBI CHAOUCH CHAL22049300

CHAOUCH.LARBI@COURRIER.UQAM.CA

~~***À remettre avant le 24-02-2025***~~

À remettre avant le 03-03-2025

1. Tâches à réaliser

Dans l'énoncé il parle de GBoost et dans la directive pour le rapport 1 il parle de XBOOT, je n'ai rien trouvé sur le XBoot ?? .

A. Tâche 1 : Compréhension de l'algorithme GBoost

Histoire et Origine du Gradient Boosting

Le Gradient Boosting a été introduit par Jerome Friedman en 1999 comme une ***généralisation*** du concept de **Boosting** proposé par Freund et Schapire avec l'algorithme AdaBoost en 1996. Le but du Boosting est d'améliorer la précision d'un modèle en combinant plusieurs apprenants faibles (weak learners) qui corrigent les erreurs des prédictions précédentes
Source : *analyticsindiamag.com*

Le Gradient Boosting est une extension du Boosting où chaque nouvel apprenant est formé pour réduire les erreurs résiduelles laissées par le modèle précédent. Contrairement à AdaBoost qui ajuste les poids des erreurs avec un facteur multiplicatif, Gradient Boosting optimise une fonction de perte à l'aide de la descente de gradient, d'où son nom.
Source : *datascientest.com*

Algorithme Brut du Gradient Boosting

L'algorithme suit un schéma récurrent d'apprentissage en plusieurs étapes :
1. Initialisation : Définir une fonction de base simple, souvent une constante, qui sert de première approximation des résultats.

2. Calcul des résidus : À chaque itération, on évalue la différence entre les prédictions actuelles et les valeurs réelles (les résidus).

3. Apprentissage des résidus : Un nouvel arbre de décision est construit pour prédire ces résidus.

4. Mise à jour du modèle : La prédiction du nouvel arbre est multipliée par un facteur d'apprentissage (**`learning rate`**) et ajoutée au modèle précédent.

5. Itération : Répéter ce processus pour un certain nombre d'arbres.

———

Mathématiquement parlons, on modélise la fonction Fm (x) à l'étape m comme suit :

$$F_m(x) = F_{m-1}(x) + \gamma h_m(x)$$

où $h_m(x)$ est l'arbre de régression entraîné sur les résidus, et γ est un facteur de pondération ajustant l'apprentissage
Source : *masedki.github.io*

Explication avec un Exemple Pratique

Imaginons que nous essayons de prédire le prix d'un appartement basé sur des caractéristiques comme la surface, le nombre de chambres et l'emplacement.
Première prédiction : J'ai fais une estimation naïve, par exemple en prenant la moyenne de tous les prix.
Calcul des erreurs : J'ai évalué à quel point cette prédiction est éloignée des valeurs réelles.
Ajout d'un premier arbre : J'ai formé un arbre pour corriger les erreurs de cette estimation initiale.
Mise à jour : J'ai ajusté la prédiction en combinant la moyenne initiale et l'arbre nouvellement appris.
Réitération : J'ai répété ce processus avec de nouveaux arbres, chacun corrigeant les erreurs restantes.
Ce processus permet d'obtenir un modèle plus précis après plusieurs itérations
Source : *datascientest.com*

Avantages du Gradient Boosting

+ Réduction du biais et de la variance : Contrairement aux méthodes classiques, il améliore continuellement la précision du modèle.
+ Capacité à modéliser des relations complexes : Il capte mieux les interactions entre les variables que les modèles linéaires.
+ Flexibilité : Peut être utilisé pour des tâches de classification et de régression avec différentes fonctions de perte.
+ Meilleure gestion du bruit que AdaBoost : Moins sensible aux données aberrantes en utilisant des techniques de régularisation
Source : *datascientest.com*

Voici un exemple d'implémentation :

Implémentation en Python

J'ai implémenté le **`Gradient Boosting`** avec **`Scikit-Learn`** et je vais expliquer chaque ligne de code. (Cet exemple est mon propre exemple que j'ai inventé), et ceci est à titre explicatif pas plus.

```
import numpy as np
import pandas as pd
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# --- 1. Génération de données factices ---
np.random.seed(42)
X = np.random.rand(1000, 5)  # 1000 échantillons avec 5
caractéristiques
y = np.random.randint(0, 2, 1000)  # Classification binaire (0 ou 1)

# --- 2. Séparation des données en train et test ---
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# --- 3. Initialisation et entraînement du modèle GBoost ---
gboost = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1, max_depth=3, random_state=42)
gboost.fit(X_train, y_train)

# --- 4. Prédiction sur l'ensemble de test ---
y_pred = gboost.predict(X_test)

# --- 5. Évaluation des performances ---
accuracy = accuracy_score(y_test, y_pred)
print(f"Précision du modèle : {accuracy:.4f}")
```

Explication du code :

1. Importation des bibliothèques

```
import numpy as np
import pandas as pd
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy score
```

EXPLICATION :

+ **numpy** : Manipulation de tableaux numériques.
+ **pandas** : Gestion des données sous forme de tableaux (DataFrame).
+ **GradientBoostingClassifier** : Implémentation du Gradient Boosting de Scikit-Learn.
+ **train_test_split** : Sépare les données en jeu d'entraînement et de test.
+ **accuracy_score** : Évalue la performance du modèle en comparant les prédictions aux valeurs réelles.

2. Génération de données

```
np.random.seed(42)
X = np.random.rand(1000, 5)  # 1000 échantillons avec 5 caractéristiques
y = np.random.randint(0, 2, 1000)  # Classification binaire (0 ou 1)
```

+ **`np.random.seed(42)` :** Définit un point de départ pour la génération aléatoire, pour garantir que les mêmes données soient générées à chaque exécution.

+ **`X = np.random.rand(1000, 5)` :** Génère une matrice (1000, 5) de nombres entre 0 et 1, simulant un jeu de données avec 1000 échantillons et 5 variables.

+ **`y = np.random.randint(0, 2, 1000)` :** Génère 1000 étiquettes (0 ou 1) aléatoirement, simulant un problème de classification binaire.

3. Séparation des données en train et test

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

+ **`train_test_split(X, y, test_size=0.2, random_state=42)`** : Sépare X et y en 80% pour l'entraînement et 20% pour le test (test_size=0.2).
+ **`random_state = 42`** garantit que la séparation est toujours la même à chaque exécution.
+ **`X_train, X_test`** : Contiennent les caractéristiques (features).
+ **`y_train, y_test`** : Contiennent les étiquettes (0 ou 1).

4. Initialisation du modèle Gradient Boosting

```
gboost = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1, max_depth=3, random_state=42)
```

+ **GradientBoostingClassifier**(...) : Création du modèle avec les paramètres suivants :
+ **n_estimators=100** → 100 arbres de décision faibles seront créés.
+ **learning_rate=0.1** → Chaque nouvel arbre apprend à 10% de la puissance pour éviter le surajustement.
+ **max_depth=3** → Chaque arbre peut avoir jusqu'à 3 niveaux pour limiter la complexité.
+ **random_state=42** → Assure une reproductibilité des résultats.

5. Entraînement du modèle

```
gboost.fit(X_train, y_train)
```

+ **.fit(X_train, y_train)** : Apprend à partir des données d'entraînement (X_train, y_train). L'algorithme crée 100 arbres et les entraîne à corriger les erreurs du modèle précédent.

6. Prédiction sur l'ensemble de test

```
y_pred = gboost.predict(X_test)
```

+ **.predict(X_test)** : Utilise le modèle entraîné pour prédire les classes (0 ou 1) sur les données de test (X_test).
+ **y_pred** : Stocke les prédictions du modèle.

7. Évaluation du modèle

```
accuracy = accuracy_score(y_test, y_pred)
print(f"Précision du modèle : {accuracy:.4f}")
```

+ **accuracy_score(y_test, y_pred)** : Compare les prédictions (y_pred) aux valeurs réelles (y_test) et retourne un score entre 0 et 1, représentant la précision du modèle.
+ Affichage du résultat : **{accuracy:.4f}** affiche la précision avec 4 décimales.

8. Résultat du code :

```
$ python index.py
Précision du modèle : 0.5450
```

On peut modifier cette performance on jouant sur les paramètres, par exemple au lieu d'avoir un max de 3 niveaux dans l'arbre, on peut modifier à deux, voici le changement et le résultat :

Modification :

```
gboost = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1, max_depth=2, random_state=42)
```

Nouveau résultat :

```
$ python index.py
Précision du modèle : 0.5500
```

On peut modifier encore :

```
gboost = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.5, max_depth=2, random_state=42)
```

Nouveau résultat :

```
$ python index.py
Précision du modèle : 0.5900
```

Et ainsi de suite …

B. Tâche 2 : Préparation des données

N.B :

-1- La séparation entre les colonnes a été faite par les tabulations, et entre les valeurs de la même colonne a été faite par les « , »

-2- Il y a des données sous la forme de chaine de caractère, et l'espace qui se trouve entre les mots, on ne le considère pas comme une séparation.

-a- D'après le fichier fournie : "**`content polluters.txt`**" contient des informations sur le profile des pollueurs sous la forme de

UserID	CreatedAt	CollectedAt	NumerOfFollowings	NumberOfFollowers	NumberOfTweets	LengthOfScreenName	LengthOfDescriptionInUserProfile

-b- D'après le fichier fournie : "**`content polluters followings.txt`**" contient des informations sous la forme de

UserID	SeriesOfNumberOfFollowings (chaque nombre est séparé par ,)

-c- D'après le fichier fournie : "**`content polluters tweets.txt`**" contient des tweets sous la forme de

UserID	TweetID	Tweet	CreatedAt

-d- D'après le fichier fournie : "**`legitimate users.txt`**" contient des informations sur le profile des utilisateurs légitimes sous la forme de

UserID	CreatedAt	CollectedAt	NumerOfFollowings	NumberOfFollowers	NumberOfTweets	LengthOfScreenNamen	LengthOfDescriptionInUserProle

-e- D'après le fichier fournie :"**`legitimate users followings.txt`**" contient des informations sous la forme de

UserID	SeriesOfNumberOfFollowings (chaque nombre est séparé par ,)

-e- D'après le fichier fournie :**"`legitimate users tweets.txt`"** contient des tweets sous la forme de

UserID	TweetID	Tweet	CreatedAt

Identification des caractéristiques

- **`UserID`**
- **`CreatedAt`** (date de création du compte)
- **`CollectedAt`** (date de collecte des données)
- **`NumerOfFollowings`** (nombre de followings)
- **`NumberOfFollowers`** (nombre de followers)
- **`NumberOfTweets`** (nombre de tweets postés)
- **`LengthOfScreenName`** (longueur du nom d'utilisateur)
- **`LengthOfDescriptionInUserProfile`** (longueur de la description du profil)

1. Caractéristiques implémentée

Pour cette partie, j'avais extrait les **11 caractéristiques** suivantes à partir des données fournies. Voici comment elles ont été calculées :

- **`LengthOfScreenName`** : Nombre de caractères dans le nom d'utilisateur.
- **`LengthOfDescriptionInUserProfile`** : Nombre de caractères dans la bio de l'utilisateur.
- **`AccountLifetime`** : Nombre de jours entre la création du compte et la collecte des données.
- **`NumerOfFollowings`** : Nombre de comptes suivis par l'utilisateur.
- **`NumberOfFollowers`** : Nombre d'abonnés de l'utilisateur.
- **`FollowingsFollowersRatio`** : Ratio entre les comptes suivis et les abonnés.
- **`TweetsPerDay`** : Moyenne du nombre de tweets publiés par jour.
- **`URLRatio`** : Proportion de tweets contenant des liens.
- **`MentionRatio`** : Proportion de tweets contenant des mentions (@).
- **`HashtagRatio`** : Proportion de tweets contenant des hashtags (#).
- **`AvgTweetLength`** : Longueur moyenne des tweets publiés par l'utilisateur.

En plus des 11 caractéristiques, j'ai ajouté de **MeanTimeBetweenTweets** et **MaxTimeBetweenTweets** qui peut expliquer et se justifie par l'importance de **l'intervalle temporel** dans l'analyse du comportement d'un compte Twitter :

A. MeanTimeBetweenTweets (temps moyen entre deux tweets)

- Cette mesure permet de conclure le rythme de publication anormalement élevé ou très régulier.
- Un compte qui tweete à des intervalles quasi constants ou extrêmement rapprochés est souvent automatisé ou manipulé par des scripts (bots).
- Au contraire, un utilisateur humain tend à présenter davantage de variations dans ses délais de publication, en fonction de facteurs réels (heures de sommeil, heures de travail, etc.).

B. MaxTimeBetweenTweets (temps maximum entre deux tweets)

- Cette caractéristique est utile pour détecter des comportements irréguliers, tels qu'une longue inactivité suivie d'une soudaine avalanche de tweets.
- Un compte pollueur peut rester inactif pendant une période, puis envoyer massivement des tweets pour envahir la plateforme.
- Ce comportement est moins courant chez un utilisateur légitime qui, malgré des variations, tweete généralement de manière plus gradée et constante dans le temps.

Ces deux variables offrent une vision ***fine*** de la dynamique de publication d'un compte, ce qui est essentiel pour identifier les comptes suspects. Elles complètent utilement les informations purement quantitatives (nombre de followers, nombre de tweets…) en révélant l'aspect temporel du comportement, souvent révélateur d'activités malveillantes ou automatisées.

2. Prétraitement des données

J'avais appliqué plusieurs transformations pour améliorer la qualité des données avant de les utiliser. Pour cette partie je vais expliquer le rôle de chaque classe :

Extraction des Caractéristiques

L'extraction a été effectuée par la classe `FeatureExtractor` dans `feature.py`.

Classe **`FeatureExtractor`**

Rôle :

- Charger et fusionner les fichiers de données (content_polluters.txt, legitimate_users.txt, etc.).
- Extraire des caractéristiques à partir des profils et tweets des utilisateurs.
- Sauvegarder les résultats sous forme de fichiers CSV dans **`Datatest/Tache2/Partie1/`**.

Méthodes principales :

`extract_basic_features()` → Calcule des métriques de base comme la longueur du nom, la durée de vie du compte, le nombre de **followings/followers**.

`extract_tweets_per_day()` → Calcule le nombre moyen de tweets par jour.

`extract_url_ratio()` → Détecte les liens dans les tweets et calcule le ratio d'URL.

`extract_mentions_ratio()` → Identifie le ratio de mentions (@) dans les tweets.

`extract_time_between_tweets()` → Mesure les intervalles de temps entre les tweets.

`extract_hashtag_ratio()` → Analyse le taux d'utilisation des hashtags.

`extract_follow_back_ratio()` → Calcule le ratio FollowBack (Nombre de followers / Nombre de followings).

Caractéristiques générées

1. `LengthOfScreenName` : Nombre de caractères dans le nom d'utilisateur.
2. `LengthOfDescriptionInUserProfile` : Nombre de caractères dans la bio.
3. `AccountLongevity` : Nombre de jours entre la création et la collecte.
4. `NumerOfFollowings` : Nombre de comptes suivis.
5. `NumberOfFollowers` : Nombre d'abonnés.
6. `RatioFollowingFollowers` : Ratio entre followings et followers.
7. `TweetsPerDay` : Fréquence moyenne de tweets.
8. `URLRatio` : Proportion de tweets contenant des liens.
9. `MentionRatio` : Ratio de mentions (@) par tweet.
10. `HashtagRatio` : Ratio de hashtags (#) par tweet.
11. `FollowBackRatio` : Ratio abonnés / followings.

Caractéristiques supplémentaires

12. `MeanTimeBetweenTweets` : Temps moyen entre deux tweets.
13. `MaxTimeBetweenTweets` : Temps maximum entre deux tweets consécutifs.

Ces caractéristiques permettent d'analyser l'activité et d'identifier des modèles de comportement atypiques.

3. Prétraitement des Données

Le prétraitement a été effectué via la classe **`DataPreprocessor`** dans **`preprocessing.py`**.

Classe **`DataPreprocessor`**

Rôle :

- Vérifier la qualité des données.
- Supprimer les valeurs aberrantes et gérer les valeurs manquantes.
- Préparer les données pour la normalisation.

Méthodes principales :

- `remove_duplicates()` → Supprime les doublons.
- `handle_missing_values()` → Remplace les valeurs manquantes par la médiane.

4. Transformation et Normalisation

La normalisation et la transformation des valeurs ont été appliquées via **`DataPreparation`** dans **`data_preparation.py`**.

Classe `DataPreparation`

Rôle :

- Appliquer une normalisation aux variables pour éviter les biais.
- Limiter l'influence des valeurs extrêmes.
- Garantir une cohérence dans l'échelle des variables.

Méthodes principales :

- `normalize_data()` → Applique les transformations de normalisation (**`Min-Max Scaling`**, transformation logarithmique).
- `apply_z_score_capping()` → Limite l'influence des valeurs aberrantes avec le **`Z-Score`**.

Normalisation appliquée

i. Min-Max Scaling (échelle entre 0 et 1) → Préserve la distribution des données

- LengthOfScreenName
- LengthOfDescriptionInUserProfile
- MentionRatio

ii. Transformation logarithmique (réduit l'impact des écarts extrêmes)

- AccountLongevity, NumerOfFollowings, NumberOfFollowers
- TweetsPerDay, URLRatio, MeanTimeBetweenTweets
- MaxTimeBetweenTweets, HashtagRatio, FollowBackRatio

iii. Traitement des valeurs extrêmes avec Z-Score

- RatioFollowingFollowers

5. Fusion et Finalisation des Données

La structuration finale des données a été réalisée avec **`DataFinalizer`** dans **`data_final.py`**.

Classe **`DataFinalizer`**

Rôle :

- S'assurer de la conformité des fichiers finaux.
- Ajouter une colonne Classe (1 = Pollueur, 0 = Légitime).
- Fusionner et exporter les données prêtes à être utilisées.

Méthodes principales :

- `finalize_dataset()` → Vérifie la cohérence des données et repositionne la colonne Classe.
- `export_final_dataset()` → Sauvegarde les données finales dans **`Datatest/final_dataset.csv`**.

Fichier final généré

- **`Datatest/final_dataset.csv`** → Contient les données nettoyées et transformées.

Conclusion

1. `FeatureExtractor` → Extraction des caractéristiques
2. `DataPreprocessor` → Nettoyage des données
3. `DataPreparation` → Normalisation et transformation
4. `DataFinalizer` → Fusion et validation finale

Traitement des valeurs manquantes

J'ai remplacé les valeurs manquantes par **la médiane** de chaque colonne. J'ai choisi la médiane car elle est plus robuste aux valeurs extrêmes que la moyenne.

Gestion des valeurs extrêmes

Au départ, j'avais utilisé le **99e percentile** pour limiter les valeurs extrêmes. Mais je n'ai pas réussi à limiter les valeurs au 99e centile, et donc finalement j'ai utilisé la méthode Z-score « **Z-Score (3σ) »**, qui est plus précis et permet de mieux identifier les anomalies, et ça marché directement.

C. Tâche 3 : Analyse comparative

I. Description des étapes clés

Dans cette section, j'ai suivi une méthodologie structurée pour comparer les performances de six algorithmes d'apprentissage supervisé : **`Decision Tree`**, **`Bagging`**, **`AdaBoost`**, **`Gradient Boosting`**, **`Random Forest`**, **`et Naive Bayes`**. Voici les étapes clés effectuées :

---

(1) Préparation des données :

Les données finales ont été divisées en deux ensembles :

- **`80 % pour l'entraînement`**
- **`20 % pour les tests`**.

Cette séparation garantit que l'entraînement des modèles se fait sur des données distinctes de celles utilisées pour l'évaluation, **réduisant ainsi les biais**.

---

(2) Entraînement des modèles :

Chaque algorithme ***a été encapsulé dans une classe dédiée*** pour assurer une implémentation modulaire et claire.
Les modèles ont été entraînés sur l'ensemble d'entraînement en utilisant leurs paramètres par défaut, sauf indication contraire (par exemple, **la graine aléatoire fixée à 42 pour garantir la reproductibilité**).
Une **validation croisée** a été appliquée pour mesurer la robustesse des modèles avant l'évaluation finale.

---

(3) Évaluation des modèles :

Les modèles ont été évalués sur l'ensemble de test en utilisant les métriques demandées :

`A) taux de vrais positifs (TP Rate)`
`B) taux de faux positifs (FP Rate)`
`C) F-mesure (F1-score)`
`D) Aire sous la courbe ROC (AUC).`

Les prédictions des modèles ont été comparées aux valeurs réelles de la classe cible pour calculer ces métriques.

---

(4) Présentation des résultats :

Les résultats ont été présentés sous **deux formes** :

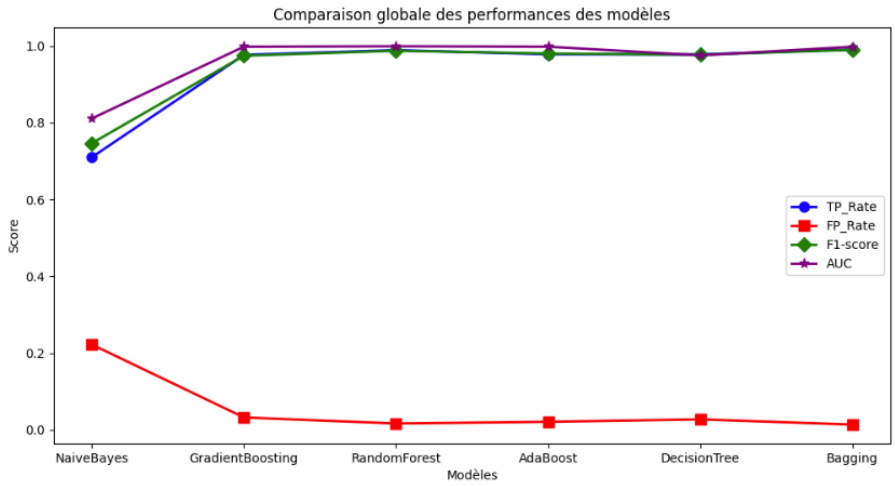
a. **`Graphique`** barres comparant les performances des algorithmes sur les différentes métriques.
b. **`Tableau récapitulatif`** détaillant les scores pour chaque métrique et chaque modèle, sauvegardé en format PNG pour inclusion dans le rapport final.

II. Affichage des résultats

- Les résultats ont été organisés pour maximiser leur clarté et éviter toute ambiguïté. Toutes les valeurs sont enregistrées à 16 chiffres après la virgule. Les points suivants ont été respectés :

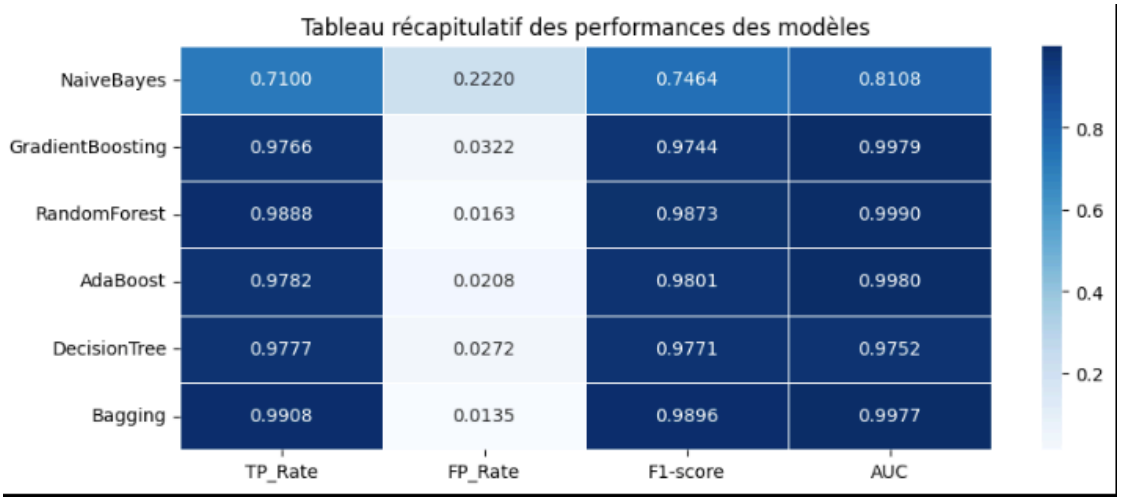
1. Graphique :

Un graphique montre les performances des six algorithmes selon les quatre métriques demandées.

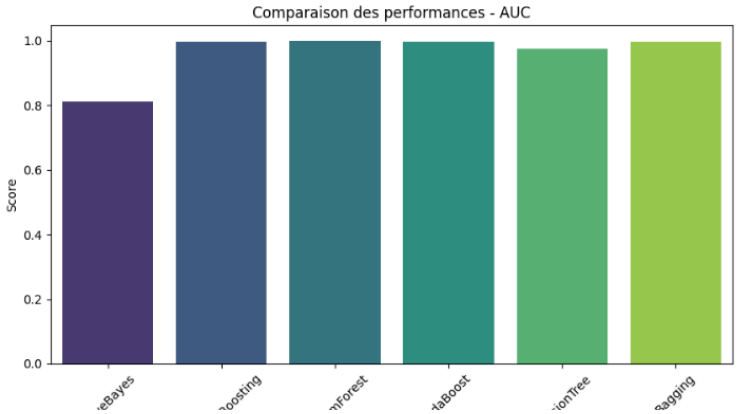


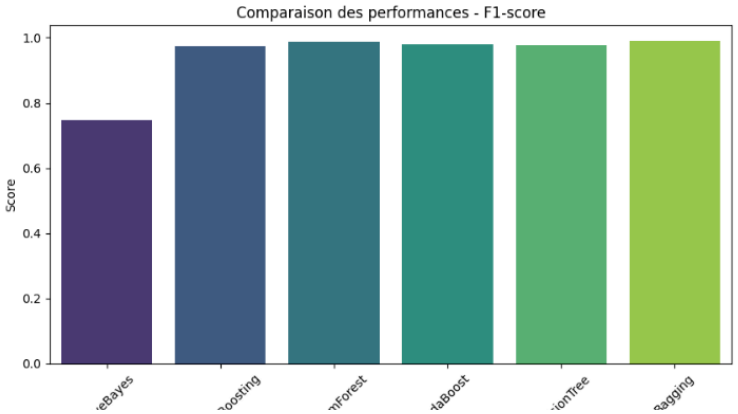
2. Tableau :

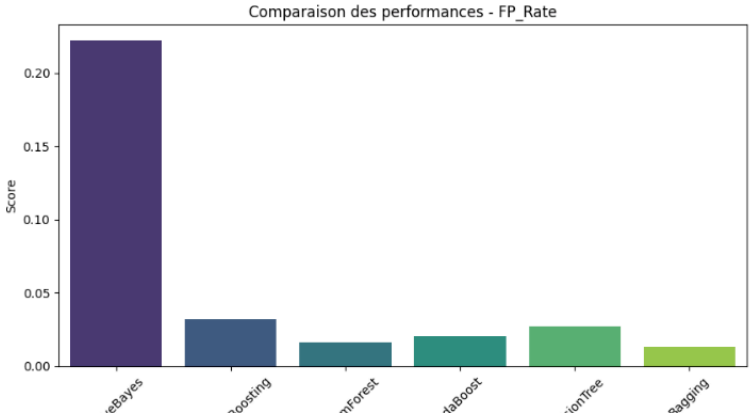
Les résultats quantitatifs ont été sauvegardés sous forme d'un tableau PNG. Ce tableau inclut un titre explicite, des couleurs pour améliorer la lisibilité, et des valeurs précises pour chaque métrique.

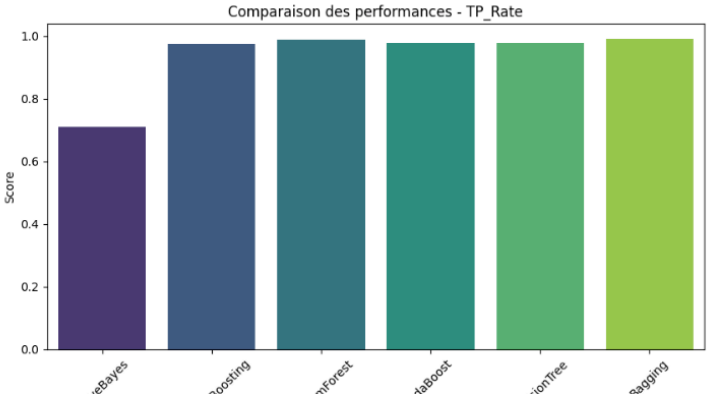


Des autres graphes :









III. Analyse des résultats

1. Performance globale

Modèles d'ensemble : une domination claire

- Les algorithmes d'ensemble tels que **`DecesionTree, Bagging`**, **`AdaBoost`**, **`Gradient Boosting`**, et **`Random Forest`** offrent des performances remarquables :

AUC élevées (≈0.91 à 0.99) : Cela indique une excellente capacité à différencier les classes "pollueurs" et "légitimes".

`F-mesures élevées (≥0.82)` : Ces modèles maintiennent un équilibre entre précision (réduction des fausses alertes) et rappel (capturer les pollueurs réels).

Ces performances s'expliquent par leur capacité à :

+ Combiner plusieurs apprenants faibles pour réduire le biais et la variance.

+ Exploiter des relations complexes dans les données, grâce à des structures telles que les **`arbres de décision`** et des mécanismes comme la pondération des erreurs (**`AdaBoost`**) ou les ajustements progressifs (**`Gradient Boosting`**).

- Naive Bayes : Une performance limitée

Le modèle **`Naive Bayes`**, avec une **`F-mesure`** de 0.49, est significativement en retrait. Cela est dû à son hypothèse forte selon laquelle les caractéristiques sont indépendantes, une simplification rarement vérifiée dans les données réelles. Bien qu'il soit rapide et efficace sur des jeux de données simples, son incapacité à capturer des dépendances entre les variables réduit son efficacité sur ce problème. Et avec FP_Rate est presque null résume notre conclusion.

2. Points forts des algorithmes

Random Forest :

- ***AUC la plus élevée (0.9977)*** : Ce modèle excelle grâce à sa capacité à combiner plusieurs arbres de décision, chacun formé sur un échantillon aléatoire des données (**`bagging`**) et des caractéristiques.
- ***Robustesse contre le surapprentissage*** : L'utilisation d'arbres non corrélés réduit les risques d'ajustement excessif aux données d'entraînement.
- ***Simplicité des hyperparamètres*** : Comparé à **`Gradient Boosting`**, il est moins sensible aux choix d'hyperparamètres, rendant son optimisation plus intuitive.

Gradient Boosting :

- ***Performance proche du Random Forest (AUC ≈ 0.9964) :*** Grâce à son approche incrémentale qui ajuste les erreurs des modèles précédents, il capture efficacement des relations subtiles dans les données.
- ***Adaptation aux données déséquilibrées*** : **`Gradient Boosting`** est particulièrement efficace dans les scénarios où les classes sont déséquilibrées, ce qui est souvent le cas avec les données de pollueurs. (**`FP Rate`**)

- **Bagging et AdaBoost :**

Ces deux modèles offrent une forte robustesse avec des taux de faux positifs relativement faibles. Bagging se concentre sur la réduction de la variance en entraînant des modèles indépendants. **`AdaBoost`**, en revanche, utilise des pondérations pour se concentrer sur les exemples difficiles, ce qui le rend utile lorsque des pollueurs difficiles à détecter sont présents.

- **Decesion Tree :**

Adaboost, AUC est la valeur la plus grande pour tous les modèles.

---

3. Faiblesses des algorithmes

- ***Naive Bayes :***

Sa faiblesse principale est son hypothèse d'indépendance conditionnelle des caractéristiques, qui est rarement vérifiée dans des données complexes.
Par exemple, dans le cas des tweets, des caractéristiques comme la présence d'URL, de mentions, et de hashtags sont souvent corrélées, ce qui échappe à Naive Bayes.

---

4. Considérations spécifiques

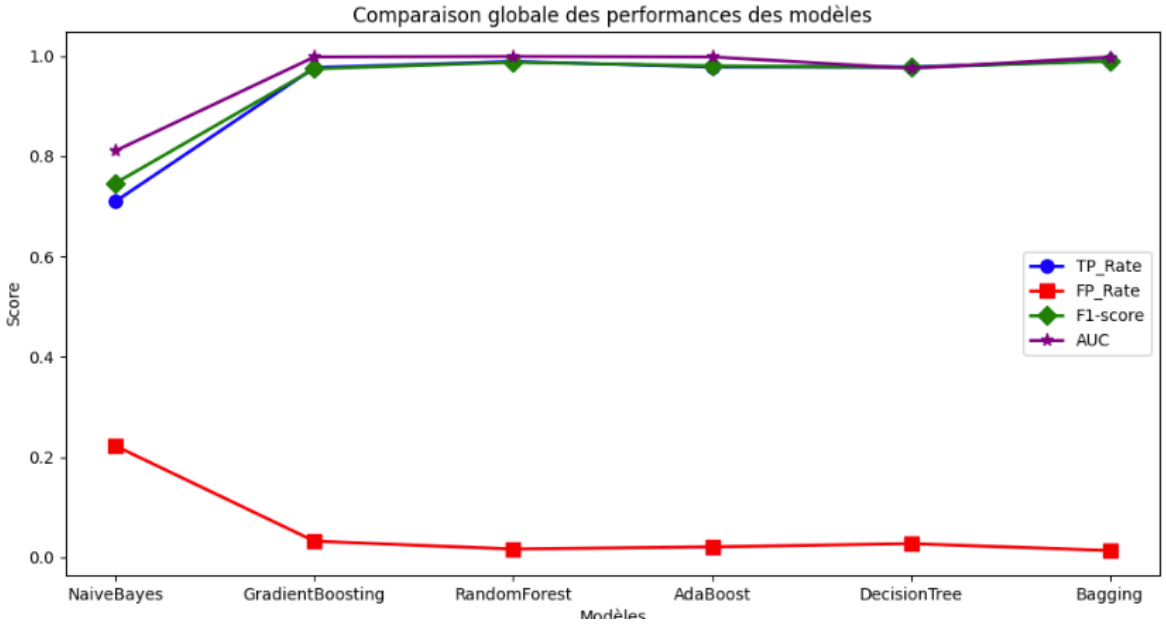
- ***Modèles basés sur des arbres (Random Forest et Gradient Boosting) :***
  - **Sensibilité au bruit :** Ces modèles peuvent parfois être influencés par des anomalies ou du bruit dans les données. Cependant, des mécanismes comme la régularisation dans **`Gradient Boosting`** et la sélection aléatoire des caractéristiques dans **`Random Forest`** atténuent ces problèmes.
  - **Temps de calcul :** Ces modèles sont plus lents à entraîner, notamment sur de grandes données, en raison de leur complexité.
- ***Naive Bayes :***

Ce modèle pourrait être amélioré en explorant des variantes comme :

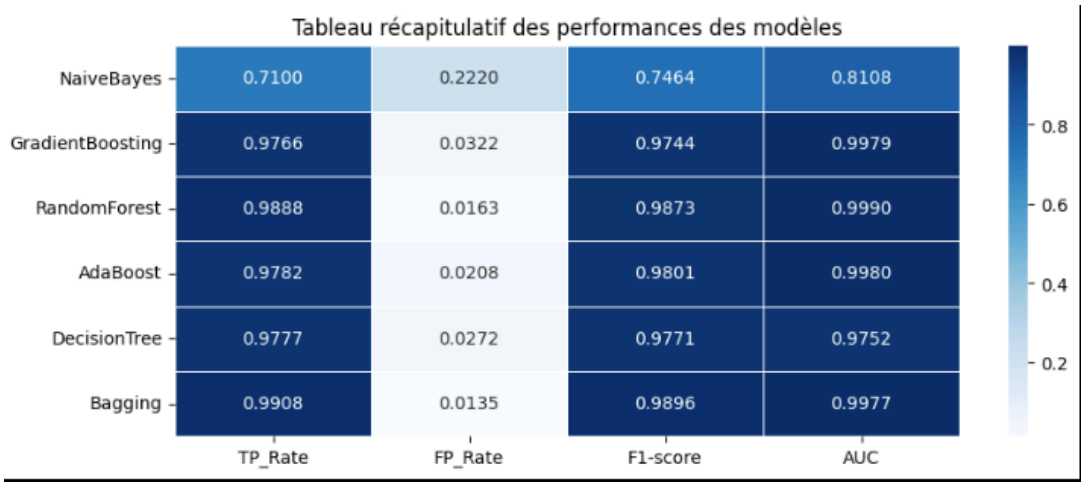
  - **Naive Bayes multinomial :** Plus adapté pour les données textuelles (exemple : fréquence des mots dans les tweets).
  - **Transformation des données :** Des transformations comme la normalisation ou l'application de techniques comme PCA (analyse en composantes principales) pourraient réduire les corrélations entre caractéristiques et améliorer sa performance.

6. Résultats

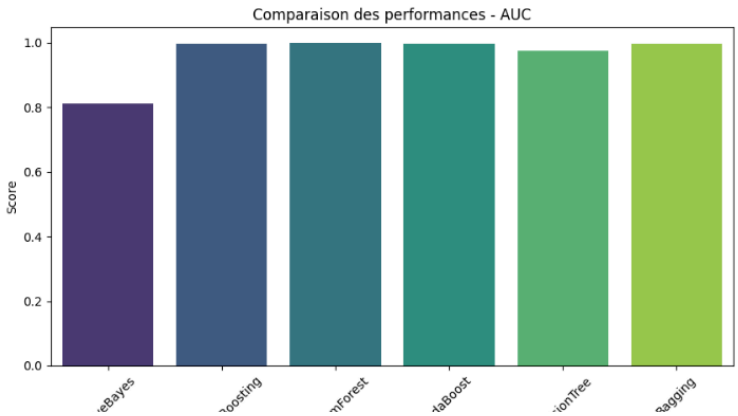
A- Graphe

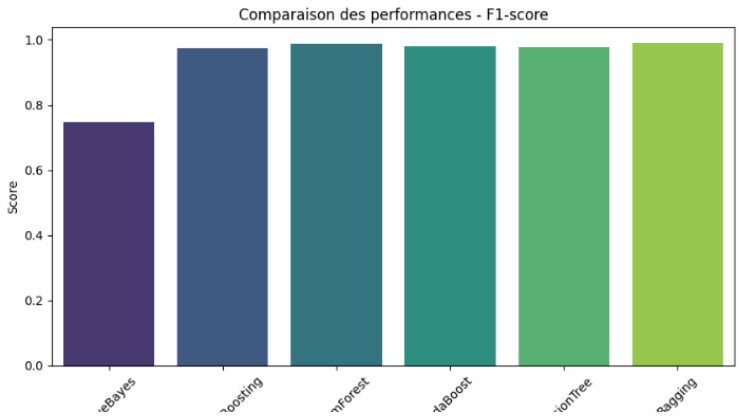


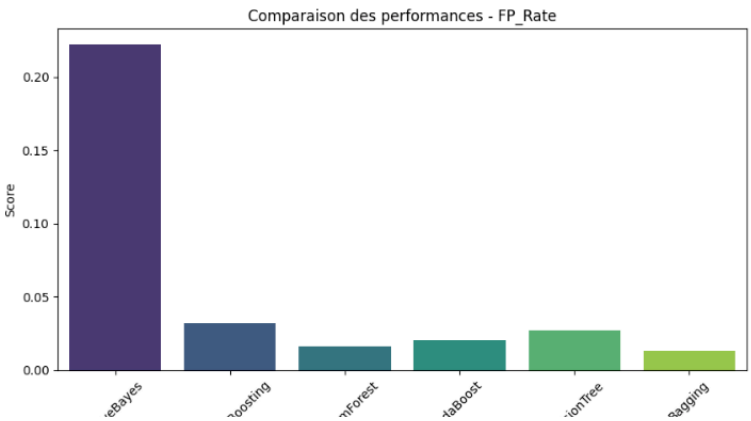
B- Tableau

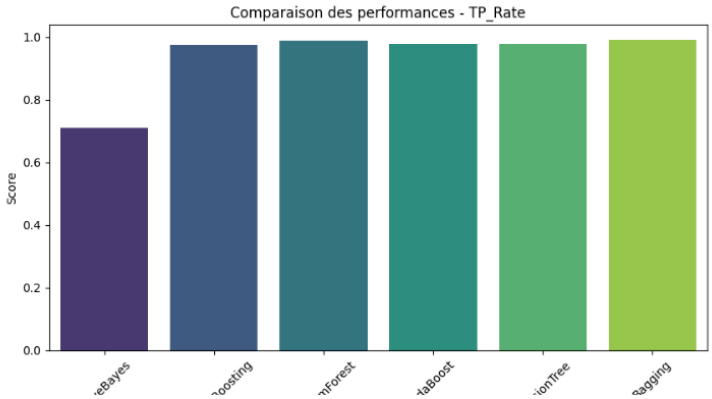


Des autres graphes :









D. Tâche 4 : Analyse comparative sur des classes déséquilibrées

Dans cette section, j'ai poursuivi l'analyse comparative des mêmes algorithmes que dans la Tâche 3, mais cette fois-ci sur des données artificiellement déséquilibrées, où les pollueurs représentent environ 5 % du nombre d'utilisateurs légitimes. L'objectif est de mesurer l'impact de ce déséquilibre sur les performances de chaque modèle et de comparer ces résultats avec ceux obtenus sur l'ensemble équilibré.

---

1. Création du sous-ensemble déséquilibré

- **5 %**
  J'ai prélevé un échantillon de la classe majoritaire (**`utilisateurs légitimes`**) et un nombre de pollueurs représentant 5 % de ce nombre.
  - *Par exemple*, si l'on conserve 1000 utilisateurs légitimes, on n'en garde que 50 de la classe pollueur, de sorte que le déséquilibre soit nettement ressenti.
- **Mise à jour des données**
  Le reste de la préparation (extraction des caractéristiques, normalisation, etc.) **est identique à la Tâche 2**. Seule la proportion de pollueurs a été modifiée.

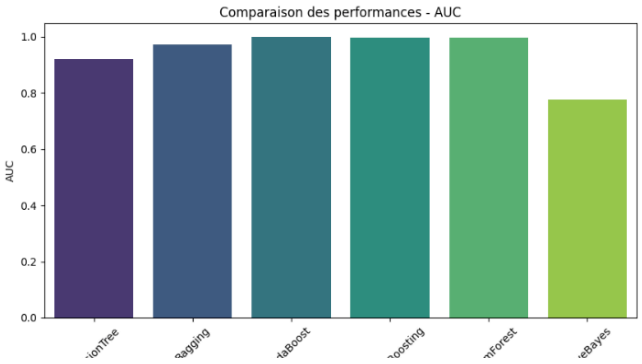
---

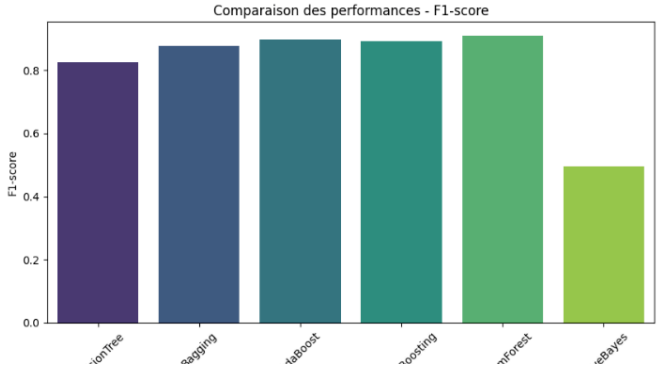
2. Entraînement et évaluation

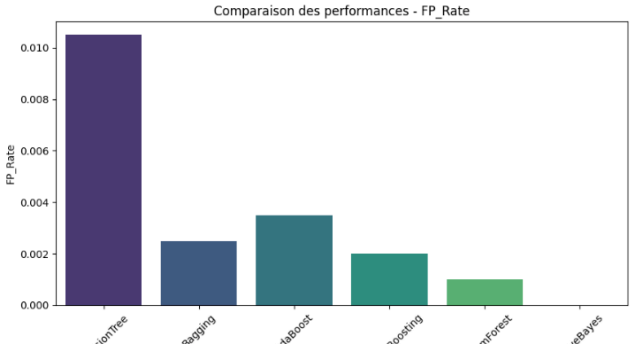
J'ai réutilisé la même procédure de Tâche 3 :

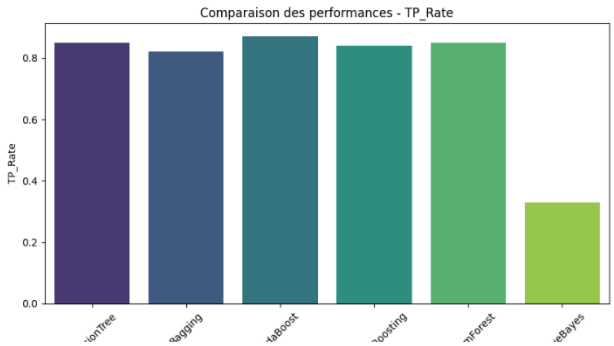
- **Séparation** : 80 % du sous-ensemble déséquilibré pour l'entraînement et 20 % pour le test.
- **Modèles évalués :**
  - **`Decision Tree`**
  - **`Bagging`**
  - **`AdaBoost`**
  - **`Gradient Boosting`**
  - **`Random Forest`**
  - **`Naive Bayes`**
- **Métriques :**
  - Taux de vrais positifs (**`TP Rate`**) pour les pollueurs,
  - Taux de faux positifs (**`FP Rate`**) pour les pollueurs aussi,
  - **`F1-score`** pour la classe pollueur aussi, et
  - **`AUC`**.
- **Comparaison :** J'avais compilé les résultats dans un fichier nommé « **`imbalanced_data.csv`** » et généré des graphiques comparatifs **(`AUC, F1-score, TP Rate, FP Rate`)** en confrontant les performances sur données équilibrées et déséquilibrées.

3. Résultats sur données déséquilibrées

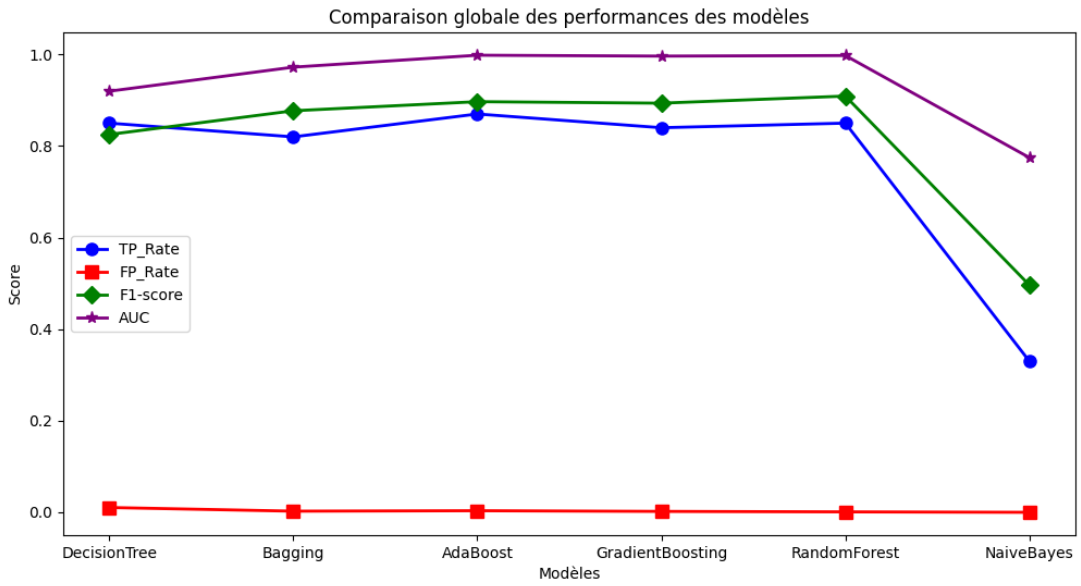








	TP_Rate	FP_Rate	F1-score	AUC
DecisionTree	0.85	0.0105	0.8252	0.9198
Bagging	0.82	0.0025	0.877	0.9723
AdaBoost	0.87	0.0035	0.8969	0.9984
GradientBoosting	0.84	0.002	0.8936	0.9964
RandomForest	0.85	0.001	0.9091	0.9977
NaiveBayes	0.33	0.0	0.4962	0.7747



Dans un premier temps, nous avons appliqué les mêmes six modèles de classification que dans la Tâche 3 sur l'ensemble de données déséquilibré :

- **Arbre de décision (Decision Tree)**
- **Bagging**
- **AdaBoost**
- **Gradient Boosting**
- **Forêt d'arbres aléatoires (Random Forest)**
- **Classification bayésienne naïve (Naive Bayes)**

La séparation entre **train (80%)** et **test (20%)** a été conservée afin d'avoir une base de comparaison cohérente avec les résultats précédents.

Les métriques d'évaluation utilisées sont :

- Taux de vrais positifs (**TP Rate**) : capacité du modèle à détecter correctement les pollueurs.
- Taux de faux positifs (**FP Rate**) : proportion d'utilisateurs légitimes incorrectement classés comme pollueurs.
- F-mesure (**F1-score**) : mesure d'équilibre entre la précision et le rappel des pollueurs.
- Aire sous la courbe ROC (**AUC**) : capacité globale du modèle à distinguer les classes.

Résultats des modèles sur données déséquilibrées

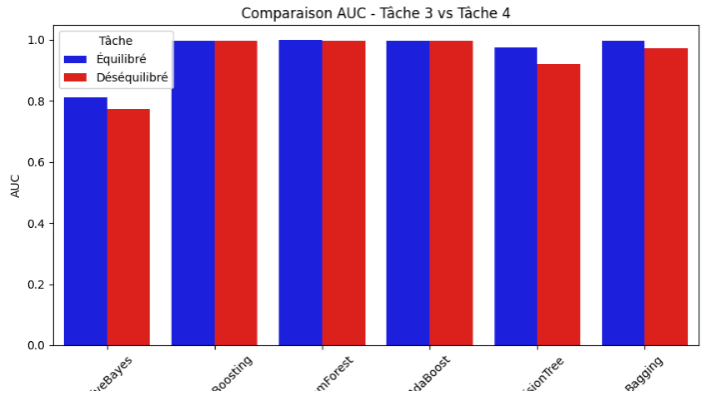
Les performances des modèles sur l'ensemble déséquilibré sont présentées dans l'image ci-dessous :

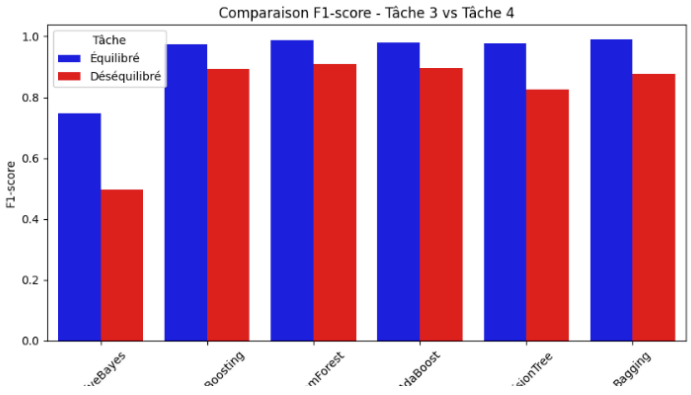
Observations principales :

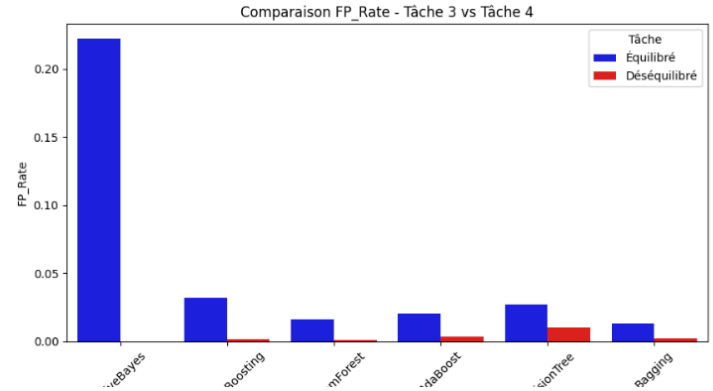
- **AdaBoost** et **Random Forest** restent les plus performants, avec un **AUC** supérieur à 0.99, ce qui signifie qu'ils arrivent encore à bien différencier les pollueurs des utilisateurs légitimes.
- Le modèle **Naïve Bayes** est très fortement impacté par le déséquilibre, avec un TP Rate de seulement 33 % et un F1-score de 0.49, confirmant son inadaptation aux données déséquilibrées.
- Le **TP Rate** des autres modèles baisse légèrement par rapport à la Tâche 3, car les pollueurs sont devenus rares dans l'ensemble d'apprentissage.
- Le **FP Rate** diminue pour tous les modèles, ce qui montre que moins d'utilisateurs légitimes sont classés à tort comme pollueurs. Toutefois, ce phénomène peut aussi être un signe que les modèles deviennent trop prudents et hésitent à classer un utilisateur comme pollueur.

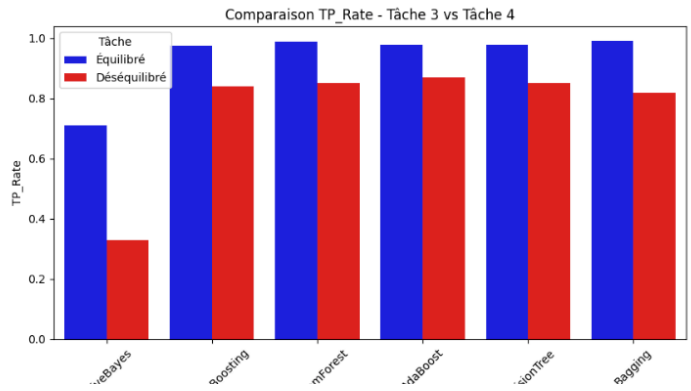
Comparaison entre les modèles :

Et voici les graphes, tableaux ci-dessous donnent un aperçu chiffré des performances enregistré est généré dans le dossier Datatest/Comparaison_Tache3_Tache4 :

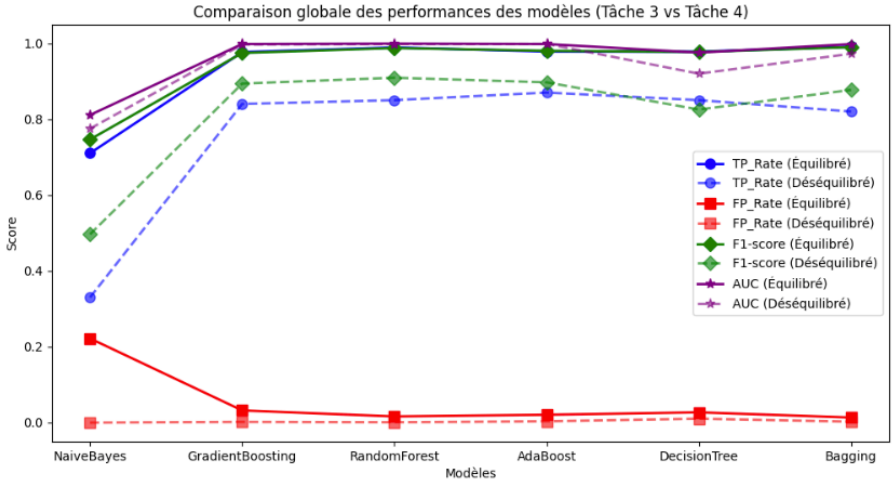








Model	TP_Rate_Tache3	FP_Rate_Tache3	F1-score_Tache3	AUC_Tache3	TP_Rate_Tache4	FP_Rate_Tache4	F1-score_Tache4	AUC_Tache4
NaiveBayes	0.71	0.222	0.7464	0.8108	0.33	0.0	0.4962	0.7747
GradientBoostin	0.9766	0.0322	0.9744	0.9979	0.84	0.002	0.8936	0.9964
RandomForest	0.9888	0.0163	0.9873	0.999	0.85	0.001	0.9091	0.9977
AdaBoost	0.9782	0.0208	0.9801	0.998	0.87	0.0035	0.8969	0.9984
DecisionTree	0.9777	0.0272	0.9771	0.9752	0.85	0.0105	0.8252	0.9198
Bagging	0.9908	0.0135	0.9896	0.9977	0.82	0.0025	0.877	0.9723



Après avoir observé les résultats sur données déséquilibrées, nous avons comparé ces performances à celles obtenues avec les données équilibrées de la Tâche 3.
Les résultats sont synthétisés sous forme de graphiques comparatifs pour mieux visualiser l'impact du déséquilibre.

- **Impact sur le taux de vrais positifs (TP Rate)**

+ Avec des données équilibrées (Tâche 3), la plupart des modèles atteignaient un TP Rate supérieur à 85 %, ce qui signifie qu'ils détectaient bien les pollueurs.
+ Avec des données déséquilibrées (Tâche 4), le TP Rate baisse légèrement pour la majorité des modèles, mais chute drastiquement pour Naïve Bayes.
—> Cela montre que les modèles sont moins performants pour détecter les pollueurs quand ces derniers sont sous-représentés dans l'apprentissage.

- **Impact sur le taux de faux positifs (FP Rate)**

+ Le FP Rate diminue fortement sur données déséquilibrées, ce qui signifie que moins d'utilisateurs légitimes sont faussement classés comme pollueurs.
* Cependant, cette diminution peut être trompeuse, car cela peut aussi signifier que les modèles deviennent trop conservateurs et prennent moins de risques pour classer un utilisateur comme pollueur.
+ Les modèles Boosting et Random Forest réussissent néanmoins à maintenir un bon équilibre entre réduction du FP Rate et détection des pollueurs.

- **Impact sur le F1-score**

Le F1-score reste stable pour AdaBoost, Gradient Boosting et Random Forest, qui conservent une capacité correcte à détecter les pollueurs même avec des données déséquilibrées.
En revanche, Naïve Bayes voit son F1-score chuter fortement, confirmant son inaptitude à gérer les classes déséquilibrées.

- **Impact sur l'AUC**

Les modèles Boosting et Random Forest maintiennent un AUC proche de 0.99, montrant leur capacité à bien séparer les deux classes, même en situation déséquilibrée.
Naïve Bayes subit une forte baisse d'AUC, démontrant qu'il ne peut pas correctement classer les pollueurs dans ce contexte.

Analyse approfondie et interprétation

Pourquoi certains modèles résistent mieux au déséquilibre ?

Les modèles comme AdaBoost, Gradient Boosting et Random Forest restent performants car :

- Ils effectuent des combinaisons de plusieurs modèles faibles, ce qui leur permet d'être plus robustes face aux classes rares.
- Le Boosting ajuste les erreurs au fur et à mesure, ce qui leur permet de mieux gérer les données déséquilibrées.
- Les forêts aléatoires prennent en compte plusieurs sous-ensembles des données, réduisant ainsi le risque de biais dû au déséquilibre.

Pourquoi Naïve Bayes échoue sur des données déséquilibrées ?

- **`Naïve Bayes`** repose sur des hypothèses de distribution qui ne sont pas adaptées aux classes déséquilibrées.
- Il suppose que chaque classe a une distribution de probabilité fixe, ce qui pose problème quand une classe est sous-représentée.
- Il ne corrige pas les déséquilibres et attribue des probabilités trop faibles aux pollueurs, les classant souvent comme utilisateurs légitimes.

Une diminution du FP Rate signifie-t-elle une meilleure performance ?

- Pas forcément. Une forte diminution du FP Rate peut indiquer que le modèle détecte moins de pollueurs par excès de prudence.
- Un modèle efficace doit maintenir un bon équilibre entre TP Rate et FP Rate pour éviter les erreurs critiques.

Les résultats montrent que :

- AdaBoost, Gradient Boosting et Random Forest sont les meilleurs modèles pour gérer un déséquilibre des classes.
- Naïve Bayes est à éviter, car il ne sait pas bien traiter les classes minoritaires.
- Un faible FP Rate n'est pas toujours un bon indicateur : il faut aussi s'assurer que le TP Rate ne chute pas trop.

Comment améliorer la détection des pollueurs dans un contexte déséquilibré ?

+ Rééquilibrer les classes en ajoutant plus d'exemples de pollueurs ou en réduisant le nombre d'utilisateurs légitimes.
+ Utiliser des algorithmes robustes, comme le Boosting et les forêts aléatoires.
+ Penser aux pondérations : certains modèles permettent d'accorder plus de poids aux pollueurs pour compenser leur rareté.

E. Conclusion

Le travail que j'avais réalisé pour ce TP met en évidence l'importance d'un cycle complet en apprentissage automatique :
+ Extraction, nettoyage et préparation des données (Tâche 2)
+ Comparaison multi-algorithmes sur données équilibrées (Tâche 3)
+ Analyse approfondie en contexte déséquilibré et comparaison des données avec la Tâche 3 (Tâche 4)

Les résultats montrent que :

- **Les modèles d'ensemble** (**`Bagging`**, **`AdaBoost`**, **`Gradient Boosting`**, **`Random Forest`**, **`Decision Tree`**) offrent les performances les plus robustes, aussi bien en termes de **`F1-score`** qu'en **`AUC`**, et conservent une capacité de généralisation intéressante *même lorsque les classes sont fortement déséquilibrées*.

- **Les modèles simples** (**`Naïve Bayes`**) s'avèrent plus sensibles aux déséquilibres de classe, notamment lorsqu'on évalue la détection de la classe minoritaire (les pollueurs).

- **Les valeurs extrêmes**, le bruit et la répartition des données (classe majoritaire vs minoritaire) influent considérablement sur la qualité des prédictions.