

# **Projet Big Data et Machine Learning**

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

**Réalisé par le groupe 3 :**

BADO Marie-Laure

BAMOGO Abdallah

KABORE Neimata

TRAORE Oumar

**Professeurs :**

**Dr Constantin DRABO**

**Pr Sadouanouan MALO**

**Master 2 - Sciences de données**

Université de Nazi Boni

Juin 2025

## Table des matières

I.	Introduction .....	4
1.	Objectif du projet .....	4
2.	Présentation du site Polygon.io .....	4
3.	Problématique .....	5
4.	Choix de l'architecture Kappa .....	5
II.	Architecture du pipeline .....	6
1.	Composants techniques .....	6
III.	Collecte des données .....	6
1.	Récupération des données (Producteur Kafka) .....	6
2.	Consommation des données (Consommateur Kafka) .....	8
3.	Envoi vers Elasticsearch .....	9
IV.	Modélisation Prédictive .....	11
1.	Méthodologie .....	11
2.	Résultats et interprétations .....	12
1.	Modèle de régression linéaire .....	12
2.	Modèle de régression par Forêt Aléatoire .....	14
3.	Modèle de régression par Gradient Boosting .....	14
V.	Difficultés rencontrées .....	15
VI.	Conclusion et perspectives .....	16
1.	Conclusion .....	16
2.	Perspectives d'évolution .....	17
	Références .....	18

## Table de illustrations

Figure 1 : Code d'appel API à Polygon.io pour le ticker AAPL .....	7
Figure 2 : Visualisation d'un appel API à Polygon.io pour le ticker AAPL.....	8
Figure 3 : Code du consommateur Kafka pour la récupération des messages .....	9
Figure 4 : Affichage des messages consommés depuis le topic Kafka finance-data .....	9
Figure 5 : Code du consommateur Kafka pour l'indexation des messages dans Elasticsearch.....	10
Figure 6 : Affichage des réponses d'Elasticsearch après indexation des messages .....	11
Figure 7 : Code d'importation et de préparation des données.....	12
Figure 8 : Code d'entraînement et d'évaluation du modèle de régression linéaire.....	12
Figure 9 : Visualisation de la prédiction du prix de clôture selon le volume par la régression linéaire .....	13
Figure 10 : Implémentation de la régression par Forêt Aléatoire : entraînement et évaluation du modèle	14
Figure 11: Implémentation de la régression par Gradient Boosting – entraînement et évaluation.....	14

# **I. Introduction**

## **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. un contexte où les volumes de données générées par les marchés financiers.

## **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 S&P, 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.

### 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.

***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.

### 4. Choix 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. L'architecture Kappa simplifie le traitement des données massives.

## **II. 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)

### **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.

## **III. Collecte des données**

### **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** : Cours boursiers des 5 dernières années.
- ✓ **Envoi Kafka** :
  - Chaque enregistrement est envoyé sous forme de message JSON dans le topic finance-data.
  - La sérialisation est effectuée à l'aide de la fonction json.dumps().

#### Recupération des données des 5 dernières années depuis polygon.io

```

•[54]: # Configuration Polygon
API_KEY = "1LLT5xaISk8TNSmpstP3wDhGL8pkwbss"
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

```
✓ 501 résultat(s) trouvé(s) pour AAPL le 2020-06-07 :

📡 Envoyé dans Kafka : {'ticker': 'AAPL', 'timestamp': 1686196800000, 'volume': 50210681.0, 'close': 180.57, 'open': 177.895, 'high': 180.84, 'low': 177.46}
📡 Envoyé dans Kafka : {'ticker': 'AAPL', 'timestamp': 1686283200000, 'volume': 48869360.0, 'close': 180.96, 'open': 181.5, 'high': 182.23, 'low': 180.63}
📡 Envoyé dans Kafka : {'ticker': 'AAPL', 'timestamp': 1686542400000, 'volume': 54754995.0, 'close': 183.79, 'open': 181.27, 'high': 183.89, 'low': 180.97}
📡 Envoyé dans Kafka : {'ticker': 'AAPL', 'timestamp': 1686628800000, 'volume': 54867129.0, 'close': 183.31, 'open': 182.8, 'high': 184.15, 'low': 182.44}
📡 Envoyé dans Kafka : {'ticker': 'AAPL', 'timestamp': 1686715200000, 'volume': 57462782.0, 'close': 183.95, 'open': 183.37, 'high': 184.39, 'low': 182.02}
📡 Envoyé dans Kafka : {'ticker': 'AAPL', 'timestamp': 1686801600000, 'volume': 65433166.0, 'close': 186.01, 'open': 183.96, 'high': 186.52, 'low': 183.78}
📡 Envoyé dans Kafka : {'ticker': 'AAPL', 'timestamp': 1686888000000, 'volume': 101151225.0, 'close': 184.92, 'open': 186.73, 'high': 186.99, 'low': 184.27}
📡 Envoyé dans Kafka : {'ticker': 'AAPL', 'timestamp': 1687233600000, 'volume': 49799092.0, 'close': 185.01, 'open': 184.41, 'high': 186.1, 'low': 184.41}
```

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

## 2. Consommation des données (Consommateur Kafka)

- ✓ **Consommateur Kafka** est connecté au topic finance-data
- ✓ **Affichage des messages reçus :**
  - Ticker
  - Horodatage
  - Volume
  - Prix d'ouverture et de fermeture
  - Valeur la plus haute et la plus basse
- ✓ **Utilisation d'un compteur d'inactivité :**
  - Le programme s'arrête automatiquement après **3 itérations sans réception de message**



```

: # 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

```

Consommateur Kafka lancé. En attente de messages...

Aucun message reçu (tentative 1/3)...
Message reçu :
Ticker      : AAPL
Timestamp   : 1686196800000
Volume      : 50210681.0
Open        : 177.895
Close       : 180.57
High / Low  : 180.84 / 177.46
-----
Message reçu :
Ticker      : AAPL
Timestamp   : 1686283200000
Volume      : 48869360.0
Open        : 181.5
Close       : 180.96
High / Low  : 182.23 / 180.63
-----
Message reçu :
Ticker      : AAPL
Timestamp   : 1686542400000
Volume      : 54754995.0
Open        : 181.27
Close       : 183.79
High / Low  : 183.69 / 180.97
-----
Message reçu :
Ticker      : AAPL
Timestamp   : 1686628800000
Volume      : 54867129.0
Open        : 182.8
Close       : 183.31
High / Low  : 184.15 / 182.44

```

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

### 3. Envoi vers Elasticsearch

- ✓ Connexion à Elasticsearch sur localhost:9200
- ✓ Indexation des messages :

- Utilise le même topic Kafka : finance-data,
- Chaque message est indexé dans l'index **polygon-index**,
- Chaque document est inséré individuellement, et l'**ID retourné** est affiché à l'écran.

✓ **Arrêt automatique** du programme après **3 itérations sans message** (grâce à un 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

```
Lecture des messages Kafka et indexation dans Elasticsearch...

Aucun nouveau message (tentative 1/3)...
Message reçu : {'ticker': 'AAPL', 'timestamp': 1686196800000, 'volume': 50210681.0, 'close': 180.57, 'open': 177.895, 'high': 180.84, 'low': 177.46}
Document indexé avec ID : 7mk0SpcBZAvdHmjzLIXk
Message reçu : {'ticker': 'AAPL', 'timestamp': 1686283200000, 'volume': 48869360.0, 'close': 180.96, 'open': 181.5, 'high': 182.23, 'low': 180.63}
Document indexé avec ID : 7mk0SpcBZAvdHmjzLYUw
Message reçu : {'ticker': 'AAPL', 'timestamp': 1686542400000, 'volume': 54754995.0, 'close': 183.79, 'open': 181.27, 'high': 183.89, 'low': 180.7}
Document indexé avec ID : 72k0SpcBZAvdHmjzLYV_
Message reçu : {'ticker': 'AAPL', 'timestamp': 1686628800000, 'volume': 54867129.0, 'close': 183.31, 'open': 182.8, 'high': 184.15, 'low': 182.44}
Document indexé avec ID : 8Gk0SpcBZAvdHmjzLYW7
Message reçu : {'ticker': 'AAPL', 'timestamp': 1686715200000, 'volume': 57462782.0, 'close': 183.95, 'open': 183.37, 'high': 184.39, 'low': 182.42}
Document indexé avec ID : 8Wk0SpcBZAvdHmjzLoUE
Message reçu : {'ticker': 'AAPL', 'timestamp': 1686801600000, 'volume': 65433166.0, 'close': 186.01, 'open': 183.96, 'high': 186.52, 'low': 183.18}
Document indexé avec ID : 8mk0SpcBZAvdHmjzLoVA
Message reçu : {'ticker': 'AAPL', 'timestamp': 1686888000000, 'volume': 101151225.0, 'close': 184.92, 'open': 186.73, 'high': 186.99, 'low': 184.27}
Document indexé avec ID : 82k0SpcBZAvdHmjzLoWH
Message reçu : {'ticker': 'AAPL', 'timestamp': 1687233600000, 'volume': 49799092.0, 'close': 185.01, 'open': 184.41, 'high': 186.1, 'low': 184.41}
Document indexé avec ID : 9Gk0SpcBZAvdHmjzLoYD
```

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

## IV. 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 :

### 1. Méthodologie

- ✓ **Préparation des données** : Extraction des documents depuis Elasticsearch, transformation en DataFrame avec Pandas, puis séparation en deux ensembles :
  - Ensemble d'entraînement : **80 %**
  - Ensemble de test : **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

```

j]: 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

j]: 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

j]: (2533, 7)

j]: 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 données

j]: 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

## 2. Résultats et interprétations

### 1. Modèle de régression linéaire

#### Création et entraînement du modèle

```

9]: model = LinearRegression()
model.fit(X_train, y_train)

# Prédiction
y_pred = model.predict(X_test)


```

#### Évaluation des modèles

```

0]: print("Évaluation du modèle :")
print("Score R² :", r2_score(y_test, y_pred))
print("Erreur quadratique moyenne :", mean_squared_error(y_test, y_pred))

```

 Évaluation du modèle :  
 Score R² : 0.033092139804148935  
 Erreur quadratique moyenne : 512.7742193273415

```

1]: plt.scatter(X_test, y_test, color='blue', label='Vrai')
plt.plot(X_test, y_pred, color='red', label='Prédit')
plt.xlabel('Volume')
plt.ylabel('Close')
plt.title('Prédiction du prix de clôture selon le volume')
plt.legend()
plt.show()

```

Figure 8 : Code d'entraînement et d'évaluation du modèle de régression linéaire

## Interprétation

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 quasiment équivalentes à une estimation constante basée sur la moyenne.

Ce résultat montre que la régression linéaire ne parvient pas à capturer correctement la relation entre le volume et la valeur de clôture. Des améliorations envisageables incluent :

- L'ajout de variables explicatives plus pertinentes (par exemple : open, high, low, etc.),
- Un meilleur prétraitement des données,
- L'utilisation de modèles non linéaires plus performants.

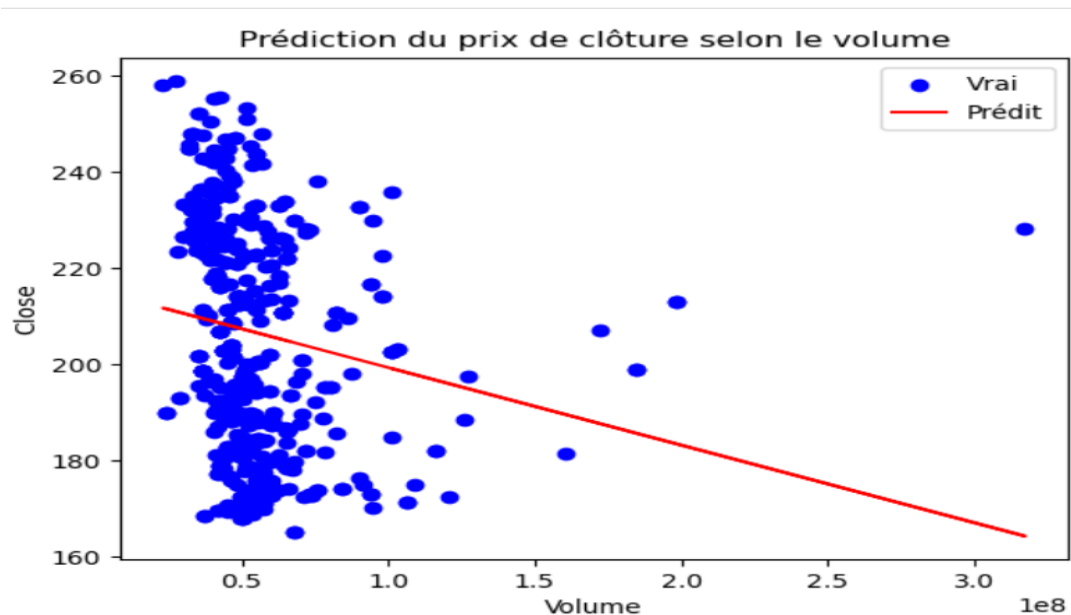


Figure 9 : Visualisation de la prédiction du prix de clôture selon le volume par la régression linéaire

Ce graphique confirme que le modèle est inefficace : il ignore la variable d'entrée et prédit une valeur moyenne constante, sans lien réel avec les données fournies.

## 2. Modèle de régression par Forêt Aléatoire

### Random Forest Regressor

```
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

Le modèle a obtenu un score  $R^2 = 0,9823$ , expliquant **98,23 %** de la variance des données. Cette performance remarquable indique une excellente capacité à modéliser la relation entre les variables.

- **MSE** : 9,364 (comparé à 512,77 pour le modèle linéaire)
- **RMSE** :  $\approx 3,06$

Ces résultats confirment la grande précision du modèle de la forêt aléatoire pour ce cas d'usage.

## 3. Modèle de régression par 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

### ✓ **Interprétation**

Le modèle de Gradient Boosting obtient un score  $R^2 = 0,402$ , expliquant **40,2 %** de la variance. Cela constitue une amélioration par rapport au modèle linéaire, mais reste inférieur à la Forêt Aléatoire.

- **MSE** : 317,16
- **RMSE** :  $\approx 17,8$

Ces résultats traduisent une amélioration modérée, mais encore insuffisante 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 **modèle de 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 **modèle Gradient Boosting** améliore sensiblement les résultats avec un score  $R^2 = 0,402$ . Il explique 40,2 % de la variance et réduit l'erreur moyenne à 17,8 unités (MSE = 317,16). Bien qu'il soit meilleur que la régression linéaire, ce modèle reste insuffisant pour des prévisions fiables.
- ✓ 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  $\approx 3,06$ ), ce qui indique 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.

## **V. Difficultés rencontrées**

- ✓ Plusieurs difficultés techniques et pratiques ont été rencontrées tout au long du projet :
- ✓ Problèmes techniques liés à l'installation de Kafka et à la configuration du producteur/consommateur.
- ✓ Limitations d'accès à l'API de Polygon.io (quotas, erreurs HTTP).
- ✓ Délai dans le traitement des flux en cas de 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 résilience du pipeline en cas d'arrêt d'un composant.

## VI. Conclusion et perspectives

### 1. Conclusion

Ce projet a permis de concevoir et de mettre en œuvre avec succès une architecture **Kappa** dédiée au traitement de données financières en temps réel. En s'appuyant sur des technologies modernes telles qu'**Apache Kafka**, **Elasticsearch** et des algorithmes de **Machine Learning**, nous avons développé un pipeline capable de **collecter**, **transformer**, **analyser** et **prédire** des flux de données haute fréquence de manière fiable et évolutive.

L'identification du **Random Forest** comme modèle le plus performant ( $R^2 = 0.982$ ,  $MSE = 9.36$ ) confirme la pertinence d'utiliser des algorithmes avancés pour modéliser des données aussi volatiles que les marchés financiers. Ce résultat met en évidence la **faiblesse des modèles linéaires** dans ce contexte et souligne l'importance de choisir des approches adaptées à la complexité des données.

Sur le plan pédagogique, ce projet a représenté un **terrain d'apprentissage concret**, combinant ingénierie de la donnée, modélisation prédictive et intégration de systèmes distribués. Il a également permis de surmonter plusieurs **défis techniques**, notamment en lien avec la configuration des outils, les limitations des API ou la gestion de la latence.



## 2. Perspectives d'évolution

- ✓ Enrichissement des données d'entrée avec de nouvelles variables
- ✓ Déploiement sur le cloud avec mise à l'échelle automatique
- ✓ Intégration de modèles séquentiels (type LSTM)
- ✓ Mise en place de prédictions dynamiques directement intégrées au flux Kafka

En définitive, ce projet illustre le **potentiel des architectures orientées streaming** pour les applications de **trading algorithmique** ou de **gestion du risque en temps réel**, en conjuguant **scalabilité, réactivité et intelligence prédictive**.

## Références

1. Polygon.io – *Real-time and historical market data APIs*  
Disponible sur : <https://polygon.io>
2. Elasticsearch – *Moteur de recherche distribué*  
Disponible sur : <https://www.elastic.co/elasticsearch>
3. GitHub du projet :  
<https://github.com/AbdallahBAMOGO/projetbigdatamachinelearninggroupe03/tree/main/docs>