Projet Big Data et Machine Learning

Conception d'un pipeline Kappa pour la prédiction en temps réel

Réalisé par le Groupe 3 :

K.Inès Marie-Laure	**BADO**
Abdallah	**BAMOGO**
Neimata	**KABORE**
Oumar (2001)	**TRAORE**

Professeurs:
Dr Constantin DRABO et **Pr Sadouanouan MALO**

Master 2 - Sciences de données
Université Nazi Boni

07 /Juin /2025

Contents

List of Figures

1. Introduction

1.1. Objectif du projet

Dans un contexte où les volumes de données générées par les marchés financiers augmentent de manière exponentielle, les technologies de Big Data et de Machine Learning jouent un rôle essentiel pour extraire de la valeur en temps réel. Le Big Data permet de collecter, stocker et traiter des flux massifs et continus de données, tandis que le Machine Learning offre des méthodes avancées pour analyser ces données et réaliser des prédictions précises.
Ce projet, vise à concevoir un pipeline de traitement de données en temps réel basé sur l'architecture Kappa. L'objectif est de développer un système capable de collecter des données financières en continu, de les stocker dans une infrastructure scalable, et d'exploiter ces données pour prédire la valeur de clôture (close) d'un actif financier à partir du volume des transactions (volume), illustrant ainsi l'application concrète des approches Big Data et Machine Learning dans la finance.

1.2. Présentation du site Polygon.io

Polygon.io est une plateforme spécialisée dans la fourniture d'API financières en temps réel, offrant un accès fiable et complet à des données de marché variées pour les développeurs, entreprises et particuliers. Elle couvre plusieurs classes d'actifs, dont les actions américaines, les crypto-monnaies, les options, les contrats à terme et plus de 10 000 indices (comme le SP, Nasdaq, Dow Jones).
Polygon.io propose des données riches, telles que les cotations (tickers), historiques de prix, volumes de transactions, données agrégées (OHLCV : Open, High, Low, Close, Volume), ainsi que des données transactionnelles au niveau des ticks (transactions, cotations NBBO).
Pour faciliter l'accès à ces données, la plateforme utilise des API REST et des flux WebSocket, accompagnés de bibliothèques clientes, d'une interface S3 et d'une option d'interrogation SQL. Sa documentation complète et son support technique dédié simplifient l'intégration et l'utilisation. Parmi ses avantages, Polygon.io offre des données précises et de qualité, une large couverture des marchés financiers, une interface intuitive et des solutions économiques.

1.3. Problématique

Dans un environnement financier où les marchés évoluent rapidement et génèrent un volume important de données en temps réel, il devient essentiel de disposer d'outils performants permettant non seulement de capter ces flux continus, mais aussi d'en extraire des informations décisionnelles instantanément. L'enjeu est d'anticiper les variations du

marché en exploitant efficacement les données disponibles. La problématique à laquelle nous répondons dans ce projet est donc la suivante :

> Comment mettre en œuvre un pipeline de traitement en temps réel permettant de prédire la valeur de clôture (close) d'un actif financier à partir du volume de transactions (volume) ?

Pour y répondre, nous proposons une solution s'appuyant sur l'architecture Kappa, structurée autour des étapes suivantes :

- La collecte continue des données de marché via l'API de Polygon.io,
- Leur transmission à l'aide d'un système de messagerie en temps réel basé sur Apache Kafka,
- Leur indexation et stockage dans un moteur de recherche rapide et scalable (Elasticsearch),
- L'application d'un modèle de prédiction implémenté en Python, capable d'exploiter ces données de manière dynamique.

1.4. Justification de l'architecture Kappa

L'architecture Kappa simplifie le traitement des données massives en unifiant le traitement en temps réel et par lots au sein d'une unique couche de streaming. Contrairement à l'architecture Lambda, elle évite la redondance entre deux pipelines (batch et stream), ce qui allège la maintenance, réduit les coûts d'infrastructure, et accélère le développement. Elle est particulièrement adaptée aux cas d'usage nécessitant une faible latence, une forte scalabilité et une grande flexibilité, comme la détection de fraude, l'analyse comportementale en ligne, les applications IoT ou encore les systèmes de traitement d'événements utilisateurs.

Dans le cadre de ce projet, l'architecture Kappa répond aux exigences de traitement continu des données financières. Elle permet d'ingérer et de traiter des flux en temps réel tout en conservant la possibilité de rejouer les données pour simuler un traitement par lots. L'architecture mise en œuvre repose ainsi sur un Stream Layer basé sur Apache Kafka pour l'ingestion, un Serving Layer avec Elasticsearch pour le stockage et la visualisation, et un module Python pour la transformation des données et la prédiction de la valeur de clôture à partir du volume.

Ce choix technologique permet une réactivité accrue face aux variations des marchés financiers, et autorise des analyses et prédictions en temps quasi réel.

2. Architecture du pipeline

Le pipeline de traitement repose sur l'architecture Kappa et suit la chaîne suivante :

Polygon.io (API) → Kafka (Stream Layer) → Python Consumer → Elasticsearch (Serving Layer)

2.1. Composants techniques

- **Kafka** : configuration de broker, topics, et exécution via Docker.
- **Python Producteur** : Interroge l'API de Polygon.io et publie les données dans Kafka.
- **Python Consommateur** : Lit les messages depuis Kafka et les indexe dans Elasticsearch.
- **Elasticsearch** : stocke les données structurées et permet la visualisation.

3. Collecte des données

3.1. Récupération des données (Producteur Kafka)

- **Source :** API de Polygon.io (ticker utilisé : AAPL)
- **Période :** les 5 dernières années.
- **Appel API :**
  - **Endpoint utilisé :** API de Polygon.io (ticker utilisé : AAPL)
  - **Données récupérées** les 5 dernières années.
- **Envoi Kafka :**
  - Envoie chaque enregistrement sous forme de message JSON dans le topic finance-data
  - Sérialisation via json.dumps()

Recupération des données des 5 denières années depuis polygon.io

```
[54]:
# Configuration Polygon
API_KEY = "lLLTSxaI5k8TNsMpstP3WDhGL8pkwbss"
TICKER = "AAPL"
END_DATE = date.today() - timedelta(days=1)
START_DATE = END_DATE - timedelta(days= 5*365)

# URL de l'API Polygon
url = f"https://api.polygon.io/v2/aggs/ticker/{TICKER}/range/1/day/{START_DATE}/{END_DATE}"
params = {
    "adjusted": "true",
    "sort": "asc",
    "limit": 5000,
    "apiKey": API_KEY
}

# Initialiser le producteur Kafka
producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

# Appel API
response = requests.get(url, params=params)

if response.status_code == 200:
    data = response.json()
    results = data.get("results", [])

    if results:
        print(f"{len(results)} résultat(s) trouvé(s) pour {TICKER} le {START_DATE} :\n")

        for d in results:
            message = {
                "ticker": TICKER,
                "timestamp": d["t"],
                "volume": d["v"],
                "close": d["c"],
                "open": d["o"],
                "high": d["h"],
                "low": d["l"]
            }

            # Envoi dans Kafka
            producer.send("finance-data", value=message)
            print("Envoyé dans Kafka :", message)

        producer.flush()
    else:
        print(f"Aucun résultat trouvé pour {TICKER} à la date {START_DATE}.")
else:
    print(f"Erreur {response.status_code} : {response.text}")
```

Figure 1: Code d'appel API à Polygon.io pour le ticker AAPL

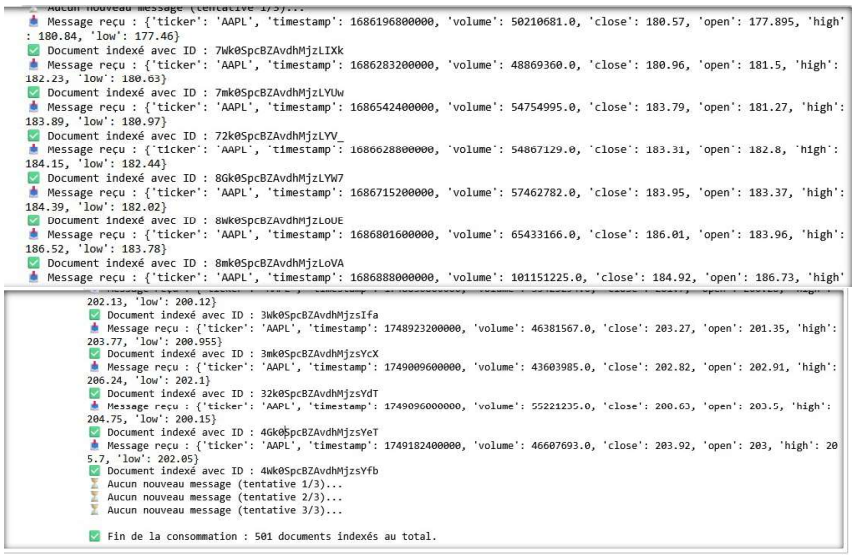

Figure 2: Visualisation d'un appel API à Polygon.io pour le ticker AAPL

3.2. Consommation des données (Consommateur Kafka)

- Consommateur Kafka sur le topic finance-data
- Affiche les messages reçus :
  - Ticker, horodatage, volume, prix d'ouverture/fermeture, haut/bas.
- Utilise un compteur d'inactivité :
  - Arrête le programme après 3 tours sans message.

```python
# Initialiser le consommateur Kafka
consumer = KafkaConsumer(
    'finance-data',
    bootstrap_servers='localhost:9092',
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    group_id='finance-display',
    value_deserializer=lambda x: json.loads(x.decode('utf-8'))
)

print(" Consommateur Kafka lancé. En attente de messages...\n")

no_message_count = 0
MAX_NO_MESSAGE_COUNT = 3  # Arrêter après 3 tours sans message

while True:
    raw_messages = consumer.poll(timeout_ms=2000)

    if not raw_messages:
        no_message_count += 1
        print(f"Aucun message reçu (tentative {no_message_count}/{MAX_NO_MESSAGE_COUNT})...")
        if no_message_count >= MAX_NO_MESSAGE_COUNT:
            print("\n Tous les messages ont été affichés. Fin du programme.")
            break
        continue

    for tp, messages in raw_messages.items():
        for message in messages:
            data = message.value
            print(" Message reçu :")
            print(f"  Ticker      : {data['ticker']}")
            print(f"  Timestamp   : {data['timestamp']}")
            print(f"  Volume      : {data['volume']}")
            print(f"  Open        : {data['open']}")
            print(f"  Close       : {data['close']}")
            print(f"  High / Low  : {data['high']} / {data['low']}")
            print("-" * 50)

    no_message_count = 0  # On a bien reçu des messages

consumer.close()
```

Figure 3: Code du consommateur Kafka pour la récupération des messages

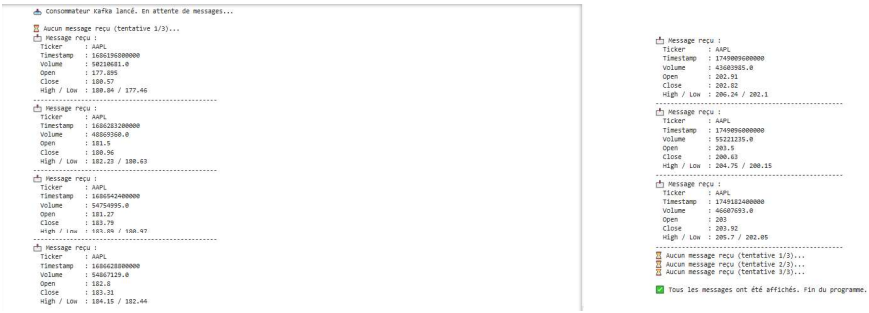

Figure 4: Affichage des messages consommés depuis le topic Kafka `finance-data`

3.3. Envoi vers Elasticsearch

- **Connexion à Elasticsearch** (localhost:9200).
- **Indexation des messages :**
  - Utilise le même topic finance-data.
  - Chaque message est indexé dans l'index polygon-index.
  - Chaque document est inséré et l'ID retourné est affiché.
- Arrêt automatique après 3 itérations sans message (compteur d'inactivité)

Envoi des données vers Elasticsearch

```
]: from elasticsearch import Elasticsearch

]: # Connexion à Elasticsearch
es = Elasticsearch("http://localhost:9200")

# Connexion au topic Kafka
consumer = KafkaConsumer(
    'finance-data',
    bootstrap_servers='localhost:9092',
    value_deserializer=lambda m: json.loads(m.decode('utf-8')),
    auto_offset_reset='earliest',
    enable_auto_commit=True,
    group_id='finance-group'
)

print("Lecture des messages Kafka et indexation dans Elasticsearch...\n")

# Compteur d'inactivité
no_message_count = 0
MAX_NO_MESSAGE_COUNT = 3  # Arrêt après 3 itérations sans messages
total_indexed = 0

while True:
    raw_messages = consumer.poll(timeout_ms=2000)

    if not raw_messages:
        no_message_count += 1
        print(f"Aucun nouveau message (tentative {no_message_count}/{MAX_NO_MESSAGE_COUNT})...")
        if no_message_count >= MAX_NO_MESSAGE_COUNT:
            print(f"\n Fin de la consommation : {total_indexed} documents indexés au total.")
            break
        continue

    for tp, messages in raw_messages.items():
        for message in messages:
            doc = message.value
            print(f"Message reçu : {doc}")

            # Indexation dans Elasticsearch
            res = es.index(index="polygon-index", document=doc)
            print(f"Document indexé avec ID : {res['_id']}")
            total_indexed += 1

    no_message_count = 0  # On a lu des messages, on remet à zéro

consumer.close()
```

Figure 5: Code du consommateur Kafka pour l'indexation des messages dans Elasticsearch

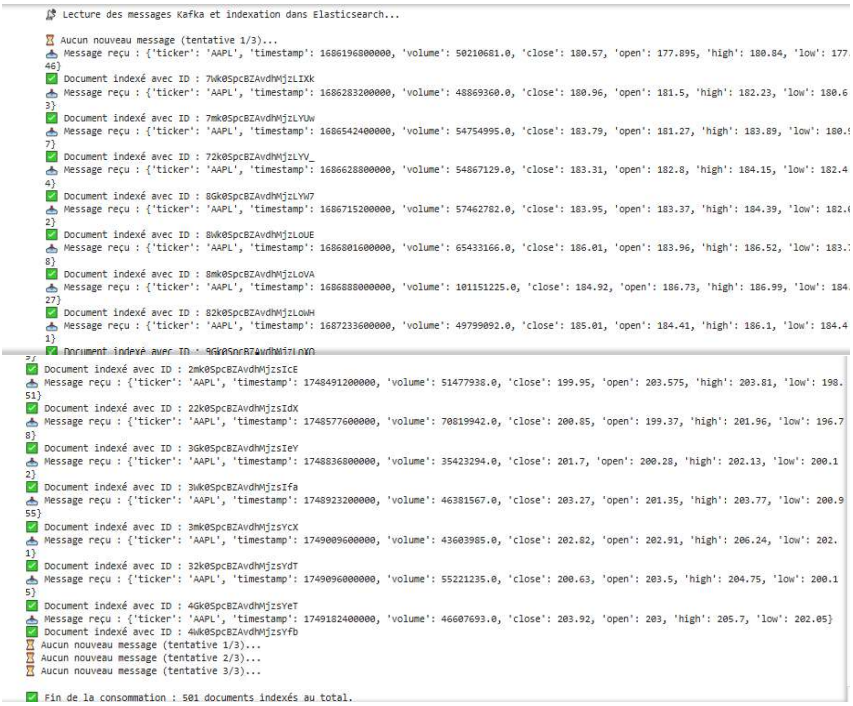

Figure 6: Affichage des réponses d'Elasticsearch après indexation des messages

4. Modélisation Prédictive

Pour répondre à notre problématique de prédiction de la valeur de clôture d'un actif financier (`close`) à partir du volume des transactions (`volume`), nous avons adopté une approche simple et explicable. Plusieurs étapes ont été suivies :

4.1. Méthodologie

- **Préparation des données :**Extraction des documents d'Elasticsearch, transformation en DataFrame pandas, puis séparation en ensembles d'entraînement et de test (80/20
- **Modèles testés :**
  - Régression linéaire
  - Forêt aléatoire (Random Forest)
  - Gradient Boosting
- **Métriques d'évaluation :**
  - MSE (Mean Squared Error)
  - R^2 (coefficient de détermination)

Création du modèle sur la base des données envoyées à Elasticsearch

```
from elasticsearch.helpers import scan
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

Importation des données depuis Elasticsearch

```
es = Elasticsearch("http://localhost:9200")

results = scan(es, index="polygon-index", query={"query": {"match_all": {}}})
data = pd.DataFrame([doc['_source'] for doc in results])
data.shape
```

```
(2533, 7)
```

```
print(data.head())
```

```
  ticker      timestamp      volume   close     open      high       low
0   AAPL  1734066000000  33081590.0  248.13  247.815  249.2902  246.2400
1   AAPL  1734325200000  51694753.0  251.04  247.990  251.3800  247.6500
2   AAPL  1734411600000  51356360.0  253.48  250.080  253.8300  249.7800
3   AAPL  1734498000000  56764701.0  248.05  252.160  254.2800  247.7400
4   AAPL  1734584400000  60559114.0  249.79  247.500  252.0000  247.0949
```

Préparation et séparation des des données

```
X = data[['volume']]        # Variable explicative
y = data['close']           # Variable cible

# 4. Séparer en ensemble 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)
```

Figure 7: Code d'importation et de préparation des données

4.2. Résultats et interprétations

4.2.1 Modèle de régression Linéaire

Figure 8: Code d'entrainement et d'évaluation du modèle de regression linéaire

Interprétation des résultats de la regression linéaire : Le score R^2 obtenu avec ce modèle est de seulement **0,033**, ce qui signifie qu'il n'explique que **3,3 %** de la variance de la variable cible. Autrement dit, ses prédictions sont presque équivalentes à une estimation constante basée sur la moyenne. Ce résultat indique que la régression linéaire ne parvient pas à capturer correctement la relation entre le volume et la valeur de clôture. Il est donc nécessaire d'envisager des améliorations telles que :

- l'ajout de variables explicatives plus pertinentes (comme `open`, `high`, `low`, etc.) ;
- un meilleur prétraitement des données ;

- ou l'utilisation de modèles non linéaires plus performants.

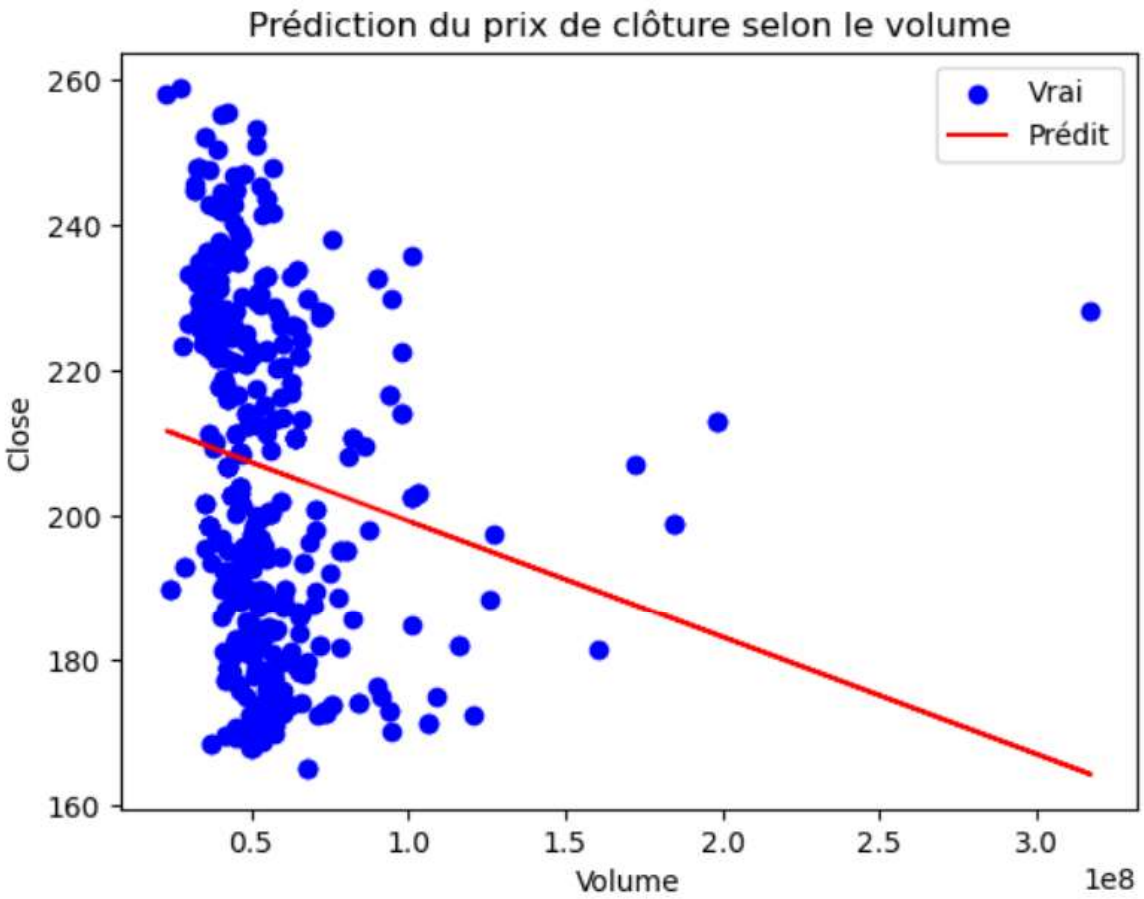

Figure 9: Visualisation de la prédiction du prix de clôture selon le volume par la regression linéaire

Le graphique ci-dessus confirme que votre modèle actuel est non-fonctionnel. Il ignore complètement la variable d'entrée (volume) et prédit systématiquement une valeur moyenne, indépendante des données fournies. Une refonte complète de l'approche de modélisation est nécessaire.

4.2.2 Modèle de régression de Forêt aléatoire

Random Forest Regressor

```
[62]: from sklearn.ensemble import RandomForestRegressor
      from sklearn.metrics import mean_squared_error, r2_score

      model = RandomForestRegressor(n_estimators=100, random_state=42)
      model.fit(X_train, y_train)
      y_pred = model.predict(X_test)

      print("R² :", r2_score(y_test, y_pred))
      print("MSE :", mean_squared_error(y_test, y_pred))

R² : 0.9823421404627342
MSE : 9.364382597302859
```

Figure 10: Implémentation de la régression par Forêt Aléatoire : entraînement et évaluation du modèle

Interprétation des résultats du Random Forest Regressor : Le modèle Random Forest a obtenu un score $R^2 = 0{,}9823$, indiquant qu'il explique 98,23 % de la variance des données. Cette performance très proche de la perfection théorique montre que le modèle capture efficacement la relation entre les variables explicatives et la variable cible.

En termes d'erreurs, le modèle affiche un **MSE** de 9,364, en nette amélioration par rapport au modèle précédent (512,77 → 9,36). La **RMSE** correspondante est d'environ **3,06**, ce qui signifie que l'erreur moyenne de prédiction est faible. Ces résultats confirment la solidité et la pertinence du modèle Random Forest pour ce cas d'usage.

4.2.3 Modèle de régression du Gradient Boosting

Gradient Boosting Regressor

```
]: from sklearn.ensemble import GradientBoostingRegressor

model = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1, random_state=42)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("R² :", r2_score(y_test, y_pred))
print("MSE :", mean_squared_error(y_test, y_pred))
```

```
R² : 0.40194451672465736
MSE : 317.16303711589757
```

Figure 11: Implémentation de la régression par Gradient Boosting : entraînement et évaluation du modèle

Interprétation des performances intermédiaires : Le modèle atteint un score $R^2 = 0{,}402$, expliquant 40,2 % de la variance des données. Cela marque une amélioration notable par rapport au premier modèle ($R^2 = 0{,}033$), bien que la performance reste nettement inférieure à celle du Random Forest ($R^2 = 0{,}982$).

L'erreur quadratique moyenne (**MSE**) s'élève à 317,16, ce qui correspond à une erreur moyenne (**RMSE**) d'environ 17,8 unités. Cette erreur est plus faible que celle du modèle initial (RMSE 22,65), mais demeure environ 35 fois plus élevée que celle du Random Forest ($MSE = 9{,}36$). Le modèle présente donc des performances modérées, insuffisantes pour une prédiction fiable à grande échelle.

Résumé des performances des modèles testés : Trois modèles de régression ont été évalués dans le cadre de la prédiction de la valeur de clôture d'un actif financier à partir du volume des transactions.

Le premier modèle, une régression linéaire, a montré des performances très faibles avec un score $R^2 = 0{,}033$, n'expliquant qu'environ 3,3 % de la variance. L'erreur quadratique moyenne (MSE) de 512,77 traduit une erreur moyenne de 22,65 unités, proche d'une prédiction naïve.

Le second modèle, un Gradient Boosting Regressor, améliore sensiblement les résultats avec un score $R^2 = 0{,}402$. Il explique 40,2 % de la variance, réduisant l'erreur moyenne à

17,8 unités (MSE = 317,16). Bien que meilleur que la régression linéaire, ce modèle reste bien en deçà du niveau attendu pour des prévisions fiables.

En revanche, le modèle de Forêt Aléatoire affiche des performances remarquables. Avec un score $R^2 = 0{,}982$, il explique 98,2 % de la variance des données. Son erreur moyenne est très faible (MSE = 9,36 ; RMSE 3,06), indiquant une excellente capacité à capturer la relation entre les variables prédictives et la variable cible. Ce modèle constitue donc l'approche la plus efficace parmi celles testées.

5. Difficultés rencontrées

- Problèmes techniques liés à l'installation de Kafka et la configuration du producteur/consommateur
- Limitations d'accès à l'API de Polygon.io (quota, erreurs HTTP)
- Délai dans le traitement des flux si surcharge
- Problèmes de compatibilité entre les versions des services (Kafka, Elasticsearch, Python, etc.)
- Latence élevée dans la transmission des messages.
- Limite de débit sur les topics Kafka
- Difficulté à assurer la tolérance aux pannes (résilience) en cas d'arrêt d'un composant

6. Conclusion et perspectives

- Compétences acquises : mise en place d'une architecture Kappa, traitement de flux, machine learning appliqué à la finance.
- Perspectives : ajout de nouvelles variables (Open, High, Low), entraînement de modèles plus complexes, déploiement dans un environnement cloud pour une mise en production.

Références

[1] Polygon.io. *Real-time and historical market data APIs*. Disponible sur : https://polygon.io

[2] *Moteur de recherche distribué*. Disponible sur : https://www.elastic.co/elasticsearch/.

[3] https://github.com/AbdallahBAMOGO/projetbigdatamachinelearninggroupe03/tree/main/docs