Projet de fin de module BIG DATA

DÉTECTION DE FRAUDE BANCAIRE EN TEMPS RÉEL ET DÉPLOIEMENT D'UNE INTERFACE INTELLIGENTE

Année universitaire : 2024-2025

Présenté par :
- KONATE BACHIROU
- SYLLA KARAMO
- ABRINI MOHAMED

Encadré par :
Prof IMAD SASSI

Date : May 26, 2025

Contents

0.1 INTRODUCTION

La fraude bancaire représente aujourd'hui un enjeu majeur pour les institutions financières, tant sur le plan économique que sur celui de la confiance des utilisateurs. Face à l'augmentation constante du volume des transactions numériques et à la sophistication croissante des attaques, les méthodes traditionnelles de détection se révèlent souvent insuffisantes. Dans ce contexte, l'intelligence artificielle et plus particulièrement le Machine Learning offrent des solutions innovantes et performantes pour anticiper et détecter ces fraudes. Le présent projet s'inscrit dans cette dynamique. Il vise à concevoir, entraîner et déployer un système capable de prédire la probabilité qu'une transaction bancaire soit frauduleuse, à partir de deux ensembles de données réelles et déséquilibrées. Pour cela, cinq modèles de machine learning aux architectures variées sont testés et comparés. À l'issue de cette comparaison, le modèle le plus performant sera intégré dans une interface interactive et dynamique, permettant à un utilisateur d'interroger le système par texte. Il permettra également de visualiser en temps réel les nouvelles transactions Grâce à l'outil Kafka. Enfin, le modèle sera déployé via une API sécurisée sur AWS, avec un système de monitoring afin d'assurer son bon fonctionnement en production. Ce projet allie ainsi rigueur algorithmique, accessibilité utilisateur et robustesse en environnement réel.

0.2 REVUE DE LITTÉRATURE

0.2.1 La fraude bancaire : définitions et enjeux

La fraude bancaire désigne toute activité illégale visant à obtenir un gain financier à travers des moyens trompeurs dans le système bancaire. Elle prend plusieurs formes : vols d'identité, paiements non autorisés, usurpation de carte, etc. Les enjeux sont considérables, notamment en termes de pertes économiques et d'image pour les institutions financières.

0.2.2 Les approches classiques de détection

Les méthodes traditionnelles reposent sur des règles métier prédéfinies (seuils de montant, localisation inhabituelle, fréquence anormale). Si elles sont simples et rapides, elles manquent de flexibilité, ne détectent pas les fraudes nouvelles ou subtiles, et génèrent souvent un taux élevé de faux positifs.

0.2.3 L'apport du Machine Learning

Le Machine Learning permet d'automatiser la détection en apprenant à partir de données historiques. Il détecte des schémas complexes que les règles fixes ne peuvent pas capter. Des algorithmes comme Random Forest, XGBoost ou LightGBM offrent de très bonnes performances sur des données déséquilibrées, en adaptant leurs prédictions au contexte de chaque transaction.

0.3 Phase d'analyse et de selection du modèle idéal

0.4 PRÉSENTATION DU DATASET

0.4.1 Origine des données

Ce projet repose sur deux jeux de données publics disponibles sur la plateforme Kaggle.

- **Dataset 1** : Credit Card Fraud Detection: Il contient les transactions réalisées par des titulaires de cartes en Europe en septembre 2013. Les données sont anonymisées via une transformation PCA. Lien: https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud
- **Dataset 2** : Fraud Detection Dataset : Plus récent et non anonymisé, ce dataset présente des attributs concrets tels que le type de transaction, l'heure, et le montant. Il permet de croiser des variables explicites avec des données sensibles. C'est Un jeu de données synthétique simulant des transactions par carte sur deux ans (2019-2020). Lien: https://www.kaggle.com/datasets/kartik2112/fraud-detection

0.4.2 Structure et description des variables

Dataset 1

- 284 807 transactions
- 30 variables :
- V1 à V28 : variables anonymisées issues de la PCA
- Time : seconde écoulée depuis la première transaction
- Amount : montant de la transaction
- Class : étiquette (0 = normal, 1 = fraude)

Dataset 2

- Environ 636 262 transactions
- Variables notables :
- type : type de transaction (TRANSFER, PAYMENT, etc.)
- amount : montant transféré
- oldbalanceOrg / newbalanceOrig : soldes avant/après pour l'émetteur
- isFraud : étiquette binaire indiquant la fraude

0.4.3 Analyse exploratoire

[4]:
```
# Importation des bibliothèques
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

[5]:
```
# Chargement du premier dataset
data1 = pd.read_csv("creditcard.csv")

# Importation du 2e dataset
data2_train = pd.read_csv("fraudTrain.csv")
data2_test = pd.read_csv('fraudTest.csv')
```

Affichage les proportions des non fraudes et des fraudes pour chaque dataset

[7]:
```
# dataset1
print("proportion de fraude pour le dataset1:")
print(data1['Class'].value_counts(normalize = True))
```

```
proportion de fraude pour le dataset1:
Class
0    0.998273
1    0.001727
Name: proportion, dtype: float64
```

[8]:
```
# dataset2
print("proportion de fraude pour le dataset2:")
print(data2_train['is_fraud'].value_counts(normalize = True))
```

```
proportion de fraude pour le dataset2:
is_fraud
0    0.994211
1    0.005789
Name: proportion, dtype: float64
```

[9]:
```
data2_train.columns
```

[9]:
```
Index(['Unnamed: 0', 'trans_date_trans_time', 'cc_num', 'merchant', 'category',
       'amt', 'first', 'last', 'gender', 'street', 'city', 'state', 'zip',
       'lat', 'long', 'city_pop', 'job', 'dob', 'trans_num', 'unix_time',
       'merch_lat', 'merch_long', 'is_fraud'],
      dtype='object')
```

.

Remarque: Il y'a un fort déséquilibre entre les classes dans les deux datasets. - 0.17% de fraude dans le datset 1 - 0.5% de fraude dans le dataset2

Corrélations : Dans le Dataset 1, peu d'information est directement exploitable sans dé-anonymisation. En revanche, le Dataset 2 permet d'identifier des patterns : les fraudes sont souvent liées aux transactions de type TRANSFER ou CASH_OUT.

Outliers et valeurs manquantes : Aucun des datasets ne présente de valeurs manquantes. Cependant, des outliers dans les montants nécessitent une normalisation.

[16]:
```
#Dataset1
data1.isnull().sum()
```

[16]:
```
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
```

[18]:
```
#dataset2
data2_train.isnull().sum()
```

[18]:
```
Unnamed: 0               0
trans_date_trans_time    0
cc_num                   0
merchant                 0
category                 0
amt                      0
first                    0
last                     0
gender                   0
street                   0
city                     0
state                    0
zip                      0
lat                      0
long                     0
city_pop                 0
job                      0
dob                      0
trans_num                0
unix_time                0
merch_lat                0
merch_long               0
```

```
is_fraud                 0
dtype: int64
```

0.4.4 Problème du déséquilibre des classes

La détection de fraude est **un problème d'apprentissage supervisé** avec classes déséquilibrées, où la classe frauduleuse est largement sous-représentée. Cela biaise les modèles en faveur de la classe majoritaire. Des techniques comme le sous-échantillonnage, le sur-échantillonnage (SMOTE), ou encore le choix de métriques adaptées (F1-score, AUC-ROC, Recall) sont alors essentielles pour une évaluation réaliste des performances.

0.5 MÉTHODOLOGIE

0.5.1 Prétraitement des données

Avant d'entraîner les modèles de Machine Learning, un prétraitement minutieux est nécessaire pour garantir la qualité des données et optimiser les performances des modèles. Ce prétraitement inclut les étapes suivantes :

Nettoyage des données : Les deux datasets ne présentent pas de valeurs manquantes, mais il est essentiel de vérifier la présence d'anomalies ou de valeurs extrêmes dans les montants de transactions.

Normalisation: Les montants des transactions peuvent être normalisés à l'aide de StandardScaler pour assurer que toutes les variables aient des échelles comparables, un aspect crucial pour les algorithmes comme la régression logistique mais cela n'est pas nécessaire pour tous les algorithme.

Encodage des variables catégorielles : Dans le Dataset 2, les variables catégorielles comme type seront encodées Les variables du dataset1 sont déjà encodées.

DATASET2 : Avant de passer à l'encodage, nous allons apporter des modifications aux variables pour ne garder que les plus pertinentes pour l'entraînement de nos modèles. Celle-ci est facultative, car les modèles peuvent identifier les variables pertinentes eux-mêmes, mais cela permet d'augmenter la performance et la rapidité du modèle. (Pas nécessaire pour le dataset1 car anonymisé).

[22]:
```
# Supprimer les colonnes non pertinentes
cols_to_drop = ['Unnamed: 0', 'cc_num', 'first', 'last', 'street','city','zip',␣
 ↪'trans_num', 'unix_time']
data2_train = data2_train.drop(columns=cols_to_drop, errors ='ignore')

# Transformer la colonne trans_date_time en des variables simples faciles à␣
 ↪exploiter

## Déjà s'assurer que c'est bien un objet datetime
data2_train['trans_date_trans_time'] = pd.
 ↪to_datetime(data2_train['trans_date_trans_time'])

## Extraire l'heure
data2_train['hour'] = data2_train['trans_date_trans_time'].dt.hour
```

```
## Extraire le jour de la semaine
data2_train['day_of_week'] = data2_train['trans_date_trans_time'].dt.dayofweek ␣
↪# lundi=0, mardi=1, ..., dimanche=6

## Extraire le jour du mois
data2_train['day'] = data2_train['trans_date_trans_time'].dt.day # le 01, le 02..
↪. le 31

## (Optionnel) Extraire le mois aussi
data2_train['month'] = data2_train['trans_date_trans_time'].dt.month

## Suppression de la colonne de départ
data2_train = data2_train.drop('trans_date_trans_time', axis = 1)

# Transformation de la date de naissance en âge
from datetime import datetime

## 1. S'assurer que dob est bien en format datetime
data2_train['dob'] = pd.to_datetime(data2_train['dob'])

## 2. Calculer l'âge
data2_train['age'] = (datetime.now() - data2_train['dob']).dt.days // 365

## 3. Suppression de la colonne dob
data2_train = data2_train.drop(columns=['dob'])
```

Nous remprenons le même travail pour la data de test (data2_test)

[24]:
```
# Supprimer les colonnes non pertinentes
cols_to_drop = ['Unnamed: 0', 'cc_num', 'first', 'last', 'street','city','zip',␣
↪'trans_num', 'unix_time']
data2_test = data2_test.drop(columns=cols_to_drop, errors ='ignore')

# Transformer la colonne trans_date_time en des variables simples faciles à␣
↪exploiter

## Déjà s'assurer que c'est bien un objet datetime
data2_test['trans_date_trans_time'] = pd.
↪to_datetime(data2_test['trans_date_trans_time'])

## Extraire l'heure
data2_test['hour'] = data2_test['trans_date_trans_time'].dt.hour

## Extraire le jour de la semaine
data2_test['day_of_week'] = data2_test['trans_date_trans_time'].dt.dayofweek  #␣
↪lundi=0, mardi=1, ..., dimanche=6
```

```
## Extraire le jour du mois
data2_test['day'] = data2_test['trans_date_trans_time'].dt.day # le 01, le 02...␣
↪le 31

## (Optionnel) Extraire le mois aussi
data2_test['month'] = data2_test['trans_date_trans_time'].dt.month

## Suppression de la colonne de départ
data2_test = data2_test.drop('trans_date_trans_time', axis = 1)

# Transformation de la date de naissance en âge
from datetime import datetime

## 1. S'assurer que dob est bien en format datetime
data2_test['dob'] = pd.to_datetime(data2_test['dob'])

## 2. Calculer l'âge
data2_test['age'] = (datetime.now() - data2_test['dob']).dt.days // 365

## 3. Suppression de la colonne dob
data2_test = data2_test.drop(columns=['dob'])
```

Encodage des valeurs de type string

[28]:
```
# Affichons tous d'abord le type des variables du dataset2
print(data2_train.dtypes)
print(data2_test.dtypes)
```

```
merchant        object
category        object
amt            float64
gender          object
state           object
lat            float64
long           float64
city_pop         int64
job             object
merch_lat      float64
merch_long     float64
is_fraud         int64
hour             int32
day_of_week      int32
day              int32
month            int32
age              int64
dtype: object
merchant        object
category        object
```

```
amt            float64
gender          object
state           object
lat            float64
long           float64
city_pop         int64
job             object
merch_lat      float64
merch_long     float64
is_fraud         int64
hour             int32
day_of_week      int32
day              int32
month            int32
age              int64
dtype: object
```

[30]:
```
# Transformation
from sklearn.preprocessing import LabelEncoder

# Pour toutes les colonnes de type "object"
for col in data2_train.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    data2_train[col] = le.fit_transform(data2_train[col])
    data2_test[col] = le.fit_transform(data2_test[col])  # Important pour␣
 ↪data2_test aussi
```

[31]:
```
# On affiche une nouvelles fois le types des variables
print(data2_train.dtypes)
print(data2_test.dtypes)
```

```
merchant         int32
category         int32
amt            float64
gender           int32
state            int32
lat            float64
long           float64
city_pop         int64
job              int32
merch_lat      float64
merch_long     float64
is_fraud         int64
hour             int32
day_of_week      int32
day              int32
month            int32
age              int64
```

```
dtype: object
merchant         int32
category         int32
amt            float64
gender           int32
state            int32
lat            float64
long           float64
city_pop         int64
job              int32
merch_lat      float64
merch_long     float64
is_fraud         int64
hour             int32
day_of_week      int32
day              int32
month            int32
age              int64
dtype: object
```

remarque: Toutes les variables de type sring ont été encodé

Division des dataset en des feautures (X) et en variable target (y)

[36]:
```
# DATASET1
## X1 reçoit tous les variables sauf la variable de prediction
X1 = data1.drop('Class', axis = 1)
## Y réçoit la variable de prédiction
y1 = data1['Class']
```

[38]:
```
#DATASET2
X2_train = data2_train.drop('is_fraud', axis = 1)
X2_test = data2_test.drop('is_fraud', axis = 1)
y2_train = data2_train['is_fraud']
y2_test = data2_test['is_fraud']
```

Séparation des données : Les jeux de données sont divisés en un ensemble d'entraînement (80 %) et un ensemble de test (20 %). Une validation croisée (k-fold) peut être utilisée pour assurer une meilleure estimation des performances. **Nb**: Ce travail sera effectué uniquement pour le dataset1 car le dataset2 est dejà divisé en "data2_train" et "data2_test"

[41]:
```
from sklearn.model_selection import train_test_split

# 80% entraînement 20% test
X1_train, X1_test, y1_train, y1_test = train_test_split( X1, y1 ,test_size = 0.
 ↪2, random_state = 42, stratify = y1)
```

0.5.2 Méthodes de rééquilibrage

Étant donné le fort déséquilibre des classes, plusieurs techniques de rééquilibrage peuvent être utilisées pour rendre le modèle plus sensible à la détection des fraudes.

Sur-échantillonnage (SMOTE) : Cette méthode génère de nouvelles instances de la classe minoritaire pour équilibrer le ratio entre les classes. (C'est elle que nous utiliserons au besoins)

Sous-échantillonnage : Il s'agit de réduire le nombre d'instances de la classe majoritaire pour rendre les classes plus équilibrées.

Poids des classes : Certains modèles, comme Random Forest et XGBoost, permettent de spécifier des poids de classe, donnant ainsi plus d'importance à la classe minoritaire.

0.5.3 Sélection des modèles

0.5.3.1 Régression logistique

Justification du choix : La régression logistique est un modèle de base en classification binaire. Elle permet de comprendre les relations entre les variables d'entrée et la probabilité d'appartenir à une classe donnée. Elle sert de référence de comparaison face à des modèles plus complexes.

Architecture du modèle :

Type : Modèle linéaire

Fonction d'activation : Sigmoïde

Sortie : Probabilité d'appartenance à la classe 1 (fraude)

Seuil de décision : 0.5 (ajustable selon les besoins en rappel ou précision)

Poids de classe : balanced pour compenser le déséquilibre

Optimisation : liblinear ou saga selon la taille des données

0.5.3.2 XGBoost

Justification du choix : XGBoost est un algorithme de boosting très performant, particulièrement adapté aux données déséquilibrées. Il intègre des options de pondération de classe, une régularisation fine, et gère bien les grandes quantités de données.

Architecture du modèle :

- Type : Ensemble d'arbres de décision avec boosting (Gradient Boosting)
- Nombre d'arbres (n_estimators) : Paramètre optimisé par validation croisée
- Profondeur maximale (max_depth) : Contrôle la complexité de chaque arbre
- Learning rate : Gère la vitesse d'apprentissage
- Scale_pos_weight : Utilisé pour équilibrer les classes
- Critère de perte : Log loss (binaire)

0.5.3.3 Random Forest

Justification du choix : Random Forest est un modèle d'ensemble robuste basé sur des arbres de décision. Il est capable de gérer les non-linéarités et les interactions entre variables, tout en limitant le surapprentissage grâce à l'agrégation des prédictions.

Architecture du modèle :

- Type : Ensemble d'arbres de décision formés en parallèle
- Nombre d'arbres (n_estimators) : Généralement entre 100 et 500
- max_depth : Profondeur maximale des arbres (ou None pour profondeur libre)
- bootstrap : Oui (par défaut), chaque arbre s'entraîne sur un sous-échantillon
- class_weight : balanced pour compenser le déséquilibre
- Critère de division : Gini ou entropie

0.5.3.4 LightGBM

Justification du choix : LightGBM est un algorithme de boosting optimisé pour la vitesse et la consommation mémoire. Il fonctionne très bien sur de gros volumes de données et offre une précision proche, voire supérieure à XGBoost, surtout sur des datasets structurés.

Architecture du modèle :

- Type : Boosting basé sur des feuilles (leaf-wise), différent du level-wise de XGBoost
- n_estimators : Nombre total de boosts
- max_depth / num_leaves : Paramètres clés pour contrôler la complexité
- learning_rate : Taux d'apprentissage
- is_unbalance / scale_pos_weight : Pour gérer les classes déséquilibrées
- Importance des variables : Accessible pour interprétation du modèle

0.5.3.5 Réseau de neurones (MLP – Multi-Layer Perceptron)

Pourquoi ce choix ? Le Multi-Layer Perceptron (MLP) est un modèle de deep learning particulièrement adapté à la classification binaire lorsqu'on dispose de données tabulaires comme dans notre cas. Contrairement aux modèles d'ensemble comme XGBoost ou Random Forest, qui reposent sur des arbres de décision, le MLP peut capturer des relations complexes non linéaires entre les variables. Son intégration permet également de varier les architectures, pour un choix final plus juste.

Ce modèle est aussi un bon représentant des approches à base de réseaux de neurones, ce qui enrichit la comparaison des performances en incluant une méthode fondamentalement différente.

Architecture du modèle Type de modèle : Réseau de neurones dense (MLP)

- Entrée : Le nombre de neurones en entrée correspond au nombre de features du dataset.
- Couches cachées :
- 1ère couche cachée : 64 neurones – Activation ReLU
- 2e couche cachée : 128 neurones – Activation ReLU
- Couche de sortie : 1 neurone – Activation sigmoid (car classification binaire)
- Fonction de perte : binary_crossentropy
- Optimiseur : Adam
- Époques : 10 (ajustable selon les performances)
- Batch size : 32

0.5.4 Critère d'évaluation

Étant donné le déséquilibre des classes, les métriques classiques comme la précision ne sont pas suffisantes. Ainsi, nous utiliserons les suivantes :

- AUC-ROC (Area Under Curve - Receiver Operating Characteristic) : Indicateur de la capacité du modèle à distinguer les classes.
- Précision, Rappel et F1-Score : Ces métriques permettent d'évaluer la performance du modèle en tenant compte à la fois des faux positifs et des faux négatifs.
- Matrice de confusion : Permet de visualiser le nombre de faux positifs et de vrais négatifs, afin d'avoir un aperçu détaillé de la performance du modèle.

0.6 ENTRÊNEMENT, ÉVALUATION ET COMPARAISON DES MODÈLES.

0.6.1 DATASET 1: Credit Card Fraud Detection

0.6.1.1 Création et Entraînement des modèles

Modèle de regression logistique: Ce modèle ne peut pas converger car il y'a des colonnes qui ont des valeurs très grandes par rapport aux autres colonnes - Solution: Il faut donc faire une normalisation standard avec le modèle (avec StandardScaler) - Objectif: Remettre toutes les colonnes sur la même échelle pour faciliter l'apprentissage du modèle. Formule: pour chaque valeur x d'une colonne: x_scaler = (x - moyenne)/ écart-type

[54]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X1_train_scaled = scaler.fit_transform(X1_train)
X1_test_scaled = scaler.transform(X1_test)
```

[60]:
```
from sklearn.linear_model import LogisticRegression  # importation of model

# Create a model
log_model1 = LogisticRegression(max_iter = 1000)
# entraînement
log_model1.fit(X1_train_scaled, y1_train)
# test de prediction
y1_pred_log = log_model1.predict(X1_test_scaled)
```

Modèle XGBost

[67]:
```
from xgboost import XGBClassifier

# Créer le modèle
xgb1 = XGBClassifier(eval_metric='logloss', random_state=42)

# Entraîner
xgb1.fit(X1_train, y1_train)

# Prédiction
y1_pred_xgb1 = xgb1.predict(X1_test)
```

Modèle Random foreste

[76]:
```
from sklearn.ensemble import RandomForestClassifier

# create the model
clf1 =␣
 ↪RandomForestClassifier(n_estimators=14,max_depth=11,bootstrap=False,random_state=101)

#Train the model
clf1.fit(X1_train, y1_train)

#prediction test
y1_pred_rf = clf1.predict(X1_test)
```

Modèle MLP

[80]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Création du modèle
model = Sequential()
model.add(Dense(64, input_dim=X1_train.shape[1], activation='relu'))  # Couche␣
 ↪d'entrée + 1ère couche cachée
model.add(Dense(128, activation='relu'))  # 2e couche cachée
model.add(Dense(1, activation='sigmoid'))  # Couche de sortie (classification␣
 ↪binaire)

# Compilation
model.compile(loss='binary_crossentropy', optimizer=Adam(), metrics=['accuracy'])

# Entraînement
history = model.fit(X1_train, y1_train, epochs=10, batch_size=32,␣
 ↪validation_split=0.2, verbose=1)
```

```
D:\Users\HP\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/10
5697/5697  22s 2ms/step -
accuracy: 0.9957 - loss: 18.4372 - val_accuracy: 0.9978 - val_loss: 0.2496
Epoch 2/10
5697/5697  12s 2ms/step -
accuracy: 0.9955 - loss: 5.8415 - val_accuracy: 0.9983 - val_loss: 0.0809
Epoch 3/10
5697/5697  13s 2ms/step -
```

```
accuracy: 0.9970 - loss: 0.8299 - val_accuracy: 0.9982 - val_loss: 0.0144
Epoch 4/10
5697/5697  13s 2ms/step -
accuracy: 0.9984 - loss: 0.0153 - val_accuracy: 0.9982 - val_loss: 0.0133
Epoch 5/10
5697/5697  16s 3ms/step -
accuracy: 0.9983 - loss: 0.0306 - val_accuracy: 0.9982 - val_loss: 0.0134
Epoch 6/10
5697/5697  13s 2ms/step -
accuracy: 0.9978 - loss: 0.0273 - val_accuracy: 0.9982 - val_loss: 0.0133
Epoch 7/10
5697/5697  13s 2ms/step -
accuracy: 0.9983 - loss: 0.0194 - val_accuracy: 0.9982 - val_loss: 0.0135
Epoch 8/10
5697/5697  37s 6ms/step -
accuracy: 0.9983 - loss: 0.0189 - val_accuracy: 0.9982 - val_loss: 0.0137
Epoch 9/10
5697/5697  17s 2ms/step -
accuracy: 0.9980 - loss: 0.0334 - val_accuracy: 0.9982 - val_loss: 0.2175
Epoch 10/10
5697/5697  12s 2ms/step -
accuracy: 0.9981 - loss: 0.0186 - val_accuracy: 0.9982 - val_loss: 0.0134
```

[154]:
```python
# Prédictions
y1_pred_proba = model.predict(X1_test)
y1_pred_mlp = (y1_pred_proba > 0.5).astype(int)
```

```
1781/1781  2s 942us/step
```

LightGBM

[186]:
```python
from lightgbm import LGBMClassifier
scale_pos_weight1 = np.sum(y1_train == 0) / np.sum(y1_train == 1)

lgbm_clf1 = LGBMClassifier(
    n_estimators=300,          # nombre total d'arbres (boosting rounds)
    learning_rate=0.05,        # shrinkage step (plus petit = + d'arbres)
    max_depth=-1,              # -1 = profondeur illimitée, on contrôle plutôt␣
 ↪avec num_leaves
    num_leaves=31,             # nbre max de feuilles par arbre (31 par défaut)
    subsample=0.8,             # échantillonnage lignes (aide à généraliser)
    colsample_bytree=0.8,      # échantillonnage colonnes
    scale_pos_weight1 = scale_pos_weight1,  # équilibre le poids de la classe 1
    objective="binary",
    random_state=101,
    n_jobs=-1,  # parallélisation CPU
    verbosity=0  # Masquer les messages pendant l'entraînement
```

```
)

#Entraînement du modèle
lgbm_clf1.fit(X1_train, y1_train)

#test prediction
y1_pred_lgbm = lgbm_clf1.predict(X1_test)
```

```
[LightGBM] [Warning] Unknown parameter: scale_pos_weight1
[LightGBM] [Warning] Unknown parameter: scale_pos_weight1
[LightGBM] [Warning] Unknown parameter: scale_pos_weight1
```

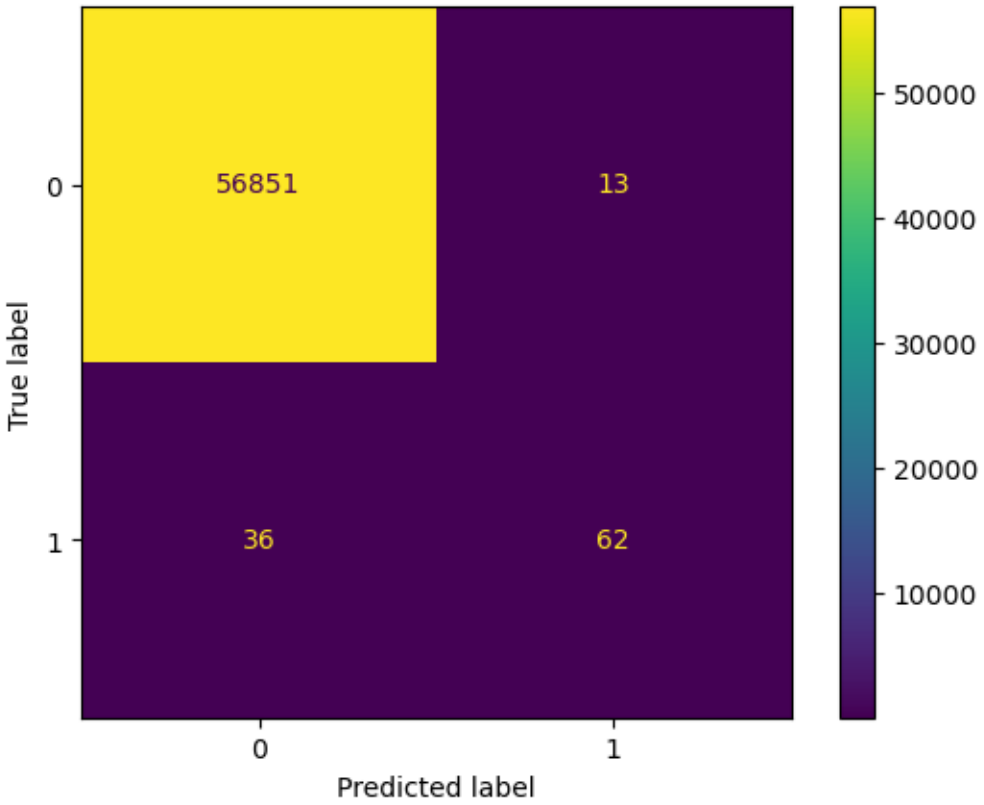
0.6.1.2 Évaluation des performances et choix du meilleur modèle

[118]:
```
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
```

Regression logistique: performance

[121]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y1_test, y1_pred_log))

#Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y1_test,y1_pred_log)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.83      0.63      0.72        98

    accuracy                           1.00     56962
   macro avg       0.91      0.82      0.86     56962
weighted avg       1.00      1.00      1.00     56962
```



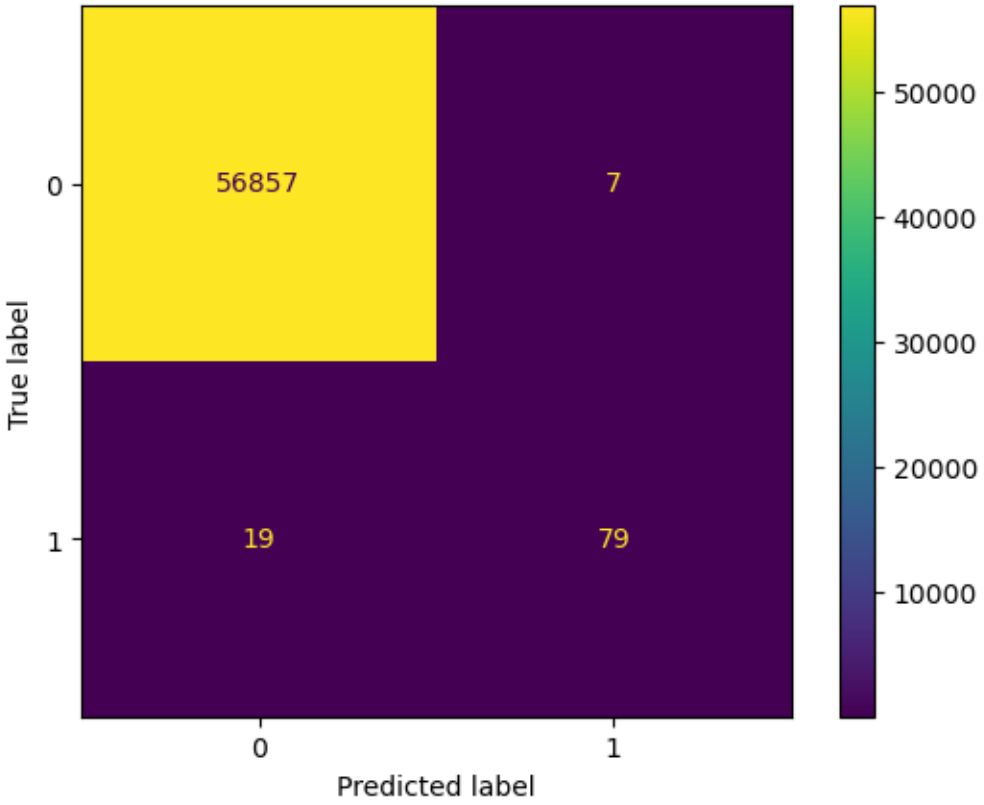
Interprétation: | Classe | Precision | Recall | F1-score | Support | Lecture rapide | |————|———|———|———-|———|————-| | 0 (normale) | 1.00 | 1.00 | 1.00 | 56864 | Seulement **13 vraies transactions** marquées à tort comme fraude (13 faux positifs). | | 1 (fraude) | 0.83 | 0.63 | 0.72 | 98 | • **Recall 0.63** : 62 fraudes détectées / 98 → **36 non détectées** (FN).• **Precision 0.83** : sur 75 alertes, **62 étaient vraies** → bon taux de détection. |

XGBoost: performances

[131]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y1_test, y1_pred_xgb1))

#Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y1_test,y1_pred_xgb1)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
              precision    recall  f1-score   support
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.92      0.81      0.86        98

    accuracy                           1.00     56962
   macro avg       0.96      0.90      0.93     56962
weighted avg       1.00      1.00      1.00     56962
```



Interprétation

Classe	Precision	Recall	F1-score	Support	Lecture rapide
0 (normale)	1.00	1.00	1.00	56864	Seulement **7 fausses alertes** pour des transactions normales.

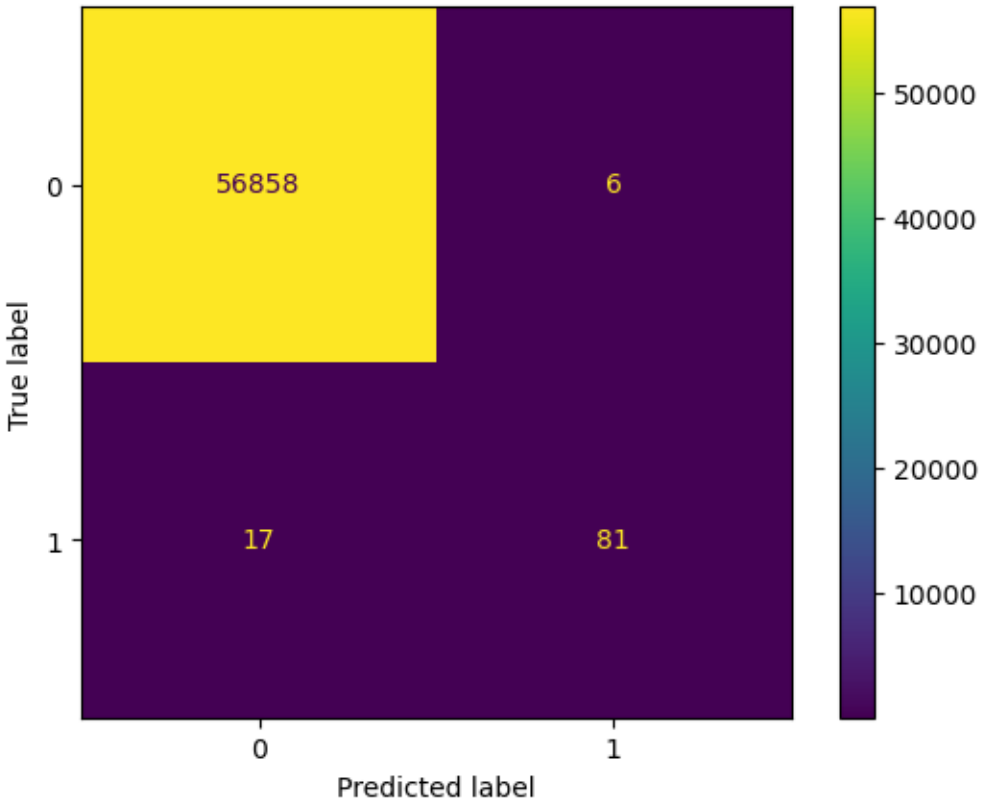
Classe	Precision	Recall	F1-score	Support	Lecture rapide
1 (fraude)	0.92	0.81	0.86	98	• **Recall 0.81** : 79 fraudes détectées / 98 → 19 FN.• Très bonne précision (0.92), très peu de fausses alertes.

Comparaison XGBoost vs Regression logistique: Nous constatons que **XGBoost est clairement plus performant que la regression logistique** car tout les scores de comparaison de XGBoost sont supérieurs à celui de la regression logistique.

Random Foreste: performances

[143]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y1_test, y1_pred_rf))

#Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y1_test,y1_pred_rf)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.93      0.83      0.88        98

    accuracy                           1.00     56962
   macro avg       0.97      0.91      0.94     56962
weighted avg       1.00      1.00      1.00     56962
```



Intrerprétation | Classe | Precision | Recall | F1-score | Support | Lecture rapide | |————|————|———|———-|———|————-| | 0 (normale) | 1.00 | 1.00 | 1.00 | 56864 | Seulement **6 faux positifs**, excellent. | | 1 (fraude) | 0.93 | 0.83 | 0.88 | 98 | • **Recall 0.83** : 81 fraudes détectées → 17 FN.• Très bon compromis entre détection et fiabilité. |

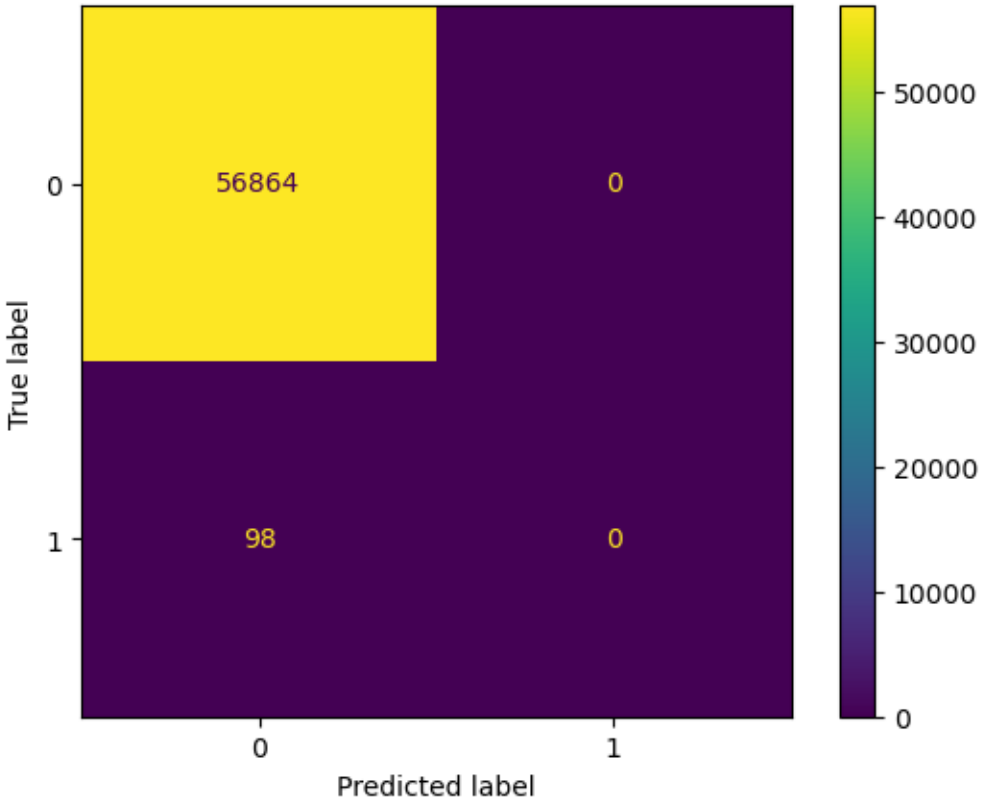
Comparaison XGBoost vs Random foreste: Nous constatons que **Random foreste est un peu plus performant que XGBoost** car tout les scores de comparaison de Randomforeste sont supérieurs à celui de XGBoost.

Modèle MLP: performances

[158]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y1_test, y1_pred_mlp, zero_division=0))

#Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y1_test,y1_pred_mlp)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	56864
1	0.00	0.00	0.00	98
accuracy			1.00	56962
macro avg	0.50	0.50	0.50	56962
weighted avg	1.00	1.00	1.00	56962



Interprétation

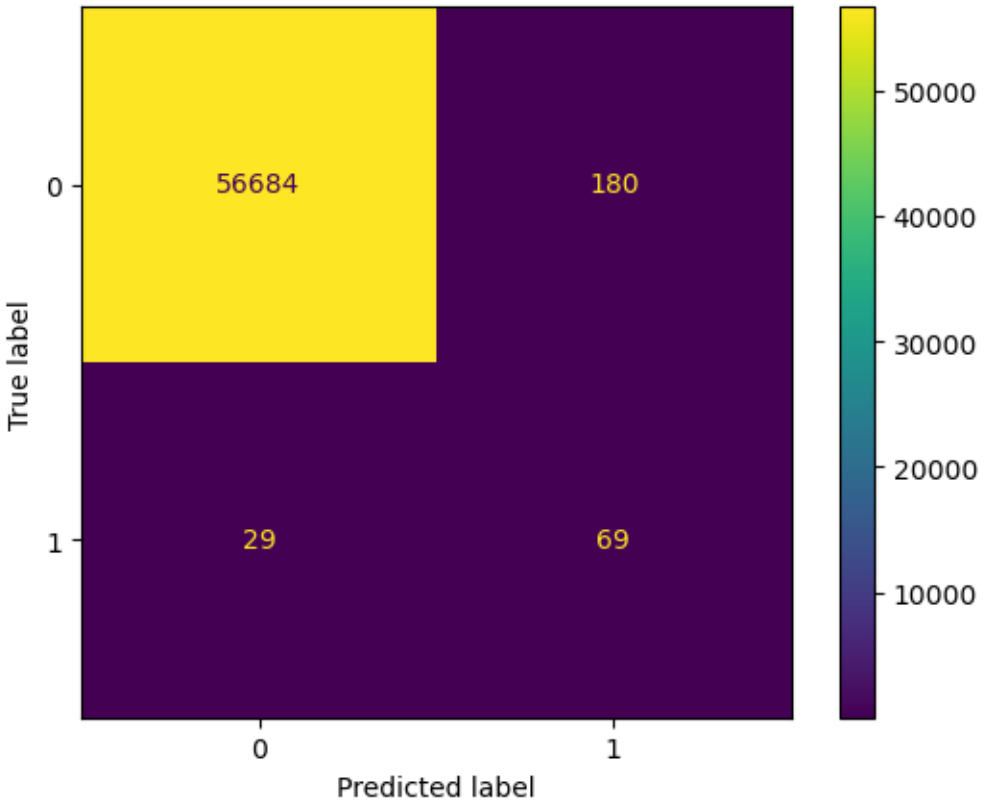
Classe	Precision	Recall	F1-score	Support	Lecture rapide
0 (normale)	1.00	1.00	1.00	56864	**0 faux positifs** : le modèle n'a détecté **aucune fraude**. • **Recall 0.00** : 98 fraudes **totalement ignorées**.• Modèle inutilisable dans ce contexte.
1 (fraude)	0.00	0.00	0.00	98	

Nous constatons que le modèle n'arrive pas à prédire les fraudes. Il a appris à détecter uniquement les non-fraudes, ce qui est dû au fort déséquilibre du dataset. Nous pourrions résoudre ce problème en utilisant un suréchantillonnage (SMOTE) ou en donnant plus de poids à la classe 1 (fraude). Toutefois, cela risquerait de biaiser les résultats de comparaison. Nous choisissons donc de conserver les résultats tels quels.

Modèle LightGBM: performances

[188]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y1_test, y1_pred_lgbm))

#Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y1_test,y1_pred_lgbm)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     56864
           1       0.28      0.70      0.40        98

    accuracy                           1.00     56962
   macro avg       0.64      0.85      0.70     56962
weighted avg       1.00      1.00      1.00     56962
```



Interprétation:

Classe	Precision	Recall	F1-score	Support	Lecture rapide
0 (normale)	1.00	1.00	1.00	56864	**180 fausses alertes** : trop de transactions normales sont marquées à tort.
1 (fraude)	0.28	0.70	0.40	98	• **Recall 0.70** : 69 fraudes détectées / 98 → 29 FN.• **Precision 0.28** : très **grand nombre de faux positifs** (250).

Ce modèle uniquement plus performant que MlP car arrive à détecté des fraudes mais moins performant que les autres modèles.

0.6.1.3 Rapport de comparaison des modèles

Rang	Précision (classe 1)	Recall (classe 1)	F1-score (classe 1)	Modèle
1	0.93	0.83	0.88	Random Forest
2	0.92	0.81	0.86	XGBoost
3	0.83	0.63	0.72	Régression logistique
4	0.28	0.70	0.40	LightGBM
5	0.00	0.00	0.00	MLP

Conclusion: Random forest est le modèle le plus adapté

0.6.2 DATASET 2: Fraud-Detection

0.6.2.1 Création et Entraînement des modèles

Modèle de regression logistique: De même que pour le dataste1, nous allons normaliser les variables pour ce modèle

[259]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X2_train_scaled = scaler.fit_transform(X2_train)
X2_test_scaled = scaler.transform(X2_test)
```

[285]:
```
from sklearn.linear_model import LogisticRegression  # importation of model

# Create a model
log_model2 = LogisticRegression(max_iter = 1000)
# entraînement
log_model2.fit(X2_train_scaled, y2_train)
# test de prediction
y2_pred_log = log_model2.predict(X2_test_scaled)
```

Modèle XGBost

[53]:
```
from xgboost import XGBClassifier

# Créer le modèle
xgb2 = XGBClassifier(eval_metric='logloss', random_state=42)

# Entraîner
xgb2.fit(X2_train, y2_train)

# Prédiction
y2_pred_xgb2 = xgb2.predict(X2_test)
```

[55]:
```
X2_test.columns
```

[55]:
```
Index(['merchant', 'category', 'amt', 'gender', 'state', 'lat', 'long',
       'city_pop', 'job', 'merch_lat', 'merch_long', 'hour', 'day_of_week',
```

```
       'day', 'month', 'age'],
      dtype='object')
```

Modèle Random foreste

[232]:
```
from sklearn.ensemble import RandomForestClassifier

# create the model
clf2 = RandomForestClassifier(n_estimators=14, max_depth=11, bootstrap=False,␣
 ↪random_state=101)

# Train the model
clf2.fit(X2_train, y2_train)

# prediction test
y2_pred_rf = clf2.predict(X2_test)
```

Modèle MLP

[248]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Création du modèle
model = Sequential()
model.add(Dense(64, input_dim=X2_train.shape[1], activation='relu'))  # Couche␣
 ↪d'entrée + 1ère couche cachée
model.add(Dense(128, activation='relu'))  # 2e couche cachée
model.add(Dense(1, activation='sigmoid'))  # Couche de sortie (classification␣
 ↪binaire)

# Compilation
model.compile(loss='binary_crossentropy', optimizer=Adam(), metrics=['accuracy'])

# Entraînement
history = model.fit(X2_train, y2_train, epochs=10, batch_size=32,␣
 ↪validation_split=0.2, verbose=1)
```

```
D:\Users\HP\anaconda3\Lib\site-packages\keras\src\layers\core\dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Epoch 1/10
32417/32417  70s 2ms/step
- accuracy: 0.9904 - loss: 32.2174 - val_accuracy: 0.9942 - val_loss: 0.0308
Epoch 2/10
32417/32417  74s 2ms/step
```

```
- accuracy: 0.9940 - loss: 0.0374 - val_accuracy: 0.9941 - val_loss: 0.0277
Epoch 3/10
32417/32417  70s 2ms/step
- accuracy: 0.9941 - loss: 0.0299 - val_accuracy: 0.9941 - val_loss: 0.0256
Epoch 4/10
32417/32417  71s 2ms/step
- accuracy: 0.9943 - loss: 0.0293 - val_accuracy: 0.9941 - val_loss: 0.0240
Epoch 5/10
32417/32417  70s 2ms/step
- accuracy: 0.9941 - loss: 0.0290 - val_accuracy: 0.9941 - val_loss: 0.0251
Epoch 6/10
32417/32417  72s 2ms/step
- accuracy: 0.9943 - loss: 0.0319 - val_accuracy: 0.9943 - val_loss: 0.0249
Epoch 7/10
32417/32417  76s 2ms/step
- accuracy: 0.9942 - loss: 0.0364 - val_accuracy: 0.9941 - val_loss: 0.0258
Epoch 8/10
32417/32417  70s 2ms/step
- accuracy: 0.9943 - loss: 0.0285 - val_accuracy: 0.9942 - val_loss: 0.0234
Epoch 9/10
32417/32417  73s 2ms/step
- accuracy: 0.9945 - loss: 0.0545 - val_accuracy: 0.9946 - val_loss: 0.0248
Epoch 10/10
32417/32417  74s 2ms/step
- accuracy: 0.9945 - loss: 0.0283 - val_accuracy: 0.9941 - val_loss: 0.0328
```

[273]:
```python
# Prédictions
y2_pred_proba = model.predict(X2_test)
y2_pred_mlp = (y2_pred_proba > 0.5).astype(int)
```

```
17367/17367  16s
812us/step
```

LightGBM

[271]:
```python
from lightgbm import LGBMClassifier
scale_pos_weight2 = np.sum(y2_train == 0) / np.sum(y2_train == 1)

lgbm_clf2 = LGBMClassifier(
    n_estimators=300,          # nombre total d'arbres (boosting rounds)
    learning_rate=0.05,        # shrinkage step (plus petit = + d'arbres)
    max_depth=-1,              # -1 = profondeur illimitée, on contrôle plutôt␣
 ↪avec num_leaves
    num_leaves=31,             # nbre max de feuilles par arbre (31 par défaut)
    subsample=0.8,             # échantillonnage lignes (aide à généraliser)
    colsample_bytree=0.8,      # échantillonnage colonnes
    scale_pos_weight=scale_pos_weight2,  # équilibre le poids de la classe 1
    objective="binary",
```

```
    random_state=101,
    n_jobs=-1,  # parallélisation CPU
    verbosity=0  # Masquer les messages pendant l'entraînement
)

# Entraînement du modèle
lgbm_clf2.fit(X2_train, y2_train)

# test prediction
y2_pred_lgbm = lgbm_clf2.predict(X2_test)
```

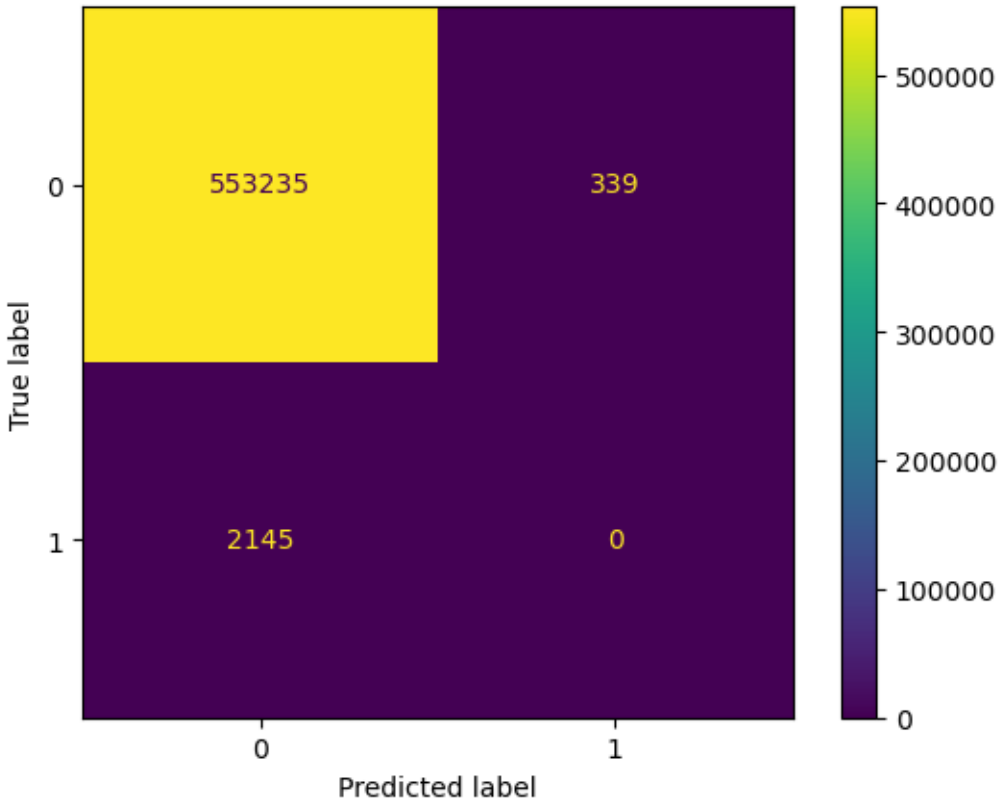
0.6.2.2 Évaluation des performances et choix du meilleur modèle dataset 2

[67]:
```
from sklearn import metrics
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
from sklearn.metrics import ConfusionMatrixDisplay
```

Regression logistique: performance

[283]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y2_test, y2_pred_log))

# Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y2_test, y2_pred_log)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    553574
           1       0.00      0.00      0.00      2145

    accuracy                           1.00    555719
   macro avg       0.50      0.50      0.50    555719
weighted avg       0.99      1.00      0.99    555719
```



Classe	Precision	Recall	F1-score	Support	Lecture rapide
0 (normale)	1.00	1.00	1.00	553 574	339 Fausses alerte. • **Recall 0.00** : Aucune fraude détectée → 2 145 fraudes manquées (FN).• **Precision 0.00** : Zéro prédiction correcte de fraude.
1 (fraude)	0.00	0.00	0.00	2 145	

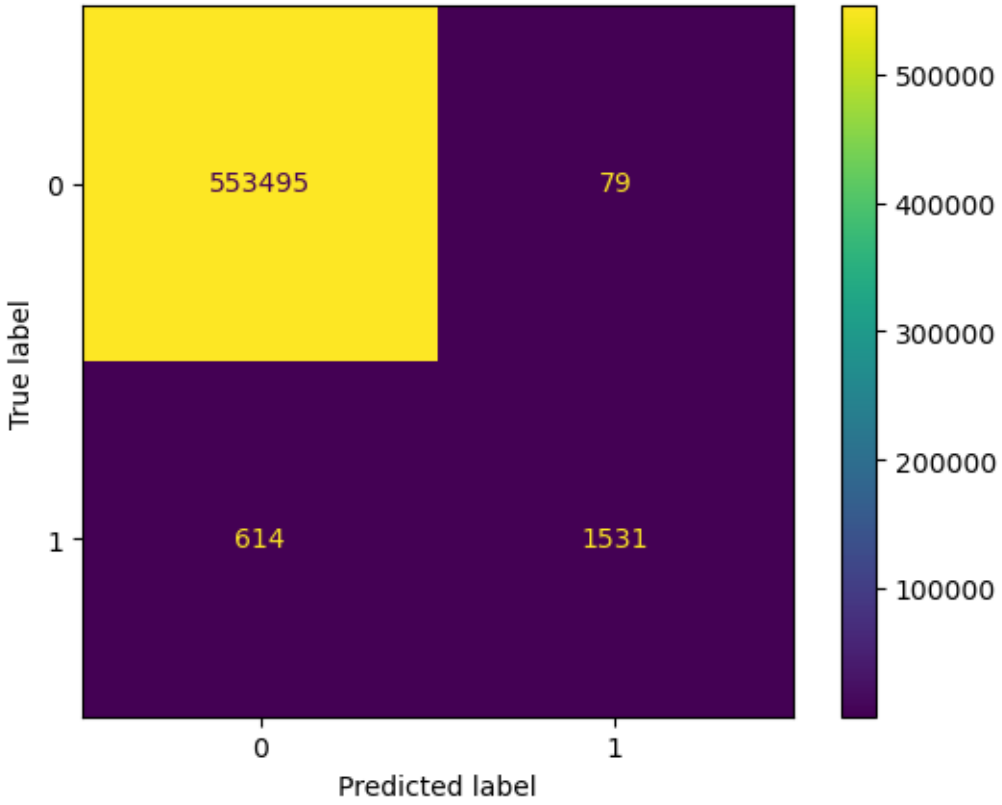
Le modèle n'est clairement pas performant pour ce dataset

XGBoost: performances

[69]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y2_test, y2_pred_xgb2))

# Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y2_test, y2_pred_xgb2)
```

```
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    553574
           1       0.95      0.71      0.82      2145

    accuracy                           1.00    555719
   macro avg       0.97      0.86      0.91    555719
weighted avg       1.00      1.00      1.00    555719
```



Classe	Precision	Recall	F1-score	Support	Lecture rapide
0 (normale)	1.00	1.00	1.00	553 574	75 fausses alertes sur plus de 553k → excellent.

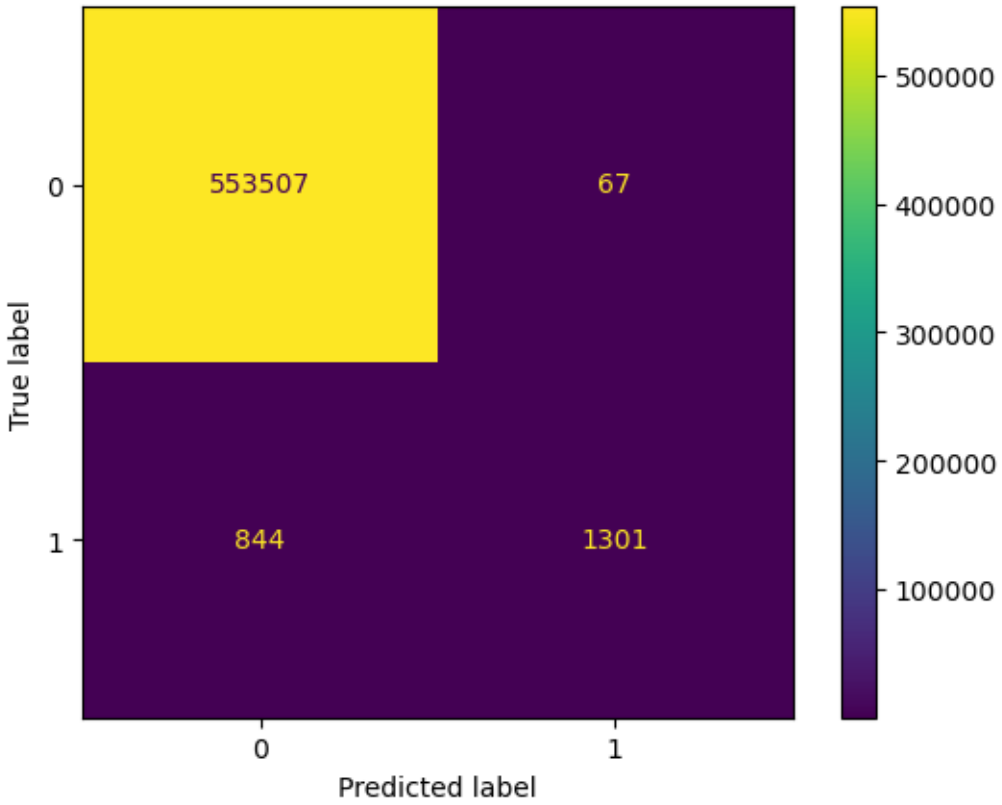
Classe	Precision	Recall	F1-score	Support	Lecture rapide
1 (fraude)	0.95	0.71	0.81	2 145	• **Recall 0.71** : 1 524 fraudes détectées / 2 145 → 621 FN.• **Precision 0.95** : Faible taux de faux positifs.

Clairement plus performant que La Régression logistique, XGboost est un bon modèle

Random Foreste: performances

[266]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y2_test, y2_pred_rf))

# Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y2_test, y2_pred_rf)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    553574
           1       0.95      0.61      0.74      2145

    accuracy                           1.00    555719
   macro avg       0.97      0.80      0.87    555719
weighted avg       1.00      1.00      1.00    555719
```



Interpétation:

Classe	Precision	Recall	F1-score	Support	Lecture rapide
0 (normale)	1.00	1.00	1.00	553 574	67 fausses alertes → très bon résultat.
1 (fraude)	0.95	0.61	0.74	2 145	• **Recall 0.61** : 1 301 fraudes détectées / 2 145 → 844 FN.• **Precision 0.95** : Faible taux de faux positifs.

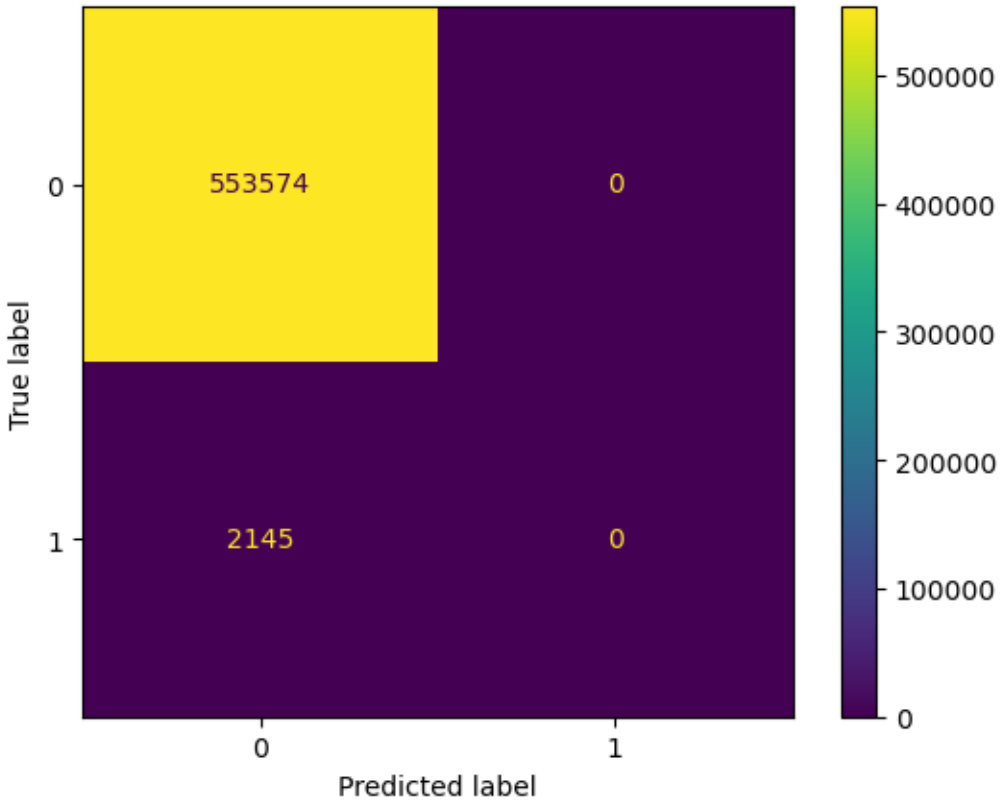
Un peu difficile à departagé mais **XGBoost l'emporte** car il donne un meilleur recall que Random forest malgré qu'ils ont la même précision.

Modèle MLP: performances

[275]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y2_test, y2_pred_mlp, zero_division=0))
```

```python
# Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y2_test, y2_pred_mlp)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    553574
           1       0.00      0.00      0.00      2145

    accuracy                           1.00    555719
   macro avg       0.50      0.50      0.50    555719
weighted avg       0.99      1.00      0.99    555719
```

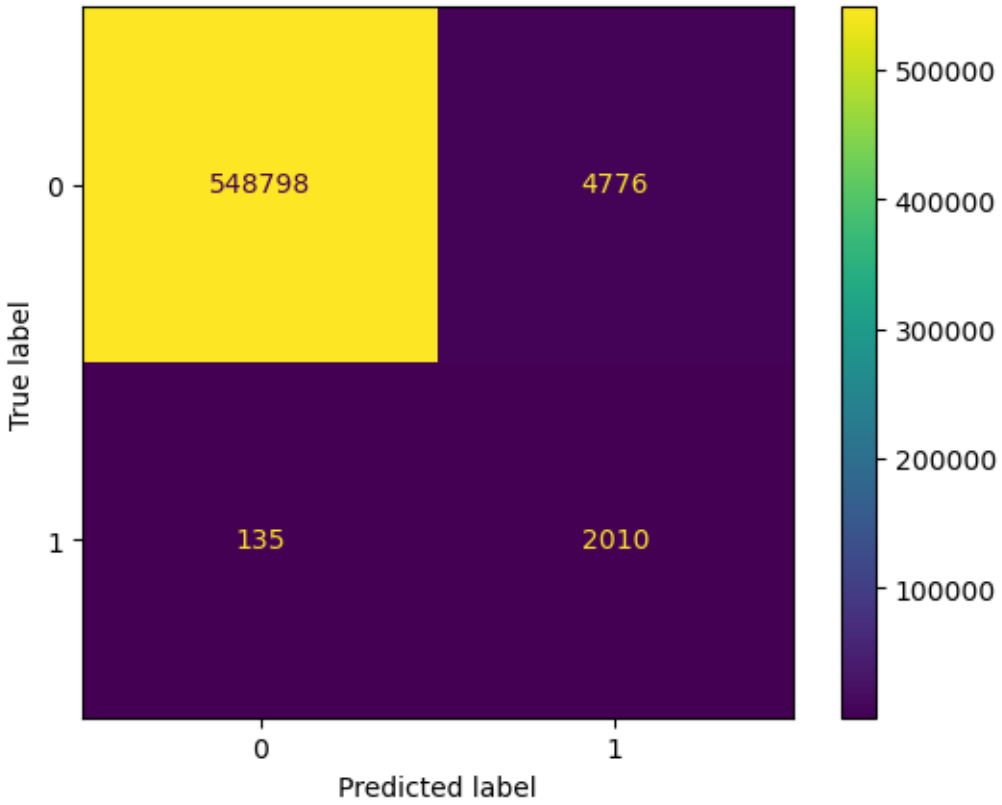


Classe	Precision	Recall	F1-score	Support	Lecture rapide
0 (normale)	1.00	1.00	1.00	553 574	Aucune transaction normale détectée comme fraude.
1 (fraude)	0.00	0.00	0.00	2 145	• **Recall 0.00** : aucune fraude détectée.• **Precision 0.00** : modèle inutile pour la détection.

Modèle LightGBM: performances

[278]:
```
# Afficher le rapport de classification
print("\nRapport de classification :")
print(classification_report(y2_test, y2_pred_lgbm))

# Afficher la matrice de confusion
confusion_matrix = metrics.confusion_matrix(y2_test, y2_pred_lgbm)
cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = confusion_matrix)
cm_display.plot()
plt.show()
```

```
Rapport de classification :
              precision    recall  f1-score   support

           0       1.00      0.99      1.00    553574
           1       0.30      0.94      0.45      2145

    accuracy                           0.99    555719
   macro avg       0.65      0.96      0.72    555719
weighted avg       1.00      0.99      0.99    555719
```



Classe	Precision	Recall	F1-score	Support	Lecture rapide
0 (normale)	1.00	0.99	1.00	553 574	4 776 fausses alertes → taux d'alerte élevé mais acceptable si tolérable.
1 (fraude)	0.30	0.94	0.45	2 145	• **Recall 0.94** : 2 010 fraudes détectées / 2 145 → 135 FN.• **Precision 0.30** : beaucoup de faux positifs.

0.6.2.3 Rapport comparatif des modèles

Rang	Précision (classe 1)	Recall (classe 1)	F1-score (classe 1)	Modèle
1	0.95	0.71	0.81	XGBoost
2	0.95	0.61	0.74	Random Forest
3	0.30	0.94	0.45	LightGBM

Rang	Précision (classe 1)	Recall (classe 1)	F1-score (classe 1)	Modèle
4	0.00	0.00	0.00	Régression Logistique
4	0.00	0.00	0.00	MLP (Réseau de neurones)

Remarques :

- XGBoost et Random Forest obtiennent les meilleurs F1-scores, avec un bon équilibre entre précision et rappel.
- LightGBM capte presque toutes les fraudes mais génère beaucoup de faux positifs.
- Régression Logistique et MLP ne détectent aucune fraude.

Conclusion: Le meilleur modèle est donc XGBoost

0.6.3 Choix du modèle final

Les deux meilleurs modèles sont: Random forest et XGBoost, nous allons donc faire une analyse croiser pour déterminer le modèle à deployer

Dataset n°	Modèle	Précision (classe 1)	Recall (classe 1)	F1-score (classe 1)
1	Random Forest	0.93	0.83	0.88
1	XGBoost	0.92	0.81	0.86
2	Random Forest	0.95	0.61	0.74
2	XGBoost	0.95	0.71	0.81

- XGBoost reste compétitif sur le dataset 1 Bien que random foreste à de meilleurs performances sur ce dataset.
- Sur le datset 2 malgré qu'ils ont la même précision le recalls de XGBoost surpasse celui de random Forest de 0.1 ce qui est considérable pour un problème de détection de fraude.
- De plus, Le dataset2 est bien plus grand que le premier, on peut donc dire que les modèles ont été mieux testés.

Nous allons donc déployer le modèle XGBoost du deuxième dataset

0.7 Phase de Déploiement

0.8 DÉPLOIEMENT DU MODÈLE

Nous allons donc sauvegarder le modèle

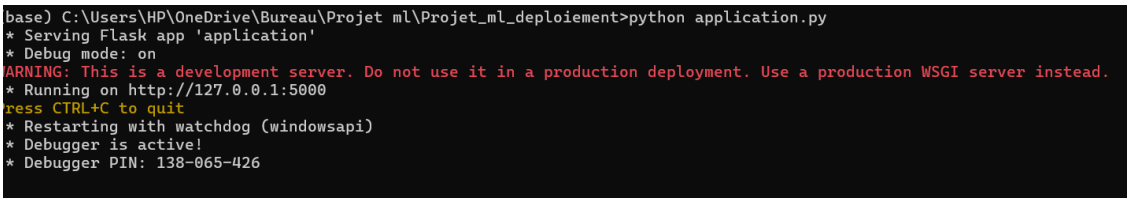
[363]:
```
import joblib

# Sauvegardons le modèle à déployer
joblib.dump(xgb2, "xgb2_model.joblib")
```

[363]:
```
['xgb2_model.joblib']
```

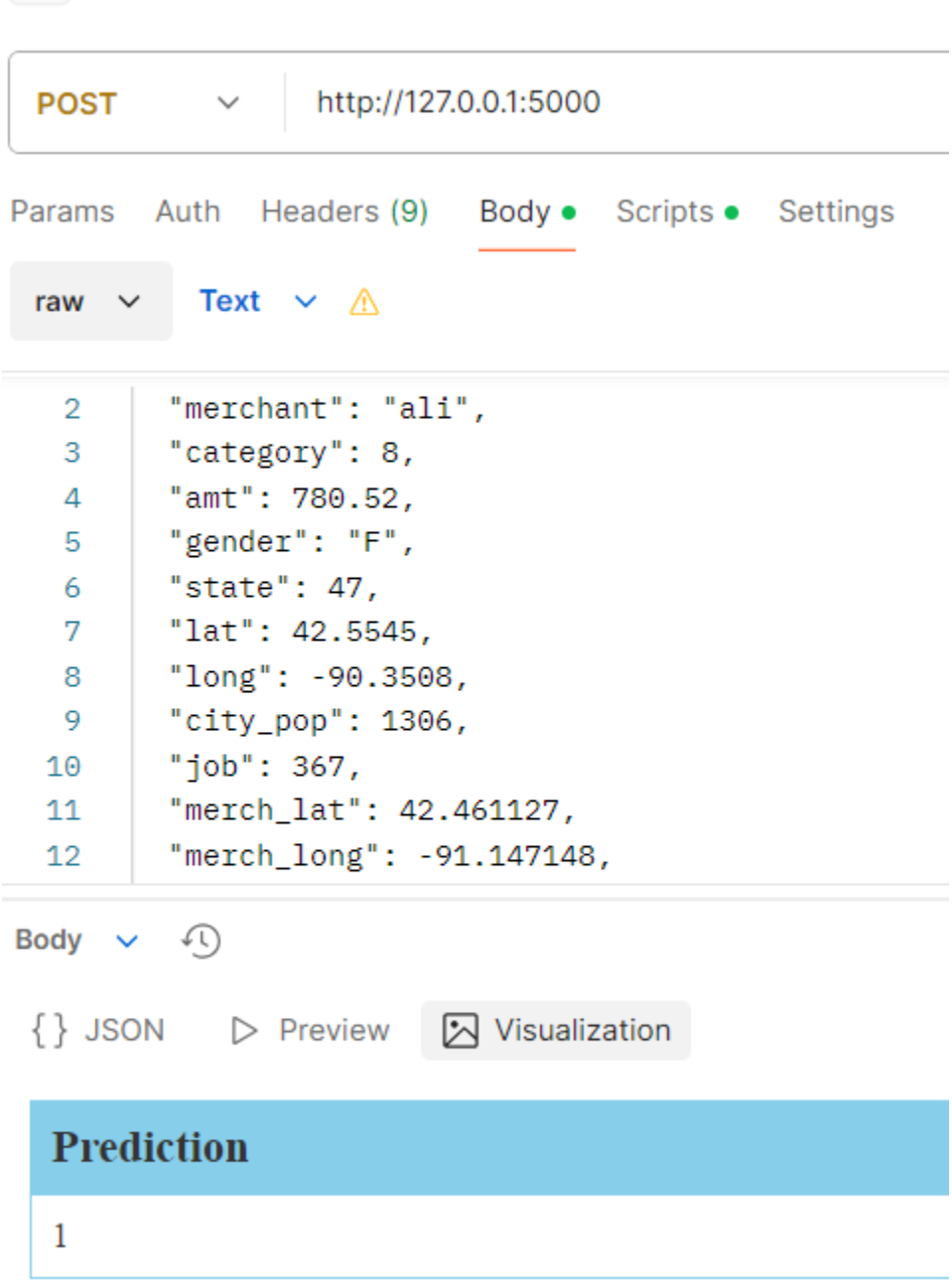
0.8.1 Choix de l'API backend (Flask)

Il s'agit de l'application qui nous permettra d'envoyer des requêtes à notre modèle de machine learning. Nous l'avons développé en Python et nous avons utilisé le framwork flask. (Vous pouvez voir un aperçu du code dans 6.2 ou consulter le fichier application)

Ensuite nous avons procédé à un test pour voir si notre modèle est capable de recevoir des requêtes et nous envoyer sa réponse, pour ce fait nous avons utilisé un logiciel appelé **postman**:



Nous constatons que l'application a été lancé avec succès en local, nous allons juste copier l'URL et le coller dans postman et envoyer une requête à notre modèle.



Nous voyons clairement que le modèle arrive a prédire. **Donc notre modèle fonctionne très bien.**

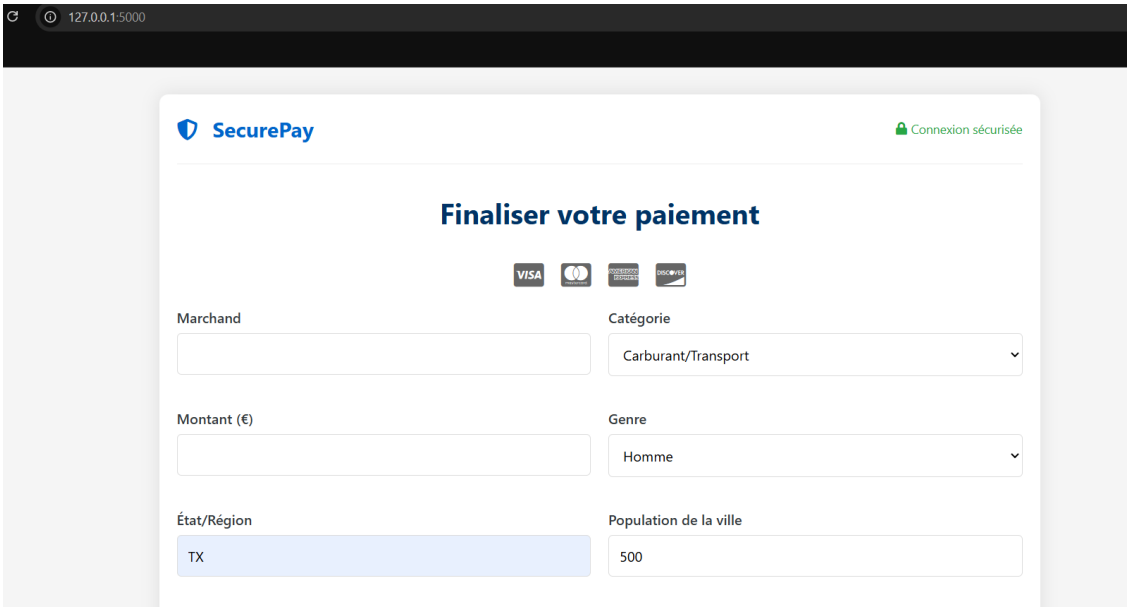
0.8.2 Interface utilisateur

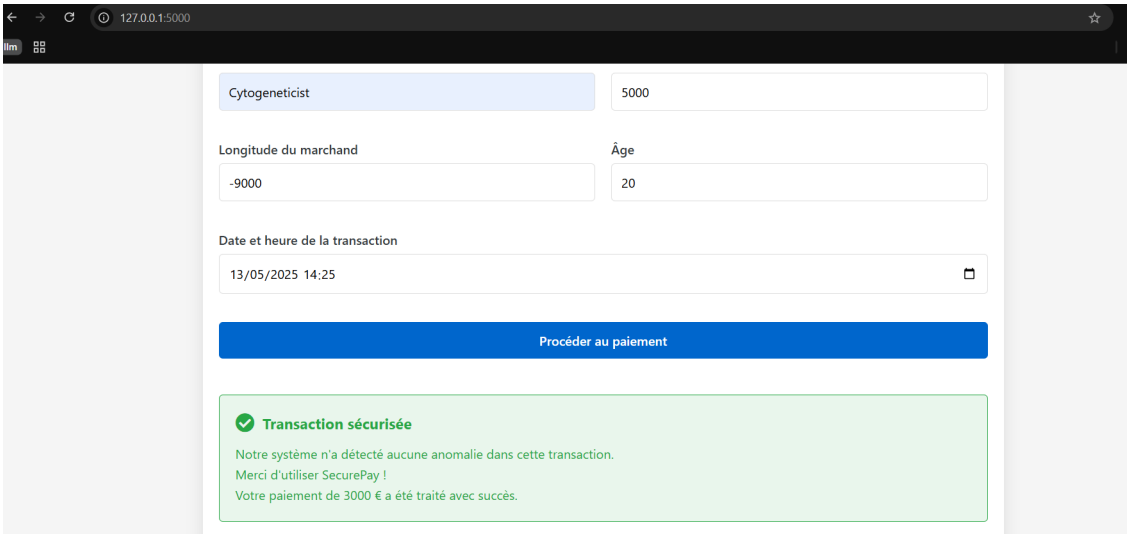
Pour allez plus loin dans notre projet, nous avons conçu une interface utilisateur moderne pour rendre le processus plus interactif. Nous avons utilisé du **HTML, CSS, JS** et nous avons connecter notre application flask à l'interface utilisateur pour qu'elle affiche le contenu de notre fichier html au lancement via le port **GET**.
Voici un aperçu de notre fichier flask

```python
# Chargement du modèle
model = joblib.load("xgb2_model.joblib")

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'GET':
        # Servir la page HTML pour l'interface utilisateur
        return render_template_string(get_html_content())
    elif request.method == 'POST':
        try:
            # Récupérer les données JSON envoyées via POST
            data = request.get_json()

            # Vérifier que les données sont correctes
            if not data:
                return jsonify({'error': 'No input data provided'}), 400

            # Appliquer le prétraitement
            input_df = preprocess_input(data)

            # Faire la prédiction
            prediction = model.predict(input_df)

            # Retourner la prédiction sous forme JSON
            return jsonify({'prediction': int(prediction[0])})

        except Exception as e:
            return jsonify({'error': str(e)}), 400

if __name__ == '__main__':
    # Écrire le HTML dans un fichier pour être utilisé par Flask
    try:
        f.write(get_html_content())
    except Exception as e:
        print(f"Attention: Impossible d'écrire le fichier HTML: {e}")
```

Nous avons relancé l'application et collé URL dans notre navigateur pour afficher l'interface et procéder à un test.

Voici le resultat





0.8.3 Déploiement sur AWS

À ce stade, notre application et notre modèle fonctionnent correctement en local. Nous allons à présent procéder à leur déploiement sur AWS afin de les rendre accessibles de n'importe où, par n'importe qui.

- Système d'exploitation choisi : Ubuntu 22.04 LTS
- Étapes effectuées : Création d'une instance t3.micro avec une clé SSH (.pem).

  Connexion

```
[ ]: ssh -i "cle.pem" ubuntu@<IP_PUBLIQUE>
```

Dans notre cas, Ip= 13.50.13.122

[ ]:
```
sudo apt update && sudo apt upgrade
sudo apt install python3 python3-pip python3-venv
pip install flask flask-cors pandas joblib scikit-learn
```

Après installation des packages necessaires nous avons lancé l'application flask et nous l'avons fait tourné même après fermeture du terminal avec les commandes:

[ ]:
```
python application.py
nohup python application.py &
```

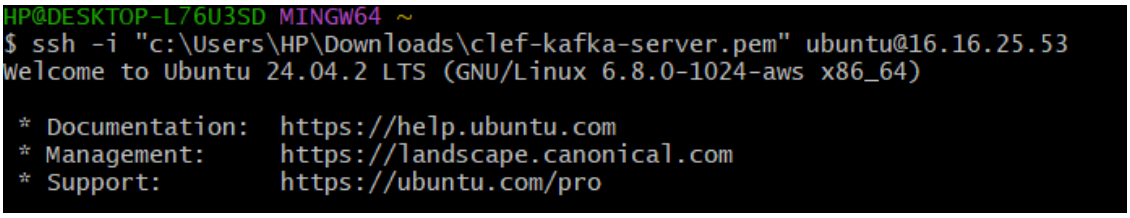
Maintenant notre application est accessible via l'url: http://13.50.13.122:5000/ Vous pouvez la consulter en direct.

0.9 INTÉGRATION D'APACHE KAFKA

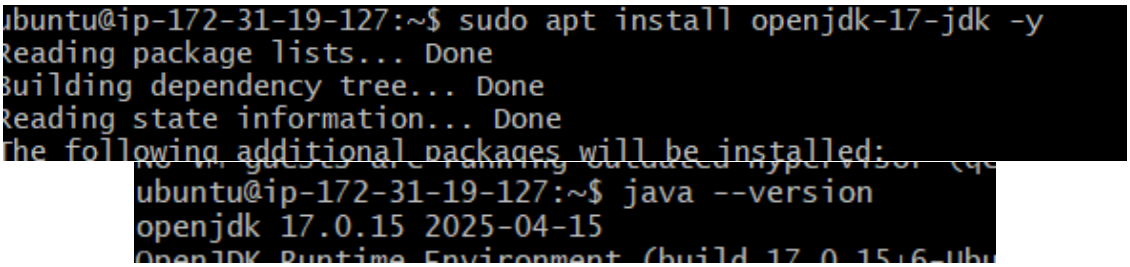
Maintenant que notre application fonctionne pour le client, nous allons donc faire une deuxième interface pour la banque et envoyer des transactions en temps réel grâce à kafka. Nous allons commencer par créer une deuxième machine virtuelle EC2 sure AWS et installer Kafka.

0.9.1 Installation et configuration de kafka

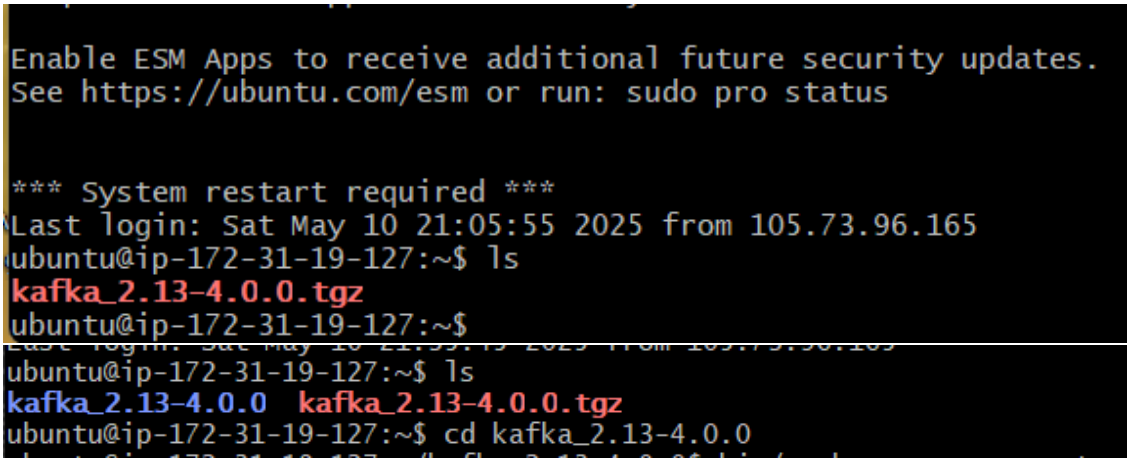
Connexion



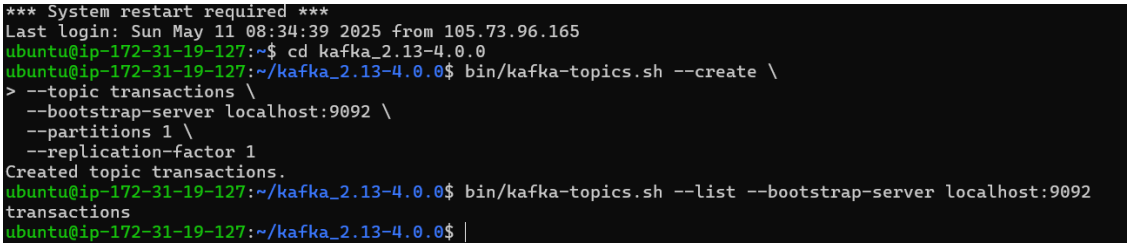
Installation de JAvA: Kafka neccessite Java pour fonctionner donc nous allons installer openjdk



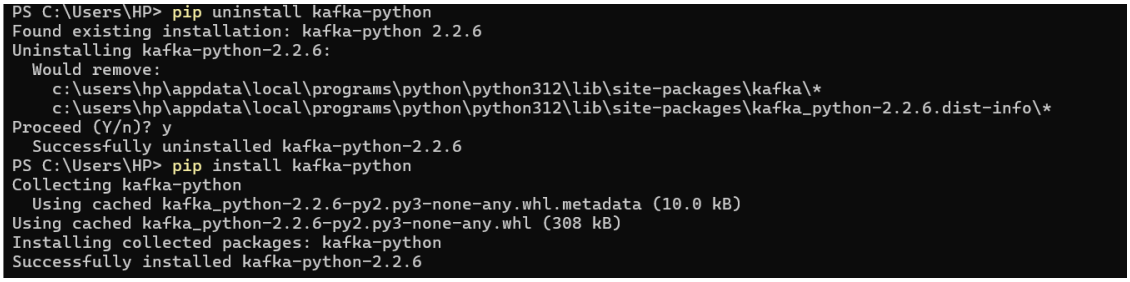
téléchargement de kafka Nous avons d'abord téléchargé kafka en sur le site officiel et nous l'avons envoyé sure notre VM



Création du topic transaction Un topic est comme un "canal" ou une "file de messages" nommée, dans lequel les producteurs envoyerons des messages et que les consommateurs liront.



Installation de Kafka en local



Création du simulateur Voir (code)

Le simulateur joue le rôle d'un générateur de données. Il reproduit le comportement de clients bancaires en envoyant en continu des transactions artificielles mais réalistes. Ces données sont envoyées dans un topic Kafka afin d'alimenter le système de détection de fraude en temps réel.

Grâce à ce simulateur, il est possible de :

- Tester le système sans utiliser de vraies données sensible.

- Générer un grand volume de données variées.

- Observer le fonctionnement du modèle de détection en conditions proches du réel.

Il représente donc une source dynamique et contrôlable de données, essentielle pour valider les performances du système.

```
HP@DESKTOP-L76U3SD MINGW64 ~
$ ssh -i "c:\Users\HP\Downloads\clef-kafka-server.pem" ubuntu@16.16.25.53
Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 6.8.0-1024-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro
```

Test du simulateur et la connection avec kafka Dans cette parties nous allons utiliser le simulateur pour envoyer des requêtes sur Kafka et tester si ces requêtes sont bien envoyés.

```
C:\Users\HP\OneDrive\Bureau\Fraud_detect>python simulateur_kafkaProducer.py
Simulateur en cours d'exécution... Appuyez sur Ctrl+C pour arrêter.
Génération d'une nouvelle transaction : {'merchant': 'jumia', 'category': 'en
tertainment', 'amt': 245.1, 'gender': 'F', 'state': 'Alaska', 'lat': 30.8401,
 'long': -69.3162, 'city_pop': 996504, 'job': 'writer', 'merch_lat': 43.86874
1, 'merch_long': -99.960142, 'trans_date_trans_time': '11/05/2025 19:47', 'ag
e': 61}
✔ Transaction envoyée : {'merchant': 'jumia', 'category': 'entertainment', '
amt': 245.1, 'gender': 'F', 'state': 'Alaska', 'lat': 30.8401, 'long': -69.31
62, 'city_pop': 996504, 'job': 'writer', 'merch_lat': 43.868741, 'merch_long'
: -99.960142, 'trans_date_trans_time': '11/05/2025 19:47', 'age': 61}
Génération d'une nouvelle transaction : {'merchant': 'amazon', 'category': 'f
ood', 'amt': 360.84, 'gender': 'M', 'state': 'Nebraska', 'lat': 26.8654, 'lon
g': -75.8791, 'city_pop': 260490, 'job': 'accountant', 'merch_lat': 48.681403
, 'merch_long': -123.880409, 'trans_date_trans_time': '11/05/2025 19:47', 'ag
e': 29}
```

Nous constatons que le producteur envoie effectivement des requêtes sur Kafka.

Consommateur:

Le consommateur est chargé de lire les transactions en temps réel depuis le topic Kafka, et de les transmettre au modèle de machine learning déployé sous forme d'API. Après avoir reçu la prédiction (fraude ou non fraude), le consommateur va l'enregistrer dans une base de données pour affichage dans l'interface de la banque.

Il assure plusieurs fonctions essentielles :

- Écouter en continu les nouvelles données dans Kafka,

- Communiquer avec l'API de détection de fraude,

- Assurer un lien fluide entre la source de données et le modèle ML.

En résumé, le consommateur est le pont entre Kafka et l'API de prédiction, et permet d'avoir un traitement des transactions entièrement automatisé et en temps réel. **Code**

```python
from kafka import KafkaConsumer
import requests
import json

# IP publique du serveur Kafka EC2
KAFKA_BROKER = '16.16.25.53:9092'
TOPIC = 'transactions'

# URL du modèle ML (API Flask sur AWS)
ML_API_URL = 'http://13.50.13.122:5000/predict'
# === Kafka Consumer Configuration ===
consumer = KafkaConsumer(
    TOPIC,
    bootstrap_servers=KAFKA_BROKER,
    value_deserializer=lambda m: json.loads(m.decode('utf-8')),
    auto_offset_reset='latest',
    enable_auto_commit=True,
    group_id='fraud-detector-group'
)

print("📡 Consommateur Kafka lancé...")

# === Boucle principale ===
for message in consumer:
    transaction = message.value
    print(f"\n📩 Transaction reçue : {transaction}")

    try:
        response = requests.post(ML_API_URL, json=transaction)
        if response.status_code == 200:
            prediction = response.json()
            print(f"✅ Réponse du modèle : {prediction}")
        else:
            print(f"❌ Erreur HTTP : {response.status_code} - {response.text}")
    except Exception as e:
        print(f"⚠️ Erreur lors de l'envoi au modèle : {e}")
```

Lancement

```
24  for message in consumer:

PROBLEMS 2  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

ty_pop': 646166, 'job': 'developer', 'merch_lat': 44.340895, 'merch_long': -112.905793, 'trans_date_trans_time': '20/05/2025 08:08', 'age': 27}
✅ Réponse du modèle : {'prediction': 0}

📩 Transaction reçue : {'merchant': 'uber', 'category': 'health', 'amt': 623.25, 'gender': 'F', 'state': 'Connecticut', 'lat': 41.5999, 'long': -99.9192, 'city_p
op': 977655, 'job': 'nurse', 'merch_lat': 45.140185, 'merch_long': -115.287617, 'trans_date_trans_time': '20/05/2025 08:08', 'age': 26}
✅ Réponse du modèle : {'prediction': 0}

📩 Transaction reçue : {'merchant': 'airbnb', 'category': 'transport', 'amt': 601.13, 'gender': 'F', 'state': 'Alaska', 'lat': 29.4515, 'long': -121.1878, 'city_
pop': 874255, 'job': 'artist', 'merch_lat': 37.335303, 'merch_long': -108.853142, 'trans_date_trans_time': '20/05/2025 08:08', 'age': 43}
✅ Réponse du modèle : {'prediction': 0}
```

Nous constatons que le consommateur fonctionne très bien et nous envoie effectivement les réponses de notre modèle.

0.10 INTÉGRATION D'UNE BASE DE DONNÉE NOSQL

0.10.1 Objectif et architecture fonctionnelle

Cette couche assure la **persistance** des transactions et de leurs prédictions dans `Google Firestore`. Le flux complet est le suivant :

1) **Production :** le simulateur publie des transactions aléatoires sur un topic Kafka ;
2) **Consommation :** le consommateur Kafka lit les données en temps réel ;
3) **Analyse :** chaque transaction est envoyée au modèle XGBoost ; l'API renvoie *is_fraud = true/false* et la probabilité associée ;
4) **Persistance :** le module `firebase_layer` écrit la transaction *et* le résultat dans Firestore.

0.10.2 Structure du module `firebase_layer`

```
backend/
  firebase_layer/
      client.py              # Connexion Firestore
      writer.py              # Fonctions d'écriture
      firestore.indexes.json # Index composites
      firestore.rules        # Règles de sécurité
```

Rôle des fichiers

- `client.py` – initialise le SDK Firebase et expose la collection `fraud_predictions`.
- `writer.py` – construit un document structuré et gère l'insertion (simple ou batch).
- `firestore.indexes.json` – déclare les index composites optimisant les requêtes analytiques.
- `firestore.rules` – autorise *lecture publique*, mais réserve l'écriture aux comptes-service backend.

0.10.3 Schéma de la collection `fraud_predictions`

Champ	Type	Description
`tx_id`	String	UUID unique de la transaction
`amount`	Number	Montant de la transaction
`is_fraud`	Boolean	Résultat de la prédiction
`merchant`	String	Nom du commerçant
`category`	String	Catégorie de la transaction
`state`	String	État géographique
`gender`	String	Genre du client
`age`	Number	Âge du client
`job`	String	Profession du client
`city_pop`	Number	Population de la ville
`lat`, `long`	Number	Coordonnées client

Champ	Type	Description
merch_lat, merch_long	Number	Coordonnées commerçant
trans_datetime	Timestamp	Date / heure réelle
hour_of_day	Number	Heure (0–23)
day_of_week	Number	Jour (0=Lun, 6=Dim)
timestamp	Timestamp	Horodatage d'insertion
raw_features	Map	Transaction brute complète
prediction_result	Map	Détails de la réponse modèle

0.10.4 Index composites configurés

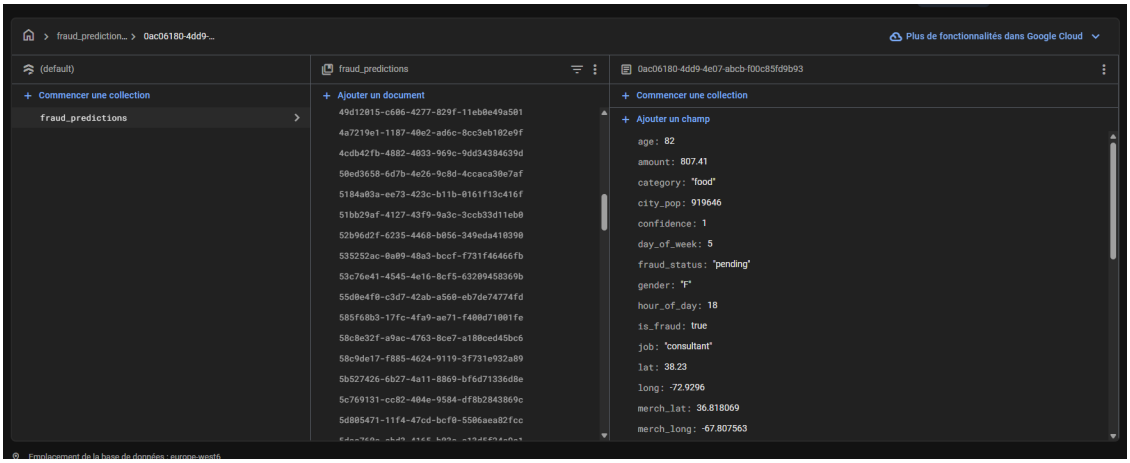
Index	Champs (ordre)	Cas d'usage
fraud_by_time	is_fraud ASC, timestamp DESC	Historique des fraudes
status_by_time	fraud_status ASC, timestamp DESC	Suivi de vérification
merchant_by_time	merchant ASC, timestamp DESC	Analyse par commerçant
category_by_time	category ASC, timestamp DESC	Analyse par catégorie
state_by_time	state ASC, timestamp DESC	Répartition géographique
hourly_pattern	hour_of_day ASC, timestamp DESC	Patterns horaires
amount_by_time	amount ASC, timestamp DESC	Analyse par montant

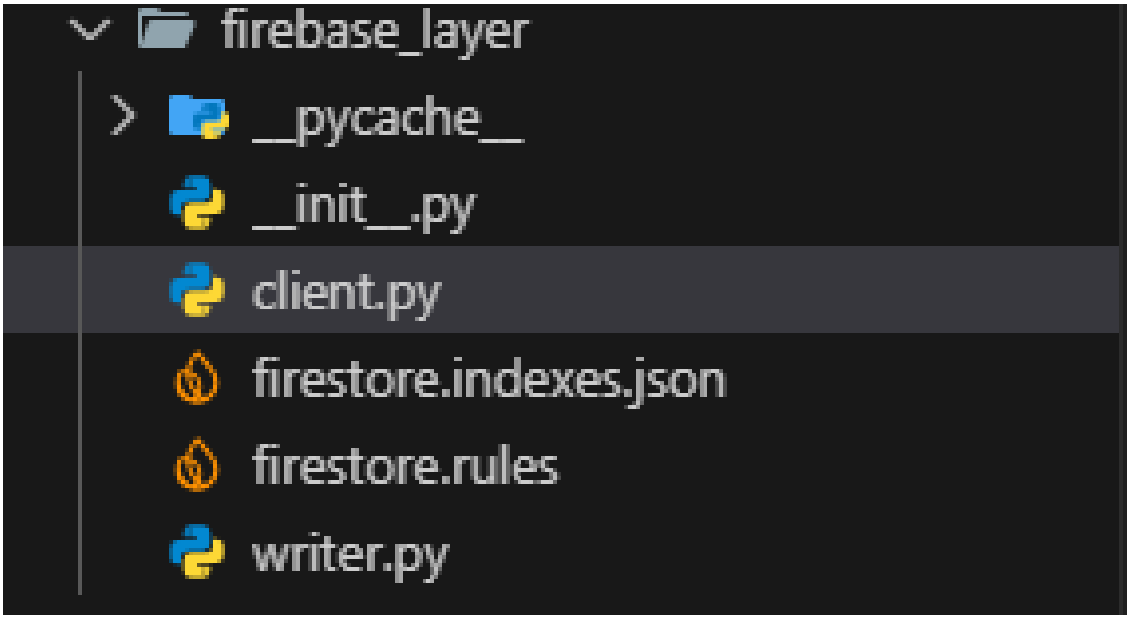
0.10.5 Installation

- **Prérequis :** Python 3.10+, environnement virtuel, projet Firebase avec Firestore activé.
- **Installation rapide** :

```
python -m venv .venv && source .venv/bin/activate
pip install firebase-admin google-cloud-firestore
export GOOGLE_APPLICATION_CREDENTIALS=/chemin/key.json
```

Cette intégration complète le pipeline temps réel : Kafka → API ML → Firestore, assurant un *stockage sécurisé* et des *requêtes performantes* pour les tableaux de bord.





0.11 INTERFACE DASHBOARD

0.11.1 Introduction

Le tableau de bord de détection de fraude est une application **frontend** moderne développée avec **Next.js 15**. Elle permet d'observer *en temps réel* l'arrivée des transactions, d'identifier rapidement les fraudes potentielles et d'explorer les dimensions *temporelles, géographiques et financières* des données.

0.11.2 Architecture technique

Technologies principales

- **Next.js 15**, **React 19**, **TypeScript 5**
- **Tailwind CSS 4** – styles *mobile-first*

- **Zustand** – gestion d'état centrale
- **Firebase / Firestore** – base de données temps réel
- **Recharts**, **Nivo**, **React-Simple-Maps** – visualisations
- **TanStack Query**, **React Hook Form + Zod**, **Framer Motion**, **next-themes**

Dépendances complémentaires

- `@nivo/core`, `@nivo/heatmap`
- `d3-geo`, `react-simple-maps`
- `date-fns`, `clsx`, `class-variance-authority`, `tw-animate-css`

0.11.3 Structure du projet

1. **App Router** : pages et routes Next.js (racine, tableau de bord, vues temps réel).
2. **Composants** : briques UI réutilisables (layout, header, sidebar, sélecteur de période, cartes, tableaux, graphiques).
3. **Lib** : configuration Firebase, store Zustand, fonctions utilitaires.
4. **Types** : déclarations TypeScript (ex. pour la cartographie).

0.11.4 Flux de données

- **Firestore** alimente en continu le hook `useFraudStream`.
- Le composant `Stream` distribue les données aux cartes KPI, aux graphiques et au tableau.
- Les interactions utilisateur (filtres, thème, sidebar) modifient l'état global stocké dans **Zustand**.

0.11.5 Gestion d'état avec Zustand

Le *store* centralise : thème clair/sombre, filtres temporels, filtres fraude, état UI (sidebar, recherche). L'API minimaliste de Zustand évite les rendus inutiles et reste 100 typée.

0.11.6 Composants principaux

DashboardLayout Structure générale : entête fixe, sidebar repliable, zone de contenu flexible.

DashboardHeader Logo, recherche, notifications, bascule thème (*sticky* avec *backdrop-blur*).

DashboardSidebar Navigation, filtres rapides.

Stream Conteneur temps réel distribuant les données.

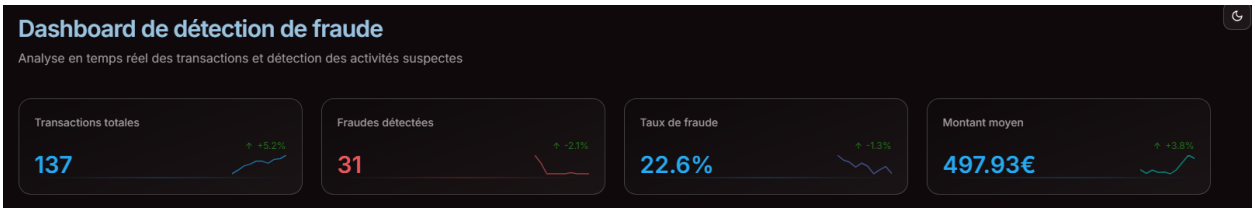
0.11.7 Visualisations et graphiques détaillés

1. KPICards (Cartes d'indicateurs clés)
Type de graphique : Cartes statistiques

Description : Affiche des indicateurs clés de performance (KPI) en temps réel :

- Nombre total de transactions
- Nombre de fraudes détectées
- Montants moyens / totaux
- Taux de fraude (%)

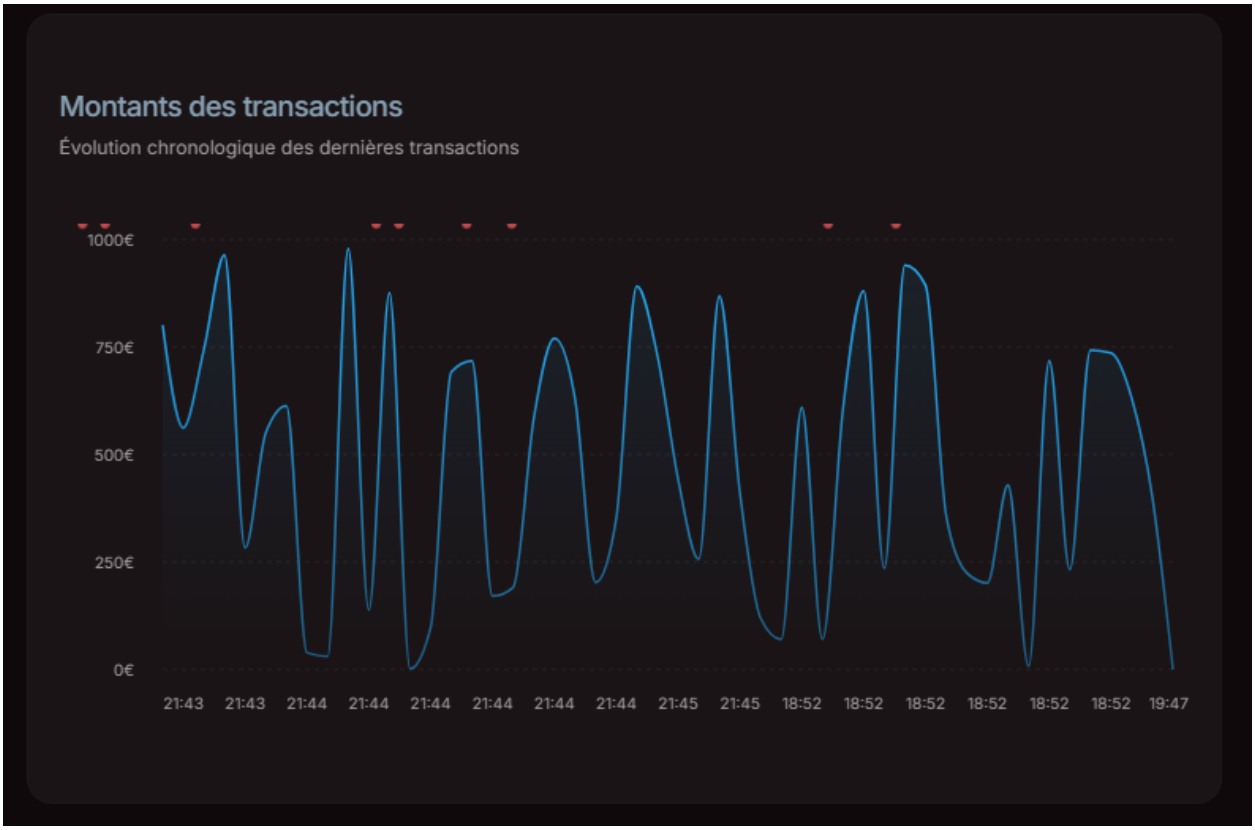
Utilité : Vue d'ensemble rapide et synthétique de l'état du système de détection.



2. AmountLine (Graphique des montants)
Type de graphique : Courbe linéaire (Recharts)

Description : Affiche l'évolution du montant des transactions au fil du temps (par minute, heure, jour selon la période sélectionnée).

Utilité : Permet de repérer des pics d'activité suspects ou des irrégularités dans les montants.



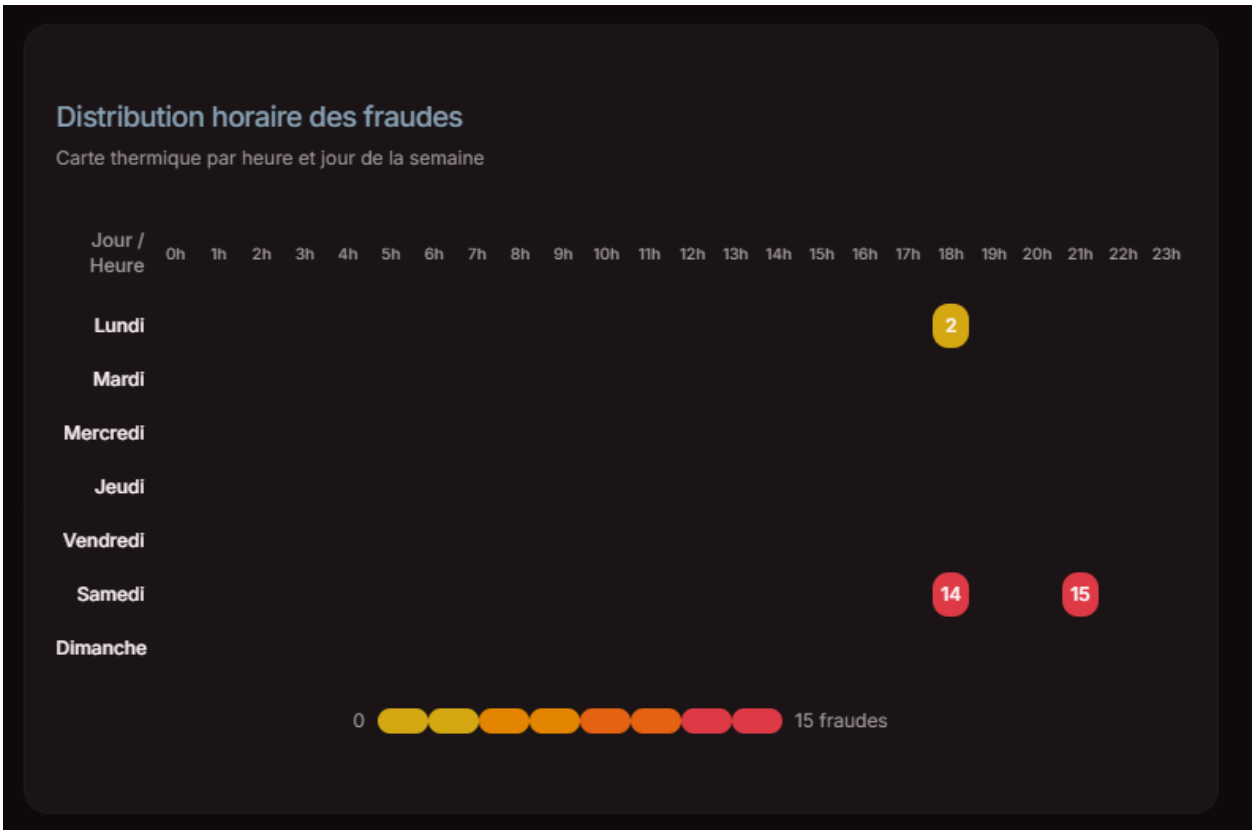
3. HourHeatmap (Carte thermique horaire)
Type de graphique : Carte thermique (Nivo)

Description : Montre la concentration des fraudes détectées par jour de la semaine et par heure de la journée.

Axes :

- **Axe X :** Heures (0 h – 23 h)
- **Axe Y :** Jours (Lundi – Dimanche)
- **Couleur :** Intensité du nombre de fraudes

Utilité : Identifier les plages horaires les plus critiques (fraudes récurrentes la nuit, en week-end, etc.).



4. MapFraud (Carte des fraudes)

Type de graphique : Carte géographique (react-simple-maps + d3-geo)

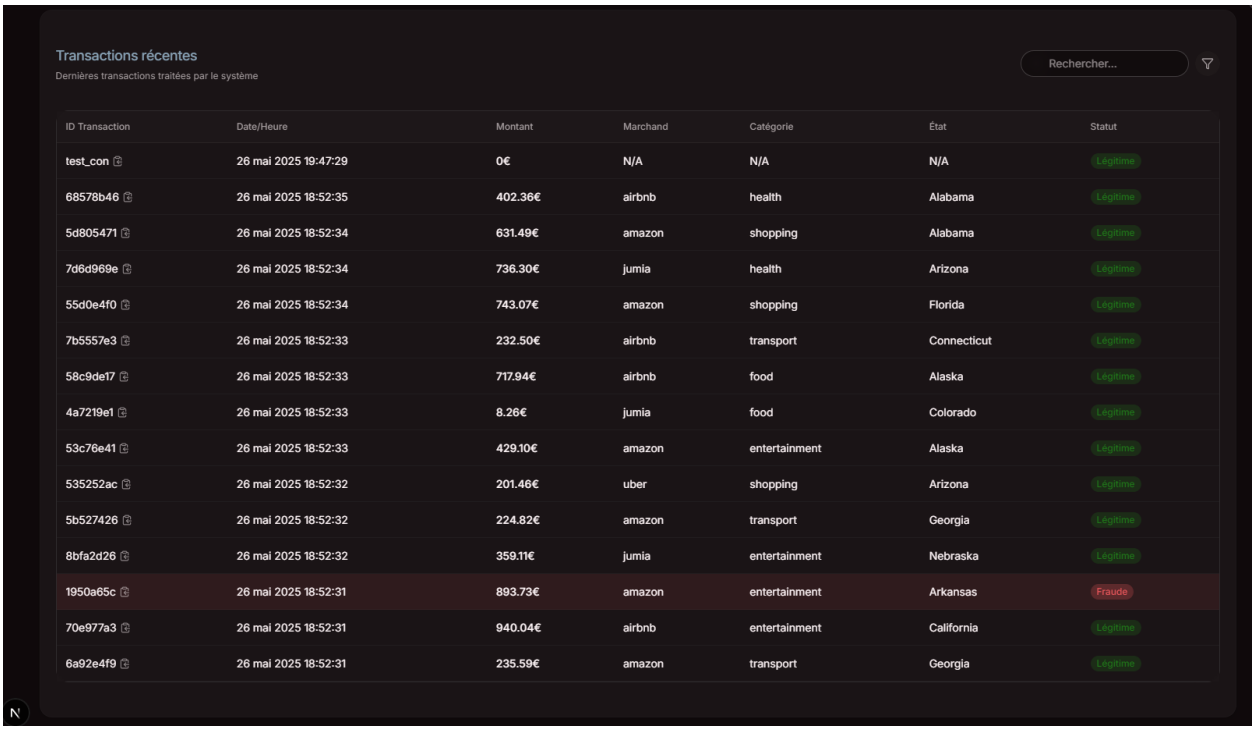
Description : Localise les fraudes détectées par région, pays ou ville en fonction des données de géolocalisation.

Utilité : Visualiser les zones géographiques à risque ou les clusters d'activités frauduleuses.

5. LiveTable (Tableau des transactions)
Type de graphique : Tableau de données (tri / filtrage)

Description : Liste détaillée des dernières transactions reçues : ID, montant, heure, statut (fraude / non), localisation, etc.

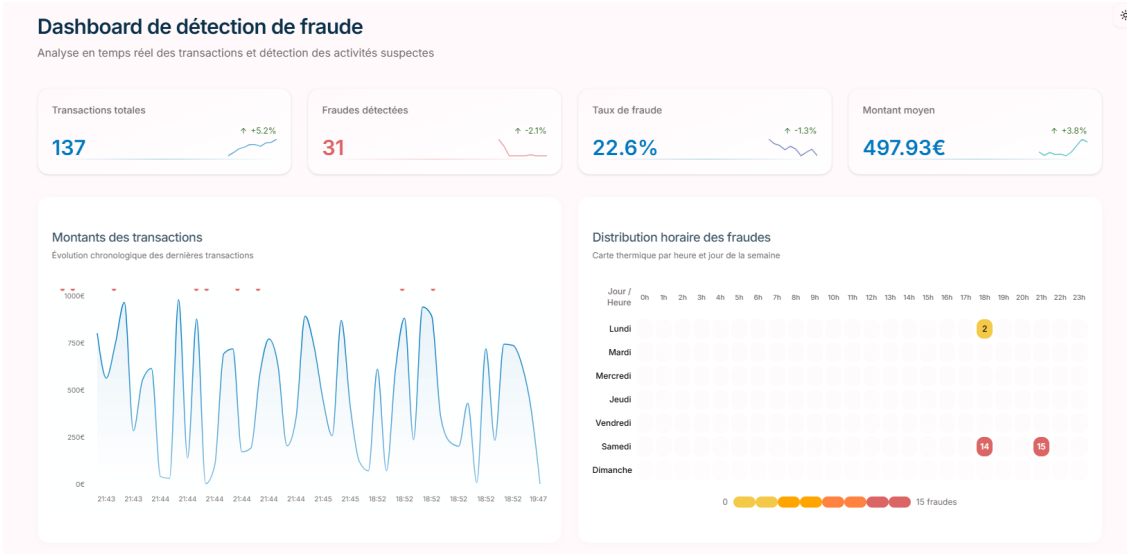
Utilité : Consultation directe des données transactionnelles, validation manuelle ou export.

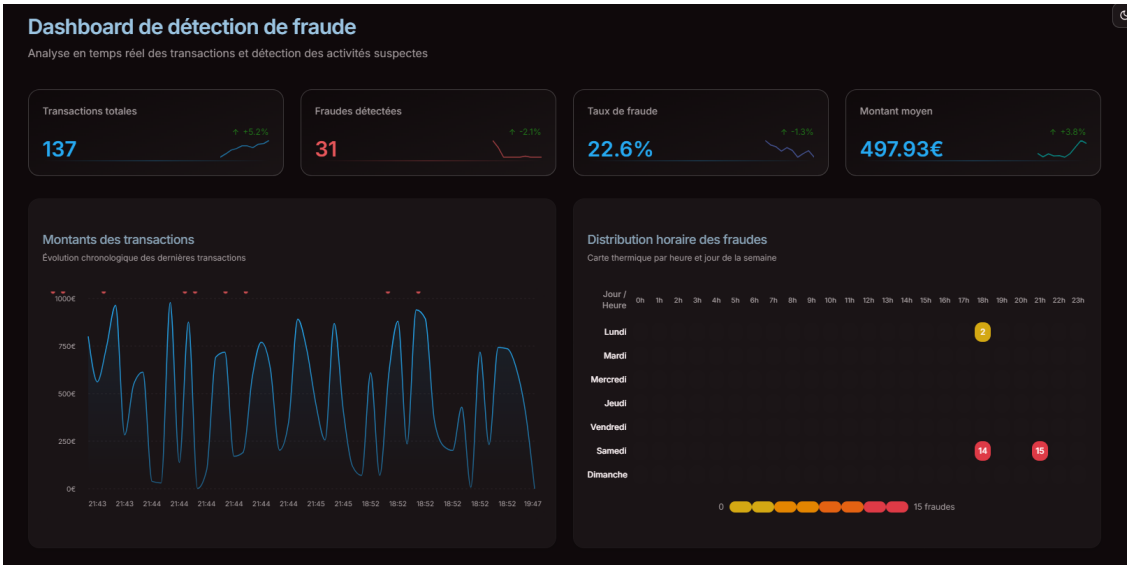


0.11.8 Impact opérationnel

- **Détection proactive** : repérage immédiat des tendances anormales.

- **Analyse multidimensionnelle** : vue temporelle, géographique, financière et par probabilité.
- **Prise de décision éclairée** : métriques et visuels intuitifs.
- **Optimisation des ressources** : allocation ciblée sur les zones et créneaux à risque.
- **Traçabilité et conformité** : historique complet des transactions et des actions d'analyse.





Note complémentaire

Pour aller plus loin, le même *dataset* a servi à produire des rapports interactifs sous **Power BI** et **Tableau**, offrant des vues macro (agrégations mensuelles, clusters clients, corrélations montants / catégories / localisations) qui complètent l'analyse temps réel du dashboard.

Fin