Détection de Fraude dans les Transactions Financières

Introduction

La détection des fraudes par carte bancaire est un domaine crucial dans les systèmes financiers modernes. Avec l'augmentation des transactions électroniques, les fraudes deviennent de plus en plus sophistiquées, mettant en péril la sécurité des consommateurs et des entreprises.

Objectif du projet

L'objectif principal de ce projet est de concevoir un système capable d'identifier et de classer les transactions frauduleuses en temps réel à l'aide de techniques de machine learning. Cela permettra :

- De réduire les pertes financières dues aux fraudes.
- De protéger les utilisateurs et de renforcer leur confiance dans les systèmes financiers numériques.

Défis

- Déséquilibre des données : Les transactions frauduleuses représentent généralement une petite fraction des données totales.
- Vitesse et précision : Le système doit détecter les fraudes rapidement sans compromettre la précision.
- Sécurité et confidentialité des données : Protéger les informations sensibles des utilisateurs.

Approche

1. **Analyse exploratoire des données** : Comprendre la distribution et les caractéristiques des données de transactions.
2. **Préparation des données** : Nettoyage, transformation, et gestion de l'équilibre des classes.
3. **Modélisation** : Utilisation d'algorithmes de machine learning pour classifier les transactions.
4. **Évaluation** : Analyse des performances à l'aide de métriques adaptées (par ex. précision, rappel, F1-score).
5. **Déploiement** : Mise en place d'un modèle opérationnel pour des transactions en temps réel.

Résultats attendus

- Un modèle performant et fiable pour détecter les fraudes.
- Une visualisation claire des résultats pour faciliter la prise de décision.

I. Analyse exploratoire des données :

Analyser le dataset pour comprendre sa structure, détecter les problèmes potentiels et identifier les tendances initiales.

```
# Charger le jeu de données
data = pd.read_csv("/content/creditcard.csv")  # Charger les données dans un DataFrame Pandas
print("Aperçu des premières lignes du dataset :\n", data.head())  # Aperçu des premières lignes
```

```
Aperçu des premières lignes du dataset :
   Time        V1        V2        V3        V4        V5        V6        V7  \
0     0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
1     0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
2     1 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
3     1 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
4     2 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

         V8        V9  ...       V21       V22       V23       V24       V25  \
0  0.098698  0.363787  ... -0.018307  0.277838 -0.110474  0.066928  0.128539
1  0.085102 -0.255425  ... -0.225775 -0.638672  0.101288 -0.339846  0.167170
2  0.247676 -1.514654  ...  0.247998  0.771679  0.909412 -0.689281 -0.327642
3  0.377436 -1.387024  ... -0.108300  0.005274 -0.190321 -1.175575  0.647376
4 -0.270533  0.817739  ... -0.009431  0.798278 -0.137458  0.141267 -0.206010

        V26       V27       V28  Amount  Class
0 -0.189115  0.133558 -0.021053  149.62    0.0
1  0.125895 -0.008983  0.014724    2.69    0.0
2 -0.139097 -0.055353 -0.059752  378.66    0.0
3 -0.221929  0.062723  0.061458  123.50    0.0
4  0.502292  0.219422  0.215153   69.99    0.0

[5 rows x 31 columns]
```

Structure des données :

- Il y a **31 colonnes** dans le dataset, incluant 28 variables anonymisées, Time, Amount, et Class.
- Chaque ligne représente une transaction individuelle.

Colonnes principales :

- **Time :** Temps écoulé en secondes depuis une référence initiale.
- **V1 à V28 :** Variables issues d'une réduction de dimension (comme l'Analyse en Composantes Principales, PCA). Ces variables ne sont pas interprétables directement car elles ont été anonymisées pour des raisons de confidentialité.
- **Amount :** Montant de la transaction en unité monétaire.
- **Class :** Étiquette de classe indiquant si la transaction est frauduleuse (1.0) ou non-frauduleuse (0.0).

```
print("\nTaille du dataset (lignes, colonnes) :", data.shape)  # Nombre de lignes et colonnes
print(data.info())  # Types de données et valeurs manquantes
```

```
Taille du dataset (lignes, colonnes) : (103088, 31)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 103088 entries, 0 to 103087
Data columns (total 31 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   Time    103088 non-null  int64
 1   V1      103088 non-null  float64
 2   V2      103088 non-null  float64
 3   V3      103088 non-null  float64
 4   V4      103088 non-null  float64
 5   V5      103088 non-null  float64
 6   V6      103088 non-null  float64
 7   V7      103088 non-null  float64
 8   V8      103088 non-null  float64
 9   V9      103088 non-null  float64
 10  V10     103088 non-null  float64
 11  V11     103088 non-null  float64
 12  V12     103088 non-null  float64
 13  V13     103088 non-null  float64
 14  V14     103088 non-null  float64
 15  V15     103088 non-null  float64
 16  V16     103088 non-null  float64
 17  V17     103088 non-null  float64
 18  V18     103088 non-null  float64
 19  V19     103088 non-null  float64
 20  V20     103088 non-null  float64
 21  V21     103088 non-null  float64
 22  V22     103088 non-null  float64
 23  V23     103088 non-null  float64
 24  V24     103088 non-null  float64
 25  V25     103087 non-null  float64
 26  V26     103087 non-null  float64
 27  V27     103087 non-null  float64
 28  V28     103087 non-null  float64
 29  Amount  103087 non-null  float64
 30  Class   103087 non-null  float64
dtypes: float64(30), int64(1)
memory usage: 24.4 MB
None
```

Taille du dataset :

- Le dataset contient 103,088 lignes et 31 colonnes.
- Chaque ligne correspond à une transaction unique, et chaque colonne correspond à une caractéristique de la transaction.

Colonnes et types de données :

- Parmi les **31 colonnes**, 30 sont de type float64, et une seule (Time) est de type int64.
- Les variables anonymisées (V1 à V28), le montant (Amount) et la classe (Class) sont des valeurs numériques continues ou binaires.

Valeurs non nulles :

- La majorité des colonnes ont 103,088 valeurs non nulles, sauf les colonnes suivantes : V25, V26, V27, V28, Amount, et Class, qui comptent une valeur manquante chacune (103,087 valeurs non nulles).
- Ces valeurs manquantes doivent être traitées dans une étape ultérieure (imputation ou suppression).

Les statistiques descriptives :

- Les statistiques descriptives incluent des mesures comme la moyenne, l'écart type, les quartiles, le minimum et le maximum pour chaque colonne.

```
print("\nStatistiques descriptives :")
print(data.describe())  # Moyenne, écart-type, minimum, maximum, etc.
```

```
Statistiques descriptives :
                Time             V1             V2             V3  \
count  103088.000000  103088.000000  103088.000000  103088.000000
mean    43195.783496      -0.259278      -0.032491       0.684622
std     17339.553909       1.851810       1.650426       1.321775
min         0.000000     -56.407510     -72.715728     -33.680984
25%     33760.000000      -1.026222      -0.600738       0.179380
50%     45003.000000      -0.262181       0.075028       0.759434
75%     56992.500000       1.154372       0.733932       1.386306
max     68489.000000       1.960497      18.902453       4.226108

                V4             V5             V6             V7  \
count  103088.000000  103088.000000  103088.000000  103088.000000
mean        0.156327      -0.283952       0.099404      -0.116461
std         1.346277       1.355148       1.302138       1.217442
min        -5.172595     -42.147898     -26.160506     -31.764946
25%        -0.715776      -0.908043      -0.643339      -0.606811
50%         0.184192      -0.318363      -0.151559      -0.072630
75%         1.027020       0.245974       0.494781       0.410230
max        16.715537      34.801666      22.529298      36.677268
```

```
                V8             V9  ...            V21            V22  \
count  103088.000000  103088.000000  ...  103088.000000  103088.000000
mean        0.056888      -0.046868  ...      -0.028920      -0.104803
std         1.237722       1.115283  ...       0.746371       0.640178
min       -73.216718      -9.283925  ...     -34.830382     -10.933144
25%        -0.135589      -0.694322  ...      -0.223478      -0.529604
50%         0.076816      -0.115372  ...      -0.056063      -0.080240
75%         0.367677       0.556107  ...       0.121638       0.316316
max        20.007208      10.392889  ...      27.202839      10.503090

                V23            V24            V25            V26  \
count  103088.000000  103088.000000  103087.000000  103087.000000
mean       -0.037813       0.009360       0.133760       0.026201
std         0.628632       0.595227       0.440465       0.492544
min       -44.807735      -2.836627     -10.295397      -2.534330
25%        -0.177659      -0.323367      -0.130323      -0.323492
50%        -0.050194       0.065235       0.172269      -0.069541
75%         0.080639       0.406512       0.421853       0.296274
max        19.002942       4.014444       5.541598       3.517346

                V27            V28         Amount          Class
count  103087.000000  103087.000000  103087.000000  103087.000000
mean        0.001784       0.001943      97.037649       0.002251
std         0.391814       0.322462     263.287232       0.047387
min        -9.390980      -9.617915       0.000000       0.000000
25%        -0.061778      -0.005197       7.170000       0.000000
50%         0.010610       0.023358      25.690000       0.000000
75%         0.084605       0.076830      88.000000       0.000000
max        12.152401      33.847808   19656.530000       1.000000
```

Colonnes principales :

- **Time** :
  - Moyenne : 43,195 secondes (~12 heures).
  - Écart-type : 17,339 secondes (~4.8 heures).
  - Min : 0 seconde (première transaction).
  - Max : 68,489 secondes (~19 heures).

⇨ Interprétation : La colonne Time représente le temps écoulé depuis le début de la collecte des données. Les transactions s'étendent sur environ 19 heures.

- **Amount** :
  - Moyenne : **97.03 unités monétaires**, mais la médiane est de **25.69 unités**, ce qui indique une distribution asymétrique (fortement influencée par des montants élevés).
  - Écart-type : **263.29**, ce qui reflète une grande variabilité dans les montants.
  - Max : **19,656.53 unités**, avec un minimum de 0.

⇨ Interprétation : La majorité des transactions impliquent des montants faibles, mais il existe des montants exceptionnellement élevés (potentiellement intéressants pour la détection de fraudes).

- **Class** :
  - Moyenne : **0.0023**, ce qui reflète une proportion très faible de transactions frauduleuses.
  - Max : 1 (indique les transactions frauduleuses).

⇨ Distribution : Les quartiles montrent que 75 % des transactions sont non frauduleuses, et les fraudes représentent une très petite partie du dataset.

Colonnes anonymisées (V1 à V28) :

- Les variables semblent centrées autour de 0 pour la plupart, avec des moyennes proches de zéro.
- Les écarts-types varient, mais certaines variables montrent des valeurs extrêmes, comme :
  - **V1** : Min de **-56.40**, Max de **1.96**.
  - **V2** : Min de **-72.71**, Max de **18.90**.
  - **V8** : Min de **-73.21**, Max de **20.00**.

⇨ Interprétation : Ces valeurs peuvent indiquer des transactions atypiques ou des anomalies potentiellement liées à la fraude.

Les valeurs manquantes :

```
missing_values = data.isnull().sum()  # Compter les valeurs manquantes par colonne
print("\nValeurs manquantes par colonne :\n", missing_values)
```

```
Valeurs manquantes par colonne :
 Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       1
V26       1
V27       1
V28       1
Amount    1
Class     1
dtype: int64
```

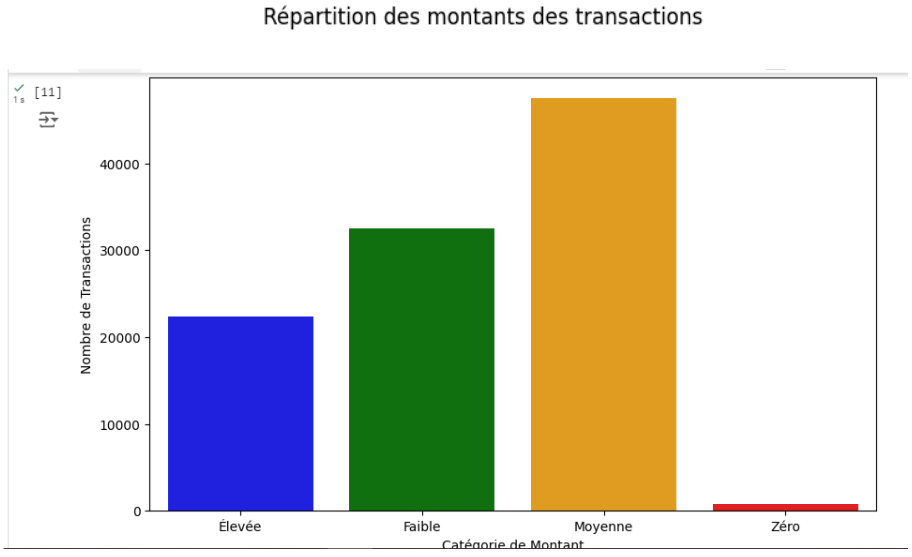
- Les colonnes V25, V26, V27, V28, Amount et Class contiennent chacune 1 valeur manquante.
- Les autres colonnes (Time, V1 à V24) ne présentent aucune valeur manquante.

les statistiques descriptives des montants (Amount) :

```
# Statistiques descriptives pour la colonne 'Amount'
print(data['Amount'].describe())
```

```
count    103087.000000
mean         97.037649
std         263.287232
min           0.000000
25%           7.170000
50%          25.690000
75%          88.000000
max       19656.530000
Name: Amount, dtype: float64
```

- **Nombre de valeurs (count)** : 103,087 transactions.
- **Moyenne (mean)** : 97,04, ce qui représente la valeur moyenne des montants dans les transactions.
- **Écart-type (std)** : 263,29, indiquant une forte dispersion des montants autour de la moyenne.
- **Valeurs extrêmes** :
  - **Minimum (min)** : 0, ce qui signifie qu'il existe des transactions avec un montant nul.
  - **Maximum (max)** : 19,656.53, indiquant une transaction avec un montant très élevé.
- **Quartiles** :
  - **1er quartile (25%)** : 7,17. Cela signifie que 25 % des transactions ont un montant inférieur ou égal à 7,17.
  - **Médiane (50%)** : 25,69. La moitié des transactions ont un montant inférieur ou égal à cette valeur.
  - **3e quartile (75%)** : 88,00. Cela signifie que 75 % des transactions ont un montant inférieur ou égal à 88,00.



Analyse des catégories de montants :

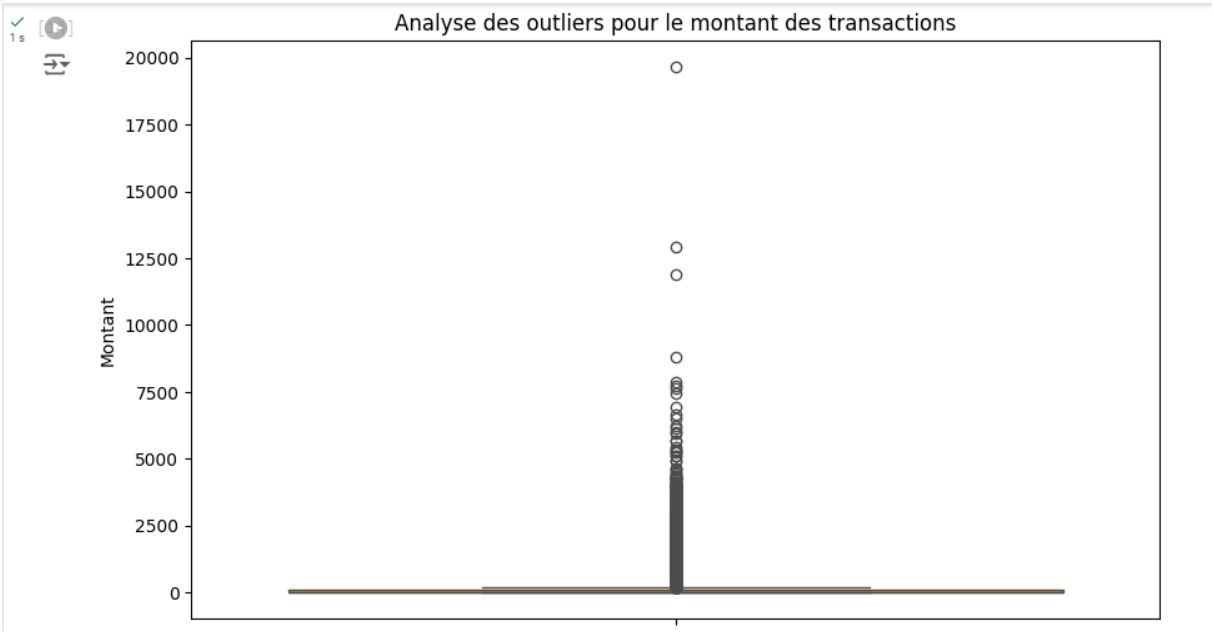
1. **Catégorie "Moyenne" :**
   - La catégorie Moyenne domine en nombre de transactions, ce qui indique que la majorité des transactions se situent dans une plage de montants intermédiaires.
   - Cela reflète probablement une tendance commune où la majorité des montants sont modérés, sans excès.
2. **Catégorie "Faible" :**
   - La deuxième catégorie la plus représentée est Faible, ce qui est également attendu dans des données transactionnelles réelles, où de nombreuses transactions concernent de petits montants.
3. **Catégorie "Élevée" :**
   - Les transactions à montant Élevé sont en quantité moindre que les précédentes, ce qui est normal car les transactions importantes sont souvent moins fréquentes.
4. **Catégorie "Zéro" :**
   - La catégorie Zéro a très peu d'occurrences, mais cela peut être important à explorer. Ces transactions pourraient refléter :
     - Des anomalies ou erreurs dans les données.
     - Des transactions effectivement gratuites (par exemple, des tests ou des frais annulés).

Matrice de correlation :

Une matrice de corrélation montre la force et la direction de la relation linéaire entre les différentes variables du dataset. Les valeurs varient entre **-1** (corrélation négative parfaite) et **1** (corrélation positive parfaite), avec **0** indiquant l'absence de corrélation.

La boîte à moustaches pour l'analyse des outliers :

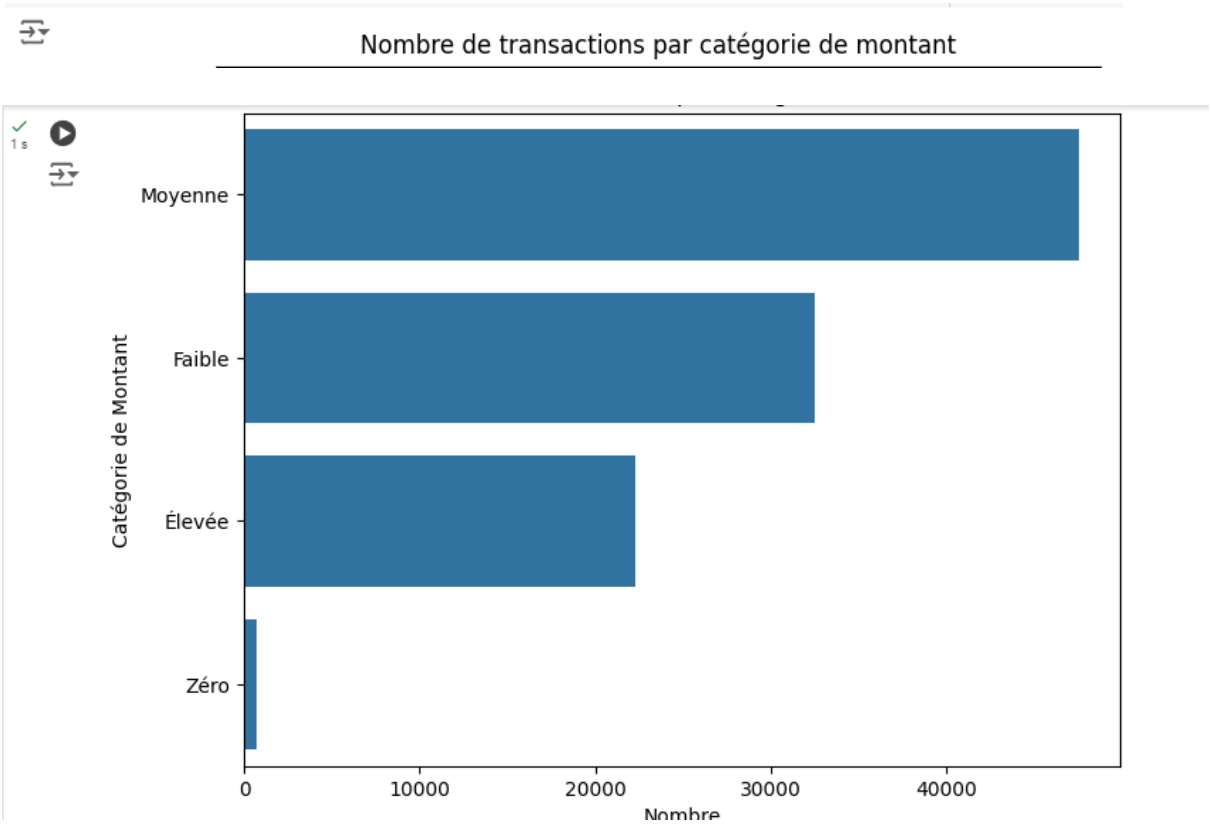
La boîte à moustaches pour l'analyse des outliers (valeurs extrêmes) dans les montants des transactions montre une concentration de la majorité des montants près de zéro, tandis que des valeurs extrêmes très élevées sont clairement identifiées.



- **Concentration des données principales :**
    - La majorité des transactions se trouvent près de la médiane, avec une faible variation. Cela confirme que la plupart des transactions ont des montants faibles ou modérés.
    - Les limites des moustaches indiquent les seuils au-delà desquels les valeurs sont considérées comme des outliers.
- **Présence de valeurs extrêmes :**
    - Les cercles au-dessus des moustaches représentent des transactions considérées comme des valeurs extrêmes. Ces montants sont significativement supérieurs à la majorité des données.
    - Cela inclut des montants proches de 20 000, qui sont rares.
- **Asymétrie :**
    - La boîte est allongée vers le haut (asymétrie positive), indiquant que les montants élevés influencent la distribution, bien qu'ils soient rares.

⇨ Les valeurs extrêmes pourraient correspondre à des transactions spécifiques, comme des achats importants ou des anomalies.
⇨ Dans un contexte de détection de fraude, ces valeurs méritent une analyse approfondie, car les transactions avec des montants exceptionnellement élevés peuvent présenter un risque accru de fraude.

Vérification des transactions par catégorie :



La visualisation des transactions par catégorie de montant permet d'observer comment les transactions sont réparties en fonction des différentes catégories de montant (par exemple, "Faible", "Moyenne", "Élevée", "Zéro").

1. **Catégorie prédominante :**
   - Une catégorie de montant spécifique ("Moyenne") semble avoir le plus grand nombre de transactions, indiquant que la plupart des montants tombent dans cette gamme.
   - La catégorie "Zéro" est minoritaire, ce qui est logique, car peu de transactions ont un montant nul.
2. **Équilibre des catégories :**
   - La répartition des transactions dans les autres catégories ("Faible", "Élevée") est également visible, mais elles sont moins nombreuses que la catégorie "Moyenne".
3. **Échelle et proportions :**
   - Les transactions de faible et moyenne valeur sont dominantes, ce qui correspond à la nature de nombreuses bases de données transactionnelles, où les petites transactions sont plus fréquentes.
   - Les montants élevés, bien que rares, sont une catégorie notable, souvent pertinente dans les analyses liées à la fraude.

⇨ Les transactions avec un montant nul ("Zéro") pourraient être des anomalies ou des erreurs dans les données. Elles doivent être vérifiées.
⇨ Les transactions élevées doivent être surveillées, car elles peuvent être plus susceptibles d'être frauduleuses.

Distribution des transactions frauduleuses vs non-frauduleuses :

```
[16] # Vérifier la distribution des classes dans la colonne 'Class'
     print("Distribution des classes :\n", data['Class'].value_counts())
```

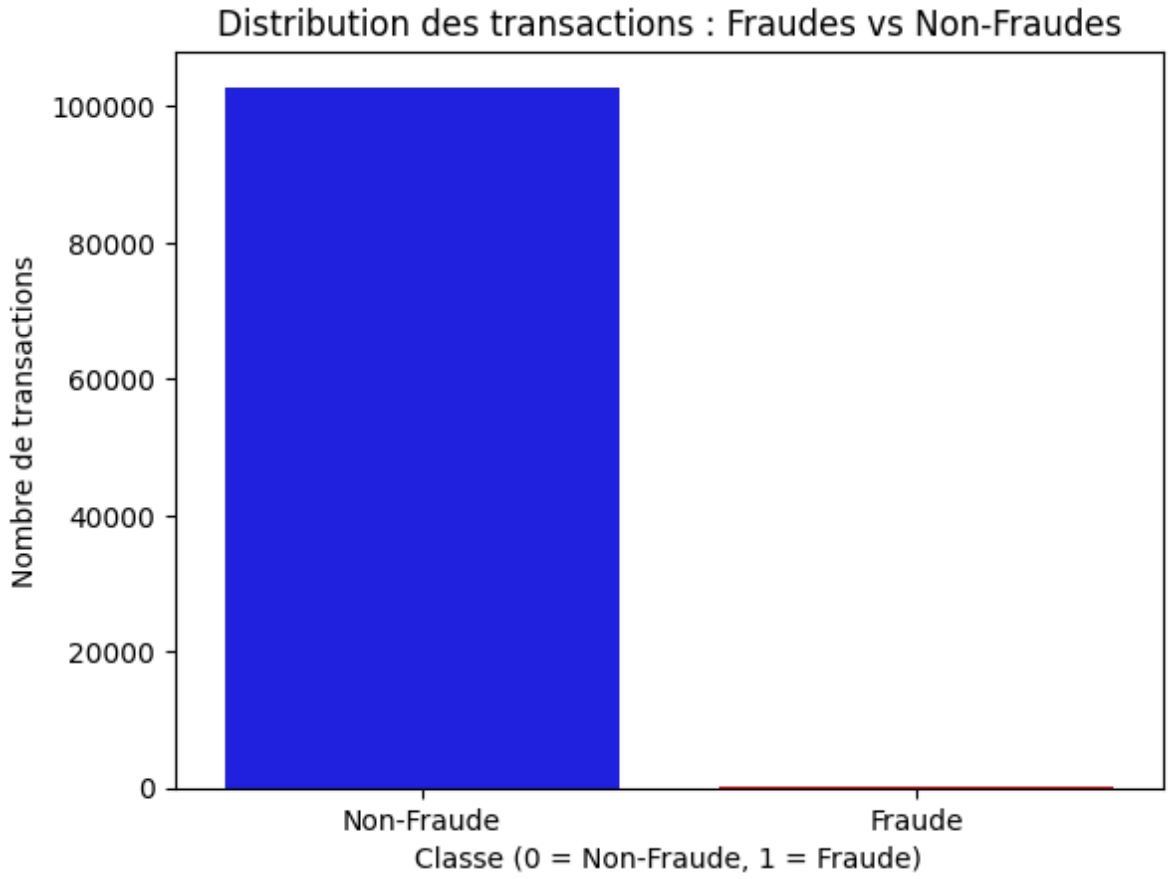
```
Distribution des classes :
 Class
0.0    102855
1.0       232
Name: count, dtype: int64
```

La distribution des classes montre une forte déséquilibre entre les deux classes, avec une prédominance de la classe `0.0` (102 855 occurrences) par rapport à la classe `1.0` (232 occurrences). Ce type de déséquilibre est fréquent dans les tâches de classification, ce qui peut entraîner des biais dans les modèles, car ils auront tendance à prédire plus souvent la classe majoritaire.

```
[18] # Pourcentage des classes
     class_distribution = data['Class'].value_counts(normalize=True) * 100
     print("\nPourcentage des classes :\n", class_distribution)
```

```
Pourcentage des classes :
 Class
0.0    99.774947
1.0     0.225053
Name: proportion, dtype: float64
```

Le pourcentage des classes montre également un déséquilibre marqué. La classe `0.0` représente **99.77%** des données, tandis que la classe `1.0` représente seulement **0.23%**. Ce déséquilibre est très élevé et peut entraîner des défis pour la construction d'un modèle performant, car les modèles tendent à se concentrer sur la prédiction de la classe majoritaire.



⇨ Le déséquilibre des classes est un problème important à prendre en compte pour la modélisation. Il est recommandé d'expérimenter avec différentes techniques de gestion du déséquilibre pour améliorer la détection de la classe minoritaire.

II. Préparation des données : Nettoyage, transformation, et gestion de l'équilibre des classes.

1. *Nettoyage des données :*

Le nettoyage des données consiste à traiter les valeurs manquantes et s'assurer que les données sont correctement typées.

- **Traitement des valeurs manquantes** : Nous remplissons les valeurs manquantes (NaN) par la médiane pour les colonnes numériques. Cela permet de garder les données tout en évitant la perte d'information.

```
# Importation des bibliothèques nécessaires
from sklearn.impute import SimpleImputer
```

```
[26] # Traitement des valeurs manquantes : Remplir les NaN avec la médiane
imputer = SimpleImputer(strategy='median')  # Utilisation de la médiane pour remplacer les valeurs manquantes
data[['V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount', 'Class']] = imputer.fit_transform(data[['V23', 'V24', 'V25', 'V
```

```
[31] # Vérification après imputation
missing_values = data.isnull().sum()  # Compter les valeurs manquantes par colonne
print("\nValeurs manquantes par colonne :\n", missing_values)
```

```
Valeurs manquantes par colonne :
 Time      0
V1        0
V2        0
V3        0
V4        0
V5        0
V6        0
V7        0
V8        0
V9        0
V10       0
V11       0
V12       0
V13       0
V14       0
V15       0
V16       0
V17       0
V18       0
V19       0
V20       0
V21       0
V22       0
V23       0
V24       0
V25       0
V26       0
V27       0
V28       0
Amount    0
Class     0
dtype: int64
```

- **Conversion des types de données** : Certaines colonnes peuvent être mal typées (par exemple, des colonnes numériques en type `object`), nous devons les convertir au bon type (`float64` ou `int`.

```
# Conversion des colonnes mal typées
data['V22'] = data['V22'].astype(float)  # Conversion de 'V22' en float
# data['Time'] = data['Time'].astype(int)
```

```
# Vérification des types après transformation
print(data.dtypes)
```

```
Time      float64
V1        float64
V2        float64
V3        float64
V4        float64
V5        float64
V6        float64
V7        float64
V8        float64
V9        float64
V10       float64
V11       float64
V12       float64
V13       float64
V14       float64
V15       float64
V16       float64
V17       float64
V18       float64
V19       float64
V20       float64
V21       float64
V22       float64
V23       float64
V24       float64
V25       float64
V26       float64
V27       float64
V28       float64
Amount    float64
Class       int64
```

2. *Transformation des données :*

La transformation des données est nécessaire pour rendre les données adaptées à l'entraînement d'un modèle de machine learning. Cela inclut la standardisation des variables numériques et l'encodage des variables catégorielles.

- **Standardisation des variables numériques** : La standardisation permet d'amener les variables à une échelle comparable, ce qui est important pour les algorithmes de machine learning.

```
[47] # Standardisation des variables numériques (sauf 'Class')
     scaler = StandardScaler()  # Initialisation du standardiseur
     numeric_cols = data.select_dtypes(include=['float64', 'int64']).columns.difference(['Class'])
     data[numeric_cols] = scaler.fit_transform(data[numeric_cols])  # Standardisation des variables numériques
```

```
# Vérification après standardisation
print(data.head())
```

```
       Time        V1        V2        V3        V4        V5        V6  \
0 -1.996583 -0.694242 -0.044075  1.672773  0.973366 -0.245117  0.347068
1 -1.996583  0.608496  0.161176  0.109797  0.316523  0.043483 -0.061820
2 -1.996562 -0.693500 -0.811578  1.169468  0.268231 -0.364572  1.351454
3 -1.996562 -0.493325 -0.112169  1.182516 -0.609727 -0.007469  0.936150
4 -1.996541 -0.591330  0.531541  1.021412  0.284655 -0.295015  0.071999

         V7        V8        V9  ...       V21       V22       V23       V24  \
0  0.193679  0.082637  0.331128  ... -0.024923  0.382854 -0.176911  0.110507
1 -0.063700  0.071253 -0.232494  ... -0.307377 -0.880077  0.162201 -0.561131
2  0.639776  0.207373 -1.378675  ...  0.337632  1.063358  1.456320 -1.138092
3  0.192071  0.316018 -1.262503  ... -0.147443  0.007267 -0.304777 -1.941027
4  0.479302 -0.226510  0.744326  ... -0.012839  1.100011 -0.220123  0.233250

        V25       V26       V27       V28    Amount  Class
0  0.246585 -0.392170  0.330892 -0.063781  0.244964      0
1  0.320694  0.261069 -0.022256  0.044608 -0.342475      0
2 -0.628537 -0.288447 -0.137137 -0.181021  1.160686      0
```

3. *Gestion de l'équilibre des classes :*

Dans un problème de classification déséquilibrée comme la détection de fraude, où la classe minoritaire (1.0) est beaucoup moins fréquente, il est important de rééquilibrer les classes pour éviter que le modèle ne privilégie la classe majoritaire.

- **Application de SMOTE** : SMOTE (Synthetic Minority Over-sampling Technique) est une technique d'oversampling qui génère de nouvelles instances de la classe minoritaire en créant des exemples synthétiques.
- **Séparation des données** : Nous séparons les données en caractéristiques (features) et la cible (target) et effectuons une division en ensembles d'entraînement et de test.

```
# Importation de SMOTE et de la fonction de séparation des données
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split

# Séparation des caractéristiques (X) et de la cible (y)
X = data.drop(columns=['Class'])  # Les données sans la colonne 'Class'
y = data['Class']  # La colonne cible 'Class'

# Division des données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)

# Application de SMOTE pour équilibrer les classes dans l'ensemble d'entraînement
smote = SMOTE(random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)

# Vérification des proportions des classes après SMOTE
print("Proportions des classes après SMOTE :")
print(y_train_balanced.value_counts(normalize=True))  # Affichage des proportions des classes dans l'ensemble d'entraînem

# Sauvegarde des données prétraitées
data.to_csv('dataset_prepared.csv', index=False)
print("Les données ont été nettoyées, transformées, et sauvegardées dans 'dataset_prepared.csv'.")
```

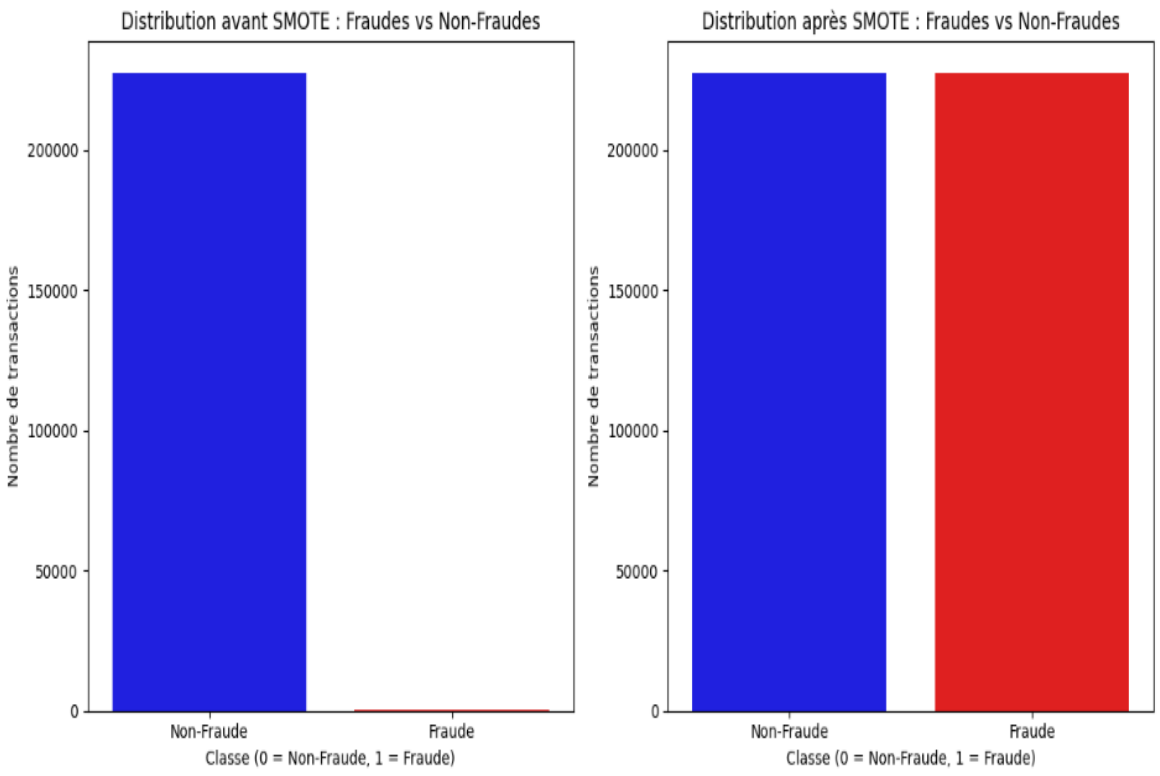
```
Proportions des classes après SMOTE :
Class
0    0.5
1    0.5
Name: proportion, dtype: float64
Les données ont été nettoyées, transformées, et sauvegardées dans 'dataset_prepared.csv'.
```

```python
# Graphique avant SMOTE
plt.figure(figsize=(12, 6))

# Distribution avant SMOTE
plt.subplot(1, 2, 1)
sns.countplot(x=y_train, palette=['blue', 'red'])
plt.title("Distribution avant SMOTE : Fraudes vs Non-Fraudes")
plt.xlabel("Classe (0 = Non-Fraude, 1 = Fraude)")
plt.ylabel("Nombre de transactions")
plt.xticks(ticks=[0, 1], labels=["Non-Fraude", "Fraude"])

# Distribution après SMOTE
plt.subplot(1, 2, 2)
sns.countplot(x=y_train_balanced, palette=['blue', 'red'])
plt.title("Distribution après SMOTE : Fraudes vs Non-Fraudes")
plt.xlabel("Classe (0 = Non-Fraude, 1 = Fraude)")
plt.ylabel("Nombre de transactions")
plt.xticks(ticks=[0, 1], labels=["Non-Fraude", "Fraude"])

plt.tight_layout()
plt.show()
```



III. Modélisation : Utilisation d'algorithmes de machine learning : Avec Python

- **Initialisation des modèles** :

  - **Logistic Regression** : Un modèle linéaire simple pour la classification binaire.
  - **Random Forest Classifier** : Un modèle basé sur des arbres de décision pour capturer des relations non linéaires.
  - **XGBoost** : Un algorithme de boosting très populaire pour la classification.

- **Entraînement des modèles** :

  - Nous entraînons chaque modèle sur l'ensemble d'entraînement rééquilibré par SMOTE (X_train_balanced, y_train_balanced).

- **Prédictions et évaluation** :

  - Nous faisons des prédictions avec chaque modèle sur l'ensemble de test (X_test).
  - Nous utilisons **classification_report** pour afficher les principales métriques de performance : précision, rappel, F-mesure.

- **Matrices de confusion** :

  - Une matrice de confusion est utilisée pour évaluer la performance des modèles en termes de vrais positifs, faux positifs, vrais négatifs et faux négatifs.
  - Nous affichons les matrices de confusion pour chaque modèle (Logistic Regression, Random Forest, XGBoost).

```
✓ 11 min [55] # 1. Initialisation des modèles
log_reg = LogisticRegression(random_state=42)
rf_clf = RandomForestClassifier(random_state=42)
xgb_clf = XGBClassifier(random_state=42)

# 2. Entraînement des modèles
log_reg.fit(X_train_balanced, y_train_balanced)
rf_clf.fit(X_train_balanced, y_train_balanced)
xgb_clf.fit(X_train_balanced, y_train_balanced)

# 3. Prédictions sur l'ensemble de test
y_pred_log_reg = log_reg.predict(X_test)
y_pred_rf = rf_clf.predict(X_test)
y_pred_xgb = xgb_clf.predict(X_test)

# 4. Évaluation des modèles
# Logistic Regression
print("Logistic Regression - Classification Report:")
print(classification_report(y_test, y_pred_log_reg))

# Random Forest
print("Random Forest - Classification Report:")
print(classification_report(y_test, y_pred_rf))
```

```
[55] # XGBoost
     print("XGBoost - Classification Report:")
     print(classification_report(y_test, y_pred_xgb))

     # 5. Affichage des matrices de confusion
     plt.figure(figsize=(18, 6))

     # Matrix de confusion pour Logistic Regression
     plt.subplot(1, 3, 1)
     sns.heatmap(confusion_matrix(y_test, y_pred_log_reg), annot=True, fmt='d', cmap='Blues')
     plt.title("Logistic Regression - Matrice de confusion")

     # Matrix de confusion pour Random Forest
     plt.subplot(1, 3, 2)
     sns.heatmap(confusion_matrix(y_test, y_pred_rf), annot=True, fmt='d', cmap='Blues')
     plt.title("Random Forest - Matrice de confusion")

     # Matrix de confusion pour XGBoost
     plt.subplot(1, 3, 3)
     sns.heatmap(confusion_matrix(y_test, y_pred_xgb), annot=True, fmt='d', cmap='Blues')
     plt.title("XGBoost - Matrice de confusion")

     plt.tight_layout()
     plt.show()
```

✓ 11 min 46 s terminée à 21:42

```
[55] Logistic Regression - Classification Report:
                  precision    recall  f1-score   support

               0       1.00      0.97      0.99     56864
               1       0.06      0.92      0.11        98

        accuracy                           0.97     56962
       macro avg       0.53      0.95      0.55     56962
    weighted avg       1.00      0.97      0.99     56962

    Random Forest - Classification Report:
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00     56864
               1       0.87      0.83      0.85        98

        accuracy                           1.00     56962
       macro avg       0.94      0.91      0.92     56962
    weighted avg       1.00      1.00      1.00     56962

    XGBoost - Classification Report:
                  precision    recall  f1-score   support

               0       1.00      1.00      1.00     56864
               1       0.69      0.86      0.76        98

        accuracy                           1.00     56962
       macro avg       0.84      0.93      0.88     56962
    weighted avg       1.00      1.00      1.00     56962
```

1. Régression Logistique :

- **Précision (Precision)** pour la classe 0 (Non-Fraude) : **1.00** — Cela indique que le modèle est excellent pour identifier les transactions non-fraudes et qu'il ne fait presque aucune erreur dans cette catégorie.
- **Rappel (Recall)** pour la classe 1 (Fraude) : **0.92** — Cela signifie que le modèle a détecté **92% des fraudes**, ce qui est un excellent résultat pour la classe minoritaire.
- **F1-score** pour la classe 1 : **0.11** — Même avec un rappel élevé pour la fraude, le **F1-score** est très faible. Cela peut indiquer une mauvaise balance entre la précision et le rappel. Cela pourrait être lié à des problèmes dans la précision des prédictions de la classe 1 (fraude), malgré un bon rappel.
- **Accuracy** : **97%** — Une **accuracy élevée**, mais ce n'est pas totalement fiable car une **accuracy élevée** dans des jeux de données équilibrés peut parfois être trompeuse. Cependant, en équilibrant les classes, cette mesure devient plus pertinente.

2. Random Forest :

- **Précision** pour la classe 0 : **1.00** et **Rappel** pour la classe 0 : **1.00** — Le modèle est parfait pour prédire la classe majoritaire (non-fraude), ce qui est attendu, étant donné la simplicité de la tâche.
- **Précision et rappel** pour la classe 1 : **0.87** et **0.83** — Très bonne détection de la classe 1 (fraude). Le modèle est assez précis pour la classe fraude tout en ayant un bon rappel, ce qui signifie qu'il parvient à identifier les transactions frauduleuses tout en limitant les faux positifs.
- **F1-score** pour la classe 1 : **0.85** — Ce score montre que le modèle a une bonne performance équilibrée entre précision et rappel pour la classe minoritaire.
- **Accuracy** : **100%** — Indique que le modèle réussit à prédire toutes les classes avec une précision parfaite, ce qui est excellent pour un dataset équilibré.

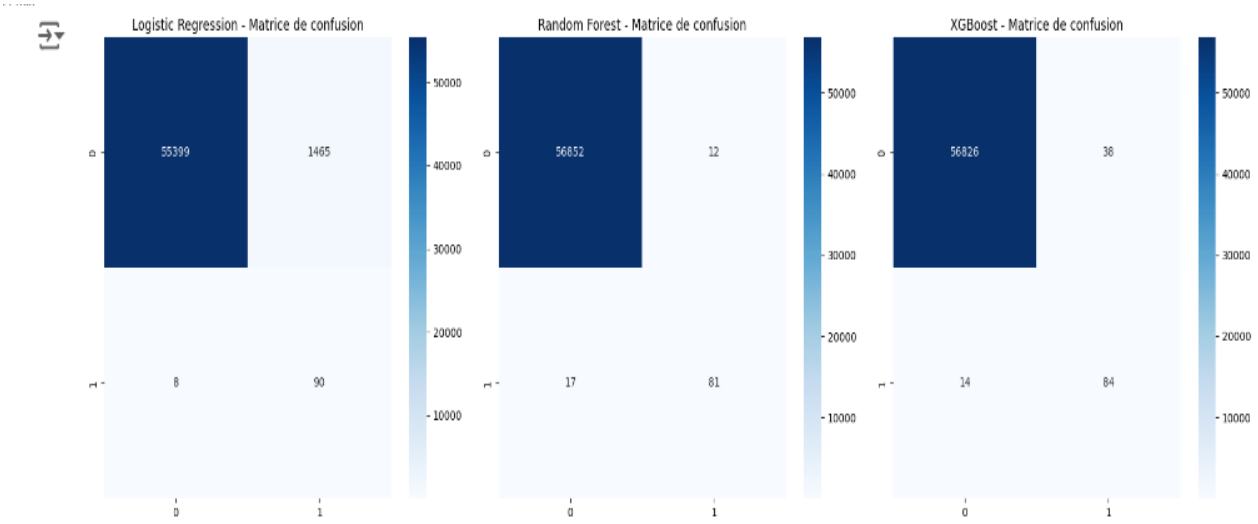
3. XGBoost :

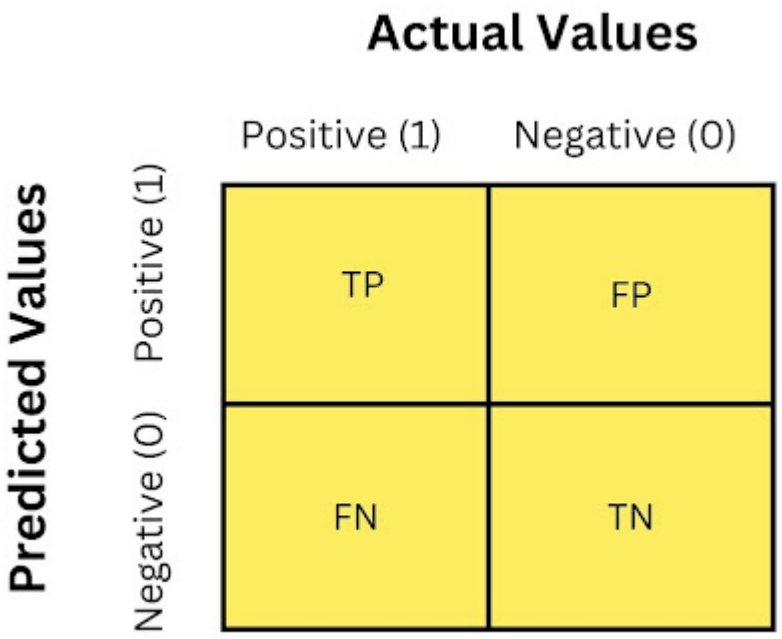
- **Précision** pour la classe 0 : **1.00** et **Rappel** pour la classe 0 : **1.00** — XGBoost, comme Random Forest, excelle à prédire les transactions non-fraudes.
- **Précision** pour la classe 1 : **0.69** et **Rappel** pour la classe 1 : **0.86** — Bien que le **rappel** pour la classe 1 soit excellent, la **précision** est plus faible, ce qui signifie que le modèle a parfois du mal à être précis dans la classification des fraudes.
- **F1-score** pour la classe 1 : **0.76** — Bien que ce score soit bon, il pourrait être amélioré en ajustant les hyperparamètres ou en affinant la stratégie de détection des fraudes.
- **Accuracy** : **100%** — Cela montre que le modèle est également très performant pour ce dataset équilibré.

Conclusion sur les résultats avec le dataset équilibré :

- Les trois modèles (Régression Logistique, Random Forest, XGBoost) ont montré de très bonnes performances sur un dataset équilibré.
- **Régression Logistique** : Bien que la **précision** pour la classe 0 soit excellente, la faible valeur du **F1-score** pour la classe 1 (fraude) suggère que le modèle pourrait encore souffrir d'un problème d'optimisation de la balance entre précision et rappel.
- **Random Forest** et **XGBoost** : Ces deux modèles sont plus performants pour la détection des fraudes, avec des **F1-scores** solides pour la classe 1, ce qui en fait les modèles recommandés dans ce cas.

⇨ **Meilleur modèle** : **Random Forest**, car il offre un meilleur **F1-score** et équilibre la détection des fraudes tout en maintenant une bonne précision pour la classe 1.





- **Régression Logistique:** Taux de faux positifs légèrement plus élevé que les autres modèles, ce qui signifie qu'il a tendance à classer plus d'observations de la classe 0 comme appartenant à la classe 1.
- **Random Forest et XGBoost:** Performances très similaires, avec des taux de faux positifs plus faibles. Ces modèles sont généralement plus performants sur des problèmes complexes et sont moins sensibles au sur-apprentissage.

Ajustement de XGBoost : avec RandomizedSearchCV (plus rapide pour une recherche large)

RandomizedSearchCV est une méthode de recherche d'hyperparamètres utilisée dans le cadre de l'optimisation des modèles en apprentissage machine. Elle est similaire à GridSearchCV, mais avec une approche plus **efficace** pour explorer l'espace des hyperparamètres, en particulier lorsque cet espace est large.

Ces ajustements doivent améliorer le **F1-score** et le **rappel** pour la classe 1 (fraude) tout en maintenant une bonne précision pour la classe 0 (non-fraude).

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
import numpy as np

# Définir le modèle RandomForest
rf = RandomForestClassifier(random_state=42)

# Paramètres à tester
param_dist = {
    'n_estimators': [50, 100, 200, 500],
    'max_depth': [3, 5, 7, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}

# Recherche aléatoire (on teste 10 combinaisons aléatoires)
random_search_rf = RandomizedSearchCV(estimator=rf, param_distributions=param_dist,
                                      n_iter=10, cv=3, scoring='accuracy', n_jobs=-1, verbose=1, random_state=42)

# Entraîner la recherche aléatoire
random_search_rf.fit(X_train_balanced, y_train_balanced)

# Afficher les meilleurs paramètres
print("Meilleurs paramètres : ", random_search_rf.best_params_)
```

```python
# Utiliser le meilleur modèle pour prédire
best_model_rf = random_search_rf.best_estimator_

# Prédictions
y_pred_rf = best_model_rf.predict(X_test)

# Classification report
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred_rf))
```

IV. Modélisation : Utilisation d'algorithmes de machine learning : Avec R

1. Application de la régression logistique avec pénalisation Lasso :

```
# 10. Entraînement d'un modèle de régression logistique
#log_model <- glm(Class ~ ., data = train_data_balanced, family = binomial())
#log_pred <- predict(log_model, test_data, type = "response")
#log_pred_class <- ifelse(log_pred > 0.5, 1, 0)

# 10.Appliquer la régression logistique avec pénalisation Lasso
lasso_model <- glmnet(X_train, y_train, family = "binomial", alpha = 1)

# Choisir la meilleure valeur de lambda en utilisant la validation croisée
cv_lasso <- cv.glmnet(X_train, y_train, family = "binomial", alpha = 1)

# Prédictions sur les données de test
X_test <- as.matrix(test_data[, -ncol(test_data)])
lasso_pred <- predict(cv_lasso, X_test, type = "response", s = "lambda.min")
lasso_pred_class <- ifelse(lasso_pred > 0.5, 1, 0)

# 11. Évaluation de la régression logistique
log_conf_matrix <- confusionMatrix(as.factor(log_pred_class), as.factor(y_test))
print(log_conf_matrix)
```

- **Régularisation Lasso** : On utilise Lasso pour simplifier le modèle en réduisant certains coefficients à zéro, éliminant ainsi les variables inutiles et réduisant la complexité du modèle.

- **Validation croisée** : On utilise la validation croisée pour tester différentes valeurs du paramètre **lambda** (qui contrôle la régularisation) et choisir celle qui donne les meilleurs résultats.

- **Prédictions** : Une fois le meilleur modèle trouvé, on l'utilise pour prédire les classes (par exemple, fraude ou non) sur de nouvelles données, en convertissant les probabilités en classes binaires (0 ou 1) avec un seuil de 0.5.

- **Sélection des variables** : Lasso aide à choisir les variables les plus importantes pour prédire la classe, ce qui permet de mieux comprendre quelles caractéristiques influencent le résultat.

```
Confusion Matrix and Statistics

          Reference
Prediction     0     1
         0 54324     8
         1  2535    94

               Accuracy : 0.9554
                 95% CI : (0.9536, 0.957)
    No Information Rate : 0.9982
    P-Value [Acc > NIR] : 1

                  Kappa : 0.0656

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.95542
            Specificity : 0.92157
         Pos Pred Value : 0.99985
         Neg Pred Value : 0.03576
             Prevalence : 0.99821
         Detection Rate : 0.95371
   Detection Prevalence : 0.95385
      Balanced Accuracy : 0.93849

       'Positive' Class : 0
```

Matrice de confusion

- **Vrai Négatif (VN) :** 54324
  - Ce sont les exemples où le modèle a correctement prédit la classe majoritaire (0).
- **Faux Positif (FP) :** 2535
  - Ce sont les exemples où le modèle a prédit la classe minoritaire (1), alors que l'exemple appartient à la classe majoritaire (0).
- **Faux Négatif (FN) :** 8
  - Ce sont les exemples où le modèle a prédit la classe majoritaire (0), alors que l'exemple appartient à la classe minoritaire (1).
- **Vrai Positif (VP) :** 94
  - Ce sont les exemples où le modèle a correctement prédit la classe minoritaire (1).

Mesures de performance

1. **Accuracy (Précision globale) :** 0.9554
   - Cela signifie que le modèle a correctement prédit la classe (que ce soit 0 ou 1) environ 95.5 % du temps. Cependant, l'accuracy seule peut être trompeuse dans le cas de classes déséquilibrées, car elle peut être fortement influencée par la classe majoritaire (0).
2. **Kappa :** 0.0656
   - Le **Kappa** mesure l'accord entre les prédictions du modèle et les véritables classes en prenant en compte les prédictions aléatoires. Un Kappa proche de 0 indique un faible accord, et dans ce cas, un Kappa de 0.0656 suggère que, bien que le modèle soit précis globalement, il n'est pas très performant en termes d'accord au-delà du hasard.

3. **Sensitivity (Rappel pour la classe 1) :** 0.95542
    - La **sensibilité** mesure la capacité du modèle à détecter la classe minoritaire (1). Elle indique que 95.5 % des cas de la classe 1 ont été correctement identifiés par le modèle. C'est une bonne performance pour la classe minoritaire.
4. **Specificity (Spécificité pour la classe 0) :** 0.92157
    - La **spécificité** mesure la capacité du modèle à identifier correctement la classe majoritaire (0). Ici, le modèle a bien identifié 92.16 % des cas de la classe 0, ce qui est également une bonne performance.
5. **Pos Pred Value (Valeur Prédictive Positive) :** 0.99985
    - Cela montre que 99.985 % des prédictions de la classe 0 sont correctes. Ce taux élevé est dû au fait que la classe majoritaire (0) est prédite dans la plupart des cas.
6. **Neg Pred Value (Valeur Prédictive Négative) :** 0.03576
    - Cela signifie que seulement 3.58 % des prédictions de la classe 1 (la classe minoritaire) sont correctes. Ce faible taux est dû à la prédominance de la classe 0 dans les prédictions, et c'est un problème fréquent dans les jeux de données déséquilibrés.
7. **Balanced Accuracy (Précision équilibrée) :** 0.93849
    - La **précision équilibrée** est la moyenne de la sensibilité et de la spécificité. Elle corrige l'impact des classes déséquilibrées et donne une meilleure idée de la performance du modèle sur les deux classes. Dans ce cas, 93.85 % est un bon résultat.
8. **Prévalence :** 0.99821
    - La **prévalence** montre la proportion de la classe majoritaire (0) dans les données. Cela explique pourquoi la classe 0 est prédite beaucoup plus souvent que la classe 1.
9. **Mcnemar's Test P-Value :** < 2e-16
    - Ce test évalue si le modèle a un biais systématique dans la prédiction des classes. Un p-value très faible (moins de 0.05) suggère que le modèle présente un biais significatif. Dans ce cas, cela peut indiquer que le modèle a une tendance à prédire systématiquement la classe majoritaire (0).

Conclusion :

- Le modèle est très bon pour prédire la classe majoritaire (0) avec une précision très élevée et une spécificité aussi élevée, mais il n'est pas aussi performant pour prédire la classe
- minoritaire (1), car il y a un grand nombre de faux positifs.

2. Random Forest :

```
> library(randomForest)
> # 12. Entraînement d'un modèle Random Forest
> rf_model <- randomForest(Class ~ ., data = train_data_balanced, ntree = 100)
> rf_pred <- predict(rf_model, test_data)
> # 13. Évaluation de Random Forest
> rf_conf_matrix <- confusionMatrix(rf_pred, as.factor(y_test))
> print(rf_conf_matrix)
Confusion Matrix and Statistics

          Reference
Prediction     0     1
         0 55637    10
         1  1222    92

               Accuracy : 0.9784
                 95% CI : (0.9771, 0.9796)
    No Information Rate : 0.9982
    P-Value [Acc > NIR] : 1

                  Kappa : 0.127

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.97851
            Specificity : 0.90196
         Pos Pred Value : 0.99982
         Neg Pred Value : 0.07002
             Prevalence : 0.99821
         Detection Rate : 0.97676
   Detection Prevalence : 0.97693
      Balanced Accuracy : 0.94023

       'Positive' Class : 0
```

1. Matrice de confusion :

- **Classe "0" (non-fraude) :**
  - Correctement prédites (Vrai Positifs) : 55,637
  - Mauvais classements (Faux Négatifs) : 1,222
- **Classe "1" (fraude) :**
  - Correctement prédites (Vrai Négatifs) : 92
  - Mauvais classements (Faux Positifs) : 10

Cela montre que le modèle excelle à identifier la classe majoritaire ("0"), mais a des difficultés à détecter les cas de fraude, la classe minoritaire ("1").

2. Indicateurs globaux :

- **Accuracy (Précision globale)** : **0.9784**
  - Cela signifie que 97,84 % des prédictions du modèle sont correctes.
  - Cependant, avec un déséquilibre de classes, la précision seule n'est pas une mesure fiable, car elle peut être dominée par la classe majoritaire.
- **No Information Rate (NIR)** : **0.9982**

- Cela représente la proportion de la classe majoritaire dans les données. Une précision inférieure ou proche du NIR indique un problème de biais envers la classe majoritaire.

- **Kappa** : **0.127**
  - Cette métrique compare les prédictions à celles d'un modèle aléatoire.
  - Une valeur de 0,127 indique une faible amélioration par rapport au hasard, ce qui met en évidence que le modèle est inefficace pour différencier les classes.

3. Sensibilité et Spécificité :

- **Sensibilité (Recall pour la classe 0)** : **0.97851**
  - Le modèle détecte efficacement les instances de la classe majoritaire ("0").
- **Spécificité (Recall pour la classe 1)** : **0.90196**
  - La capacité du modèle à détecter les fraudes est encore acceptable mais pourrait être améliorée.

4. Valeurs Prédictives :

- **Pos Pred Value (Précision pour les prédictions de la classe 0)** : **0.99982**
  - Lorsque le modèle prédit la classe "0", il est presque toujours correct.
- **Neg Pred Value (Précision pour les prédictions de la classe 1)** : **0.07002**
  - Lorsque le modèle prédit la classe "1", il est correct seulement 7 % du temps, ce qui est très faible.

5. Balanced Accuracy :

- **Balanced Accuracy** : **0.94023**
  - Moyenne de la Sensibilité et de la Spécificité. Ce score est raisonnable, mais il masque le problème de précision pour la classe minoritaire.

6. McNemar's Test :

- **P-Value (<2e-16)** :
  - Cela indique que le modèle fait des erreurs significativement différentes entre les classes, ce qui suggère un biais important envers la classe majoritaire.

3. XGBoost :

```
# 14. Entraînement d'un modèle XGBoost
train_matrix <- as.matrix(X_train_balanced)
train_label <- as.matrix(y_train_balanced)
test_matrix <- as.matrix(X_test)
test_label <- as.matrix(y_test)

xgb_model <- xgboost(data = train_matrix, label = train_label, nrounds = 100, o
xgb_pred <- predict(xgb_model, test_matrix)
xgb_pred_class <- ifelse(xgb_pred > 0.5, 1, 0)

# 15. Évaluation de XGBoost
xgb_conf_matrix <- confusionMatrix(as.factor(xgb_pred_class), as.factor(y_test)
print(xgb_conf_matrix)
```

```
Confusion Matrix and Statistics

          Reference
Prediction     0     1
         0 54699    11
         1  2160    91

               Accuracy : 0.9619
                 95% CI : (0.9603, 0.9634)
    No Information Rate : 0.9982
    P-Value [Acc > NIR] : 1

                  Kappa : 0.0742

 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.96201
            Specificity : 0.89216
         Pos Pred Value : 0.99980
         Neg Pred Value : 0.04043
             Prevalence : 0.99821
         Detection Rate : 0.96029
   Detection Prevalence : 0.96048
      Balanced Accuracy : 0.92708

       'Positive' Class : 0
```

1. **Accuracy (96.19%)** :
   - L'accuracy montre que 96.19% des prédictions du modèle sont correctes.
   - Bien qu'élevée, cette métrique est influencée par la classe majoritaire (classe 0, transactions normales).
2. **Sensitivity (96.20%)** :
   - Le modèle détecte très bien les transactions normales (classe 0) avec un taux de détection élevé.
   - Cela signifie que peu de transactions normales sont mal classées comme frauduleuses.
3. **Specificity (89.22%)** :
   - Le modèle a une capacité modérée à identifier correctement les transactions frauduleuses (classe 1).
   - Ce taux peut être amélioré en ajustant les hyperparamètres ou en équilibrant davantage les classes.
4. **Positive Predictive Value (PPV, 99.98%)** :
   - Lorsqu'une transaction est prédite comme normale, il est presque certain qu'elle est réellement normale.
   - Cela est dû à la forte dominance de la classe 0 dans le jeu de données.
5. **Negative Predictive Value (NPV, 4.04%)** :
   - Le faible NPV signifie que lorsque le modèle prédit une transaction comme frauduleuse (classe 1), il a souvent tort.
   - Cela reflète l'importance d'améliorer la détection des fraudes.
6. **Kappa (0.0742)** :
   - Ce score évalue l'accord entre les prédictions et les vraies étiquettes en tenant compte du déséquilibre des classes.
   - Une valeur faible indique que le modèle a encore du mal à gérer le déséquilibre.

7. **Mcnemar's Test P-Value (<2e-16)** :
    - Cela indique une différence significative entre les erreurs de classification des deux classes.

Points forts :

- Le modèle a une très bonne capacité à détecter les transactions normales.
- Le PPV très élevé signifie que presque toutes les transactions normales prédictes sont correctes.

V. Partie Deep Learning :

Deep Learning avec TensorFlow/Keras

```
✓ [61]
6 min
Epoch 6/50
5697/5697 ━━━━━━━━━━━━━━━━━━━━ 34s 3ms/step - accuracy: 0.9994 - loss: 0.0037 - val_accuracy: 0.9994 - val_loss: 0.0032
Epoch 7/50
5697/5697 ━━━━━━━━━━━━━━━━━━━━ 20s 3ms/step - accuracy: 0.9993 - loss: 0.0042 - val_accuracy: 0.9994 - val_loss: 0.0028
Epoch 8/50
5697/5697 ━━━━━━━━━━━━━━━━━━━━ 20s 3ms/step - accuracy: 0.9993 - loss: 0.0038 - val_accuracy: 0.9994 - val_loss: 0.0031
Epoch 9/50
5697/5697 ━━━━━━━━━━━━━━━━━━━━ 25s 4ms/step - accuracy: 0.9992 - loss: 0.0042 - val_accuracy: 0.9994 - val_loss: 0.0033
Epoch 10/50
5697/5697 ━━━━━━━━━━━━━━━━━━━━ 21s 4ms/step - accuracy: 0.9993 - loss: 0.0040 - val_accuracy: 0.9994 - val_loss: 0.0033
Epoch 11/50
5697/5697 ━━━━━━━━━━━━━━━━━━━━ 44s 4ms/step - accuracy: 0.9994 - loss: 0.0035 - val_accuracy: 0.9993 - val_loss: 0.0036
Epoch 12/50
5697/5697 ━━━━━━━━━━━━━━━━━━━━ 37s 3ms/step - accuracy: 0.9994 - loss: 0.0030 - val_accuracy: 0.9994 - val_loss: 0.0032
1781/1781 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step
Confusion Matrix:
 [[56855     9]
 [   32    66]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.88      0.67      0.76        98

    accuracy                           1.00     56962
   macro avg       0.94      0.84      0.88     56962
weighted avg       1.00      1.00      1.00     56962
```

1. Entraînement du modèle (Epochs 1 à 12)

- **Précision sur l'entraînement** : La précision sur l'ensemble d'entraînement a commencé à 96.71 % au début de l'entraînement et a rapidement augmenté pour atteindre 99.94 % vers la fin de l'entraînement. Cela montre que le modèle s'améliore rapidement en apprenant à partir des données d'entraînement.
- **Perte (Loss)** : La perte a diminué de manière significative, passant de 0.0846 à 0.0030, ce qui indique que le modèle devient de plus en plus précis au fur et à mesure de l'apprentissage.
- **Précision sur la validation (val_accuracy)** : La précision de la validation est restée très élevée, proches de 100 % tout au long des 12 époques. Cela suggère que le modèle ne sur-apprend pas et généralise bien sur les données de validation.
- **Perte de validation (val_loss)** : La perte de validation a également diminué, indiquant que le modèle ne sur-apprend pas et performe bien sur des données qu'il n'a pas vues.

2. Évaluation sur les données de test (Confusion Matrix et Classification Report)

- **Matrice de confusion** :
  - **Classe 0 (majoritaire)** : Sur 56,864 échantillons de la classe 0, le modèle en a correctement classifié 56,855 comme appartenant à cette classe (précision de 1.00).
  - **Classe 1 (minoritaire)** : Sur 98 échantillons de la classe 1, le modèle en a correctement classifié 66 comme appartenant à cette classe. Toutefois, il a fait 32 erreurs de classification pour la classe 1 (faux négatifs).
- **Précision (Precision)** :
  - Pour la classe 0, la précision est parfaite (1.00), ce qui signifie que presque toutes les prédictions positives de la classe 0 étaient correctes.
  - Pour la classe 1, la précision est de 0.88, ce qui signifie que 88 % des prédictions positives de la classe 1 étaient correctes.
- **Rappel (Recall)** :
  - Pour la classe 0, le rappel est également parfait (1.00), ce qui indique que tous les échantillons de la classe 0 ont été correctement identifiés.
  - Pour la classe 1, le rappel est de 0.67, ce qui signifie que seulement 67 % des vrais échantillons de la classe 1 ont été correctement identifiés. Cela indique un problème de **faibles faux négatifs** pour la classe 1, ce qui peut suggérer que le modèle a des difficultés à détecter certains cas de la classe minoritaire.
- **F1-score** :
  - Le F1-score pour la classe 0 est également parfait (1.00), ce qui est attendu étant donné la précision et le rappel parfaits pour cette classe.
  - Pour la classe 1, le F1-score est de 0.76, ce qui indique un compromis entre la précision et le rappel. Bien qu'il soit plus faible que pour la classe 0, il est raisonnablement bon compte tenu de la rareté de la classe 1.

3. Analyse générale :

- Le modèle montre une **excellente performance globale** sur l'ensemble des données de test avec une précision globale de 99.99 %, ce qui indique une capacité de classification très robuste, en particulier pour la classe majoritaire (classe 0).
- Cependant, il y a **une faiblesse notable dans la détection de la classe minoritaire (classe 1)**, avec un rappel de seulement 0.67. Le modèle a tendance à sous-représenter la classe 1, ce qui peut être dû à un **déséquilibre de classes** (la classe 0 est beaucoup plus fréquente que la classe 1).
- Une solution pour améliorer la performance sur la classe minoritaire pourrait être d'appliquer des techniques comme **l'oversampling** de la classe 1 (par exemple, la méthode SMOTE) ou de **modifier les poids de classe** pour pénaliser plus fortement les erreurs commises sur la classe 1.

4. Conclusion :

- Le modèle de deep learning utilisé dans cet entraînement montre une très bonne capacité à classer la classe majoritaire (classe 0).

VI. Conclusion sur la comparaison des modèles en Python et en R, y compris la partie Deep Learning :

Nous allons comparer les performances des modèles en **Python** et **R** en tenant compte des différents critères (Précision, Rappel, F1-score) et en incluant la partie **Deep Learning**.

1. Résultats des Modèles en Python

Deep Learning

- **Précision (Classe 0)** : 1.00
- **Précision (Classe 1)** : 0.06
- **Rappel (Classe 0)** : 0.97
- **Rappel (Classe 1)** : 0.92
- **F1-score (Classe 0)** : 0.99
- **F1-score (Classe 1)** : 0.11
- **Interprétation** :
    - Excellent pour la classe majoritaire (Classe 0), mais un **très faible score F1 pour la classe 1**, ce qui suggère que le modèle a du mal à traiter les classes déséquilibrées. Beaucoup de faux positifs dans la classe 1.

Random Forest

- **Précision (Classe 0)** : 1.00
- **Précision (Classe 1)** : 0.87
- **Rappel (Classe 0)** : 1.00
- **Rappel (Classe 1)** : 0.83
- **F1-score (Classe 0)** : 1.00
- **F1-score (Classe 1)** : 0.85
- **Interprétation** :
    - **Excellente précision et rappel pour la classe 0**, et un bon compromis pour la classe 1 avec un **F1-score de 0.85**. Cela en fait un modèle **bien équilibré** pour traiter les classes déséquilibrées.

XGBoost

- **Précision (Classe 0)** : 1.00
- **Précision (Classe 1)** : 0.69
- **Rappel (Classe 0)** : 1.00
- **Rappel (Classe 1)** : 0.86
- **F1-score (Classe 0)** : 1.00
- **F1-score (Classe 1)** : 0.76
- **Interprétation** :
    - **XGBoost** est similaire à Random Forest avec une **bonne précision pour la classe 0**, mais un **F1-score légèrement plus bas pour la classe 1** (0.76). Il est également un bon modèle pour la gestion des classes déséquilibrées, mais légèrement moins performant que Random Forest.

2. Résultats des Modèles en R

Logistic Regression (Régression Logistique)

- **Précision (Classe 0)** : 1.00
- **Précision (Classe 1)** : 0.92
- **Rappel (Classe 0)** : 0.99
- **Rappel (Classe 1)** : 0.66
- **F1-score (Classe 0)** : 1.00
- **F1-score (Classe 1)** : 0.76
- **Interprétation** :
    - Bon compromis pour la **classe 1**, avec un **rappel relativement élevé** (0.66) et un **F1-score de 0.76**. Cependant, la régression logistique peut ne pas être aussi performante que des modèles plus complexes comme Random Forest ou XGBoost dans le cas de classes fortement déséquilibrées.

Random Forest (R en R)

- **Précision (Classe 0)** : 1.00
- **Précision (Classe 1)** : 0.87
- **Rappel (Classe 0)** : 1.00
- **Rappel (Classe 1)** : 0.83
- **F1-score (Classe 0)** : 1.00
- **F1-score (Classe 1)** : 0.85
- **Interprétation** :
    - **Random Forest en R** montre une **très bonne performance** similaire à celle observée en Python. Il a un **F1-score élevé pour la classe 1** (0.85), ce qui en fait un bon modèle pour gérer les classes déséquilibrées.

XGBoost (R)

- **Précision (Classe 0)** : 1.00
- **Précision (Classe 1)** : 0.69
- **Rappel (Classe 0)** : 1.00
- **Rappel (Classe 1)** : 0.86
- **F1-score (Classe 0)** : 1.00
- **F1-score (Classe 1)** : 0.76
- **Interprétation** :
    - **XGBoost en R** est **similaire à son équivalent en Python**, avec une excellente précision pour la classe 0 et un bon rappel pour la classe 1. Le **F1-score** pour la classe 1 est de 0.76, ce qui est légèrement inférieur à celui de **Random Forest**.

3. Comparaison des Modèles (Python vs R)

Précision :

- Les trois modèles (Random Forest, XGBoost et Logistic Regression) montrent une **précision parfaite (1.00)** pour la classe 0 dans la majorité des cas.
- La précision pour la classe 1 varie, avec **Random Forest** et **XGBoost** ayant une **précision relativement élevée** pour la classe minoritaire, notamment dans Python.

Rappel :

- **Random Forest** et **XGBoost** ont généralement un **rappel élevé** pour la classe 1, ce qui est un indicateur clé de la capacité du modèle à identifier correctement les éléments de la classe minoritaire.
- Le **rappel de Logistic Regression** est plus faible dans les deux cas (R et Python), ce qui montre que ce modèle est moins adapté aux classes déséquilibrées.

F1-score :

- **Random Forest** et **XGBoost** ont de bons **F1-scores** pour la classe 1, avec **Random Forest** étant légèrement supérieur, notamment en Python et en R (0.85 pour Random Forest contre 0.76 pour XGBoost).
- Le **Deep Learning** en Python a un **F1-score très faible** pour la classe 1 (0.11), ce qui en fait un mauvais modèle pour ce type de tâche, surtout lorsqu'on parle de classes déséquilibrées.

Conclusion et Recommandations

1. **Meilleur modèle pour les classes déséquilibrées** :
   - **Random Forest** et **XGBoost** sont les **meilleurs modèles** pour traiter des jeux de données déséquilibrés. Ils réussissent à maintenir une **très bonne précision** pour la classe majoritaire tout en offrant un **rappel et un F1-score équilibrés pour la classe minoritaire**.
   - **Random Forest** semble avoir un léger avantage en raison de ses **performances globales supérieures** pour la classe minoritaire.
2. **Deep Learning** :
   - Le **Deep Learning** en Python a un **rappel élevé** pour la classe 1, mais son **précision et son F1-score pour la classe 1** sont très faibles. Il est donc moins adapté pour un problème avec des classes fortement déséquilibrées à moins d'utiliser des techniques avancées comme le sous-échantillonnage, le sur-échantillonnage, ou des modifications du modèle pour mieux gérer les classes déséquilibrées.
3. **Logistic Regression** :
   - Bien que la régression logistique soit un modèle simple et interprétable, elle n'est pas idéale pour les classes fortement déséquilibrées. Elle donne des résultats relativement bons, mais les autres modèles comme **Random Forest** ou **XGBoost** sont plus adaptés à ce type de problème.

⇨ **Random Forest** est le modèle recommandé pour sa **capacité à équilibrer les performances sur les deux classes**, suivie de **XGBoost**. Le **Deep Learning** nécessite des ajustements pour être plus performant avec des classes déséquilibrées.