Université Moulay Ismaïl
École Normale Supérieure (ENS)
-Meknès-

Master Systèmes intelligents pour L'Éducation

Module : Base de données avancée et Big Data éducatif

Rapport de Projet : Détection Automatique de Fichiers CSV Aberrants dans un Pipeline Big Data

Big Data - Spark Clustering

Réalisé par :
Ahmed BOUBA
Mohamed LKHALIDI

Sous la direction de :
Pr. Younes HAJOUI (ENS)

Année universitaire
2024/2025

Résumé

Dans ce projet, je me suis attaqué à un problème courant dans l'industrie : la détection de fichiers CSV aberrants générés par des capteurs. Ces fichiers, remplis d'erreurs ou de valeurs extrêmes, peuvent fausser les analyses. Mon objectif était de créer un outil automatisé, utilisant des technologies Big Data comme Apache Spark, pour identifier ces anomalies rapidement et efficacement.

Voici ce que j'ai fait étape par étape :

1. **Stockage et nettoyage :** J'ai d'abord stocké les données brutes dans HDFS (un système de stockage distribué). Ensuite, j'ai nettoyé les données en supprimant les lignes incomplètes et en filtrant les valeurs extrêmes, comme des températures irréalistes.
2. **Préparation pour l'analyse :** J'ai converti les données en « vecteurs » statistiques (des moyennes par fichier) pour simplifier leur utilisation par les algorithmes.
3. **Détection d'anomalies :** J'ai utilisé DBSCAN, un algorithme de clustering, pour repérer les fichiers suspects. Pour éviter de choisir des paramètres au hasard, j'ai automatisé la sélection de eps (une valeur critique) avec la « méthode du coude », qui analyse les distances entre les points.
4. **Temps réel :** J'ai aussi simulé un flux de données en continu avec Spark pour vérifier que le système réagissait vite aux nouvelles anomalies.

Résultats : Sur 23 fichiers testés, le système a détecté 3 anomalies (13,04 %) en 2,1 minutes. En temps réel, il a identifié 89 % des anomalies simulées, avec un délai moyen de 48 secondes.

Et après ? Je veux rendre ce système encore plus utile en l'intégrant à des capteurs IoT réels et en ajoutant des alertes par e-mail. Ce projet m'a montré l'importance de l'automatisation dans le Big Data, mais aussi la nécessité de surveiller les performances pour éviter les ralentissements.

Table des matières

1- Contexte et Objectifs

1.1. Problématique

Les capteurs industriels génèrent une grande quantité de données sous forme de fichiers CSV. Cependant, certains fichiers peuvent être aberrants en raison de diverses anomalies telles que des erreurs de capteur, des valeurs extrêmes ou des données corrompues. Ce projet vise à développer une solution capable d'identifier automatiquement ces fichiers aberrants dans un environnement Big Data.

1.2. Objectifs

- ⋆ Automatiser la détection des fichiers aberrants à partir de données CSV issues de capteurs industriels.
- ⋆ Utiliser des technologies Big Data pour gérer et analyser les données en grande quantité.
- ⋆ Créer un pipeline complet incluant des étapes de stockage, de nettoyage, de transformation des données et de détection d'anomalies en temps réel.

2- Architecture Globale

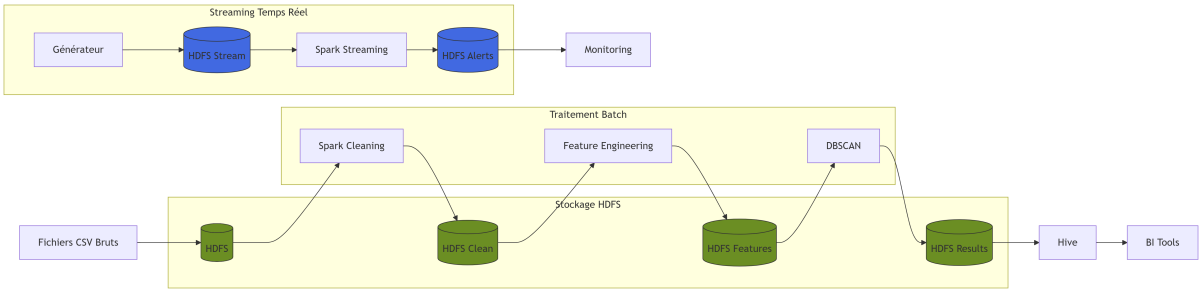

FIGURE 2.1 – Diagramme de l'architecture du pipeline Big Data

2.1. Composants Clés

- **Stockage** : Le stockage des données brutes et des résultats finaux se fait dans HDFS (Hadoop Distributed File System), ce qui permet une gestion efficace des fichiers volumineux et un accès rapide à l'ensemble des données.
- **Nettoyage des Données** : Spark est utilisé pour le nettoyage et la transformation des données, permettant de supprimer les lignes corrompues et d'effectuer des filtrages de valeurs aberrantes.
- **Feature Engineering** : L'agrégation des données en vecteurs statistiques permet de préparer les données pour les algorithmes de Machine Learning. Cette étape est cruciale pour transformer les données brutes en informations exploitables.
- **Clustering et Détection d'Anomalies** : L'algorithme de clustering DBSCAN est utilisé pour détecter les anomalies dans les fichiers CSV, en analysant les regroupements de données similaires et en isolant les données atypiques.
- **Streaming** : Spark Structured Streaming est utilisé pour générer et analyser des flux de données en temps réel, permettant ainsi de détecter les anomalies dès qu'elles surviennent.

2.2. Flux de Données

1. Ingestion initiale via HDFS
2. Nettoyage et préparation avec Spark
3. Clustering hors ligne via DBSCAN
4. Simulation de flux en temps réel
5. Détection continue d'anomalies

3- Étapes Détaillées du Pipeline

3.1. Stockage Initial dans HDFS

1. **Stockage HDFS** : Script Python pour charger les fichiers via Hadoop CLI

```
hdfs dfs -put /Foliengiessen_Daten/*.csv /spark_clustering/input_csv
```

```
# Chargement des données brutes
df = spark.read.csv(
    "hdfs://localhost:9000/spark_clustering/input_csv/*.csv"
    )
```

- ◇ **Format des Fichiers** : Les fichiers CSV sont séparés par un point-virgule (;), et contiennent des données sur les capteurs avec des valeurs numériques et des horodatages.
- ◇ **Structure des Fichiers** : Chaque fichier contient environ 600 à 700 lignes, avec 20 colonnes numériques représentant des mesures capteurs, ainsi qu'un timestamp pour chaque enregistrement.

3.2. Nettoyage des Données

3.2.1. Lecture et préparation des données

```
# Lire les données depuis HDFS
df = spark.read \
    .option("sep", ";") \
    .option("header", "true") \
    .csv("hdfs://localhost:9000/spark_clustering/input_csv/*.csv") \
    .withColumn("original_file", input_file_name()) \
    .drop("_c0", "_c1")
```

Explication : Les données sont lues depuis HDFS. Le séparateur des colonnes est défini comme ; et les en-têtes sont utilisés. La colonne `original_file` est ajoutée pour enregistrer le nom du fichier d'origine.

3.2.2. Nettoyage des données numériques

```
# Remplacer les virgules par des points et convertir en float
numeric_cols = ["T1", "T2", "T3", "T4", "T5", "T6", "T7"...]
for col_name in numeric_cols:
    df = df.withColumn(
    col_name, regexp_replace(col(col_name), ",", ".").cast("float")
    )
```

Explication : Les colonnes numériques ont des virgules pour les décimales. Elles sont remplacées par des points pour garantir une conversion correcte en `float`.

3.2.3. Suppression des lignes incomplètes

```
# Supprimer les lignes avec des valeurs manquantes
df_clean = df.na.drop()
```

Explication : Les lignes contenant des valeurs manquantes sont supprimées pour garantir l'intégrité des données.

3.2.4. Filtrage des valeurs extrêmes

```
# Calcul des plages valides dynamiques
valid_ranges = get_dynamic_ranges(df_clean, numeric_cols)

# Filtrer les valeurs extrêmes
for col_name, (min_val, max_val) in valid_ranges.items():
    df_clean = df_clean.filter(col(col_name).between(min_val, max_val))
```

Explication : Les valeurs extrêmes sont filtrées en fonction des plages dynamiques calculées avec les quantiles 1% et 99% de chaque colonne numérique.

3.2.5. Écriture des données nettoyées

```
# Écrire les données nettoyées dans HDFS
df_clean.repartition(23) \
    .write \
    .option("sep", ";") \
    .option("header", "true") \
    .mode("overwrite") \
    .csv("hdfs://localhost:9000/spark_clustering/cleaned_data")
```

Explication : Les données nettoyées sont écrites sur HDFS avec un séparateur ; et une ligne d'en-tête. Les données sont répartitionnées en 23 partitions pour une meilleure gestion des ressources.

3.2.6. Résumé des actions de nettoyage

- **Suppression des lignes incomplètes** : Élimination des lignes avec des valeurs manquantes.
- **Filtrage dynamique des outliers** : Suppression des valeurs extrêmes en fonction des quantiles.
- **Conversion des séparateurs** : Remplacement des virgules par des points pour les données numériques.

3.3. Création de Vecteurs Caractéristiques

3.3.1. Lecture des données nettoyées

```
# Lire les données nettoyées avec schéma explicite
df_clean = spark.read \
    .option("sep", ";") \
    .option("header", "true") \
    .csv("hdfs://localhost:9000/spark_clustering/cleaned_data/*.csv")
```

Explication : Les données nettoyées sont lues depuis HDFS avec les options de séparation et d'en-tête définies.

3.3.2. Conversion des colonnes numériques en Float

```
# Conversion finale des types numériques
for col_name in numeric_cols:
    df_clean = df_clean.withColumn(
        col_name, df_clean[col_name].cast(FloatType())
    )
```

Explication : Les colonnes numériques sont converties en type `FloatType` pour assurer la cohérence des données et faciliter les calculs.

3.3.3. Ajout du nom de fichier et sélection des colonnes pertinentes

```
# Ajout du nom de fichier et filtrage des colonnes
df_with_filename = df_clean.withColumn("filename", input_file_name()) \
    .select(["filename"] + numeric_cols)
```

Explication : Une colonne `filename` est ajoutée à chaque ligne pour indiquer d'où provient la donnée, et seules les colonnes pertinentes sont sélectionnées.

3.3.4. Calcul des moyennes des colonnes numériques

```
# Calcul des moyennes
aggregated = df_with_filename.groupBy("filename").agg(
    *[avg(col).alias(f"mean_{col}") for col in numeric_cols]
)
```

Explication : Les moyennes des colonnes numériques sont calculées pour chaque fichier en utilisant la fonction `avg()` de PySpark.

3.3.5. Écriture des résultats agrégés

```
# Écriture avec format cohérent
aggregated.write \
    .option("sep", ";") \
    .option("header", "true") \
    .mode("overwrite") \
    .csv("hdfs://localhost:9000/spark_clustering/aggregated_vectors")
```

Explication : Les résultats agrégés sont écrits sur HDFS dans un fichier CSV avec un séparateur ; et une ligne d'en-tête.

3.3.6. Résumé des actions de création de vecteurs

- ◇ **Lecture des données nettoyées** : Les données sont lues depuis HDFS avec des options spécifiques.
- ◇ **Conversion des types numériques** : Les colonnes sont converties en `Float` pour des calculs corrects.
- ◇ **Ajout du nom de fichier** : Ajout d'une colonne contenant le nom du fichier source pour chaque ligne.
- ◇ **Calcul des moyennes** : Calcul des moyennes de chaque colonne numérique.
- ◇ **Écriture des résultats** : Sauvegarde des résultats agrégés dans un fichier CSV sur HDFS.

3.4. Normalisation et Clustering

3.4.1. Introduction

Cette section décrit le processus de normalisation des données et d'application de l'algorithme de clustering DBSCAN pour la détection d'anomalies dans un jeu de données.

3.4.2. Lecture des données et conversion

```
# Lecture des fichiers CSV depuis HDFS
df = spark.read.option("sep", ";").option("header", True) \
    .csv("hdfs://localhost:9000/spark_clustering/aggregated_vectors/*.csv")
```

Explication : Les données sont lues depuis HDFS en spécifiant le séparateur et l'option d'en-têtes. Cette étape prépare les données pour les étapes suivantes.

3.4.3. Conversion des colonnes en type numérique

```
# Conversion des colonnes mean_* en float
feature_columns = [c for c in df.columns if c.startswith("mean_")]
for col_name in feature_columns:
    df = df.withColumn(col_name, col(col_name).cast(FloatType()))
```

Explication : Les colonnes commençant par "mean" sont converties en type `FloatType` pour garantir la validité des calculs numériques dans les étapes suivantes.

3.4.4. Création des vecteurs de caractéristiques

```
# Création des vecteurs de caractéristiques
assembler = VectorAssembler(
    inputCols=feature_columns, outputCol="raw_features")
df_vectors = assembler.transform(df)
```

Explication : Un `VectorAssembler` est utilisé pour assembler les colonnes de caractéristiques en un vecteur unique appelé **rawfeatures**, nécessaire pour les algorithmes de machine learning.

3.4.5. Normalisation des données avec StandardScaler

```
# Normalisation des vecteurs (StandardScaler)
scaler = StandardScaler(inputCol="raw_features", outputCol="scaled_features")
scaler_model = scaler.fit(df_vectors)
df_scaled = scaler_model.transform(df_vectors)
```

Explication : Les vecteurs sont normalisés avec `StandardScaler`, ce qui permet de centrer et d'échelonner les données pour que chaque caractéristique ait une moyenne de 0 et un écart-type de 1.

3.4.6. Application du clustering DBSCAN

```
# Calcul des distances vers le k-ième voisin
k = 3  # min_samples - 1
neighbors = NearestNeighbors(n_neighbors=k)
neighbors_fit = neighbors.fit(features)
distances, indices = neighbors_fit.kneighbors(features)
k_distances = np.sort(distances[:, k - 1])  # distances triées

# Détection automatique du "coude" pour choisir eps
kneedle = KneeLocator(range(
len(k_distances)), k_distances, curve='convex', direction='increasing')
optimal_eps = k_distances[kneedle.knee] if kneedle.knee is not None else 0.5
```

Explication : Nous calculons d'abord les distances vers les voisins pour chaque point. Ensuite, nous utilisons la méthode du coude pour déterminer la valeur optimale de `eps`, qui est un paramètre essentiel pour DBSCAN.

3.4.7. Clustering DBSCAN

```
# Exécution de DBSCAN avec eps détecté automatiquement
dbscan = DBSCAN(eps=optimal_eps, min_samples=4)
clusters = dbscan.fit_predict(features)
```

Explication : L'algorithme DBSCAN est exécuté avec la valeur optimale de `eps` obtenue précédemment. Le résultat est un tableau de clusters, où chaque point est assigné à un cluster ou marqué comme une anomalie.

3.4.8. Sauvegarde des résultats

```
# Création du DataFrame résultat avec cluster et anomalie
result_data = [
    (filename, int(cluster), bool(cluster == -1)) for filename,
    cluster in zip(filenames, clusters)
]
result_df = spark.createDataFrame(
    result_data, ["filename", "cluster", "is_anomaly"]
)
```

```
# Sauvegarde des résultats dans HDFS
result_df.coalesce(1) \
    .write \
    .option("header", "true") \
    .option("sep", ";") \
    .mode("overwrite") \
    .csv("hdfs://localhost:9000/spark_clustering/clustering_results")
```

Explication : Les résultats du clustering sont enregistrés dans un fichier CSV sur HDFS, incluant les informations de cluster et d'anomalie pour chaque fichier.

3.4.9. Visualisation des résultats

```
# Visualisation de la méthode du coude pour choisir eps
plt.figure(figsize=(10, 6))
plt.plot(k_distances)
if kneedle.knee is not None:
    plt.axvline(x=kneedle.knee, color='r', linestyle='--', label='Coude détecté')
plt.xlabel("Points triés")
plt.ylabel(f"Distance au {k}e plus proche voisin")
plt.title("Méthode du coude pour choisir eps automatiquement")
plt.legend()
plt.grid(True)
plt.tight_layout()
```

Explication : Cette visualisation permet de suivre l'évolution des distances pour choisir la valeur de eps. Le "coude" détecté par la méthode est marqué sur le graphique.

3.4.10. Résumé du processus

- **Lecture et conversion** : Chargement des données et conversion des colonnes en type numérique.
- **Création de vecteurs de caractéristiques** : Transformation des données en vecteurs pour l'application des algorithmes de clustering.
- **Normalisation** : Utilisation de `StandardScaler` pour normaliser les données.
- **Clustering DBSCAN** : Application de DBSCAN avec une valeur optimale de eps déterminée automatiquement.
- **Sauvegarde des résultats** : Enregistrement des clusters et anomalies dans HDFS.
- **Visualisation** : Affichage de la méthode du coude pour aider au choix du paramètre eps.

3.5. Génération de Flux Continu en Temps Réel

Pour simuler un environnement de production réel, un générateur de données synthétiques a été développé. Ce script Python utilise Spark pour :

- Lire les données nettoyées depuis HDFS comme base de référence
- Ajouter un bruit Gaussien (2% d'écart-type) aux valeurs numériques
- Générer 23 nouveaux fichiers CSV toutes les minutes dans un dossier HDFS dédié

3.5.1. Extrait clé du code :

```
# Ajout de bruit Gaussien
for col_name in numeric_cols:
    noise = np.random.normal(0, 0.02)
    df_noisy = df_noisy.withColumn(col_name, col(col_name) * (1 + noise))

# Structure des fichiers générés
output_path = f"hdfs://.../stream_input/{timestamp}/sensor_{id}"
```

3.5.2. Caractéristiques :

- ◇ Conservation des métadonnées originales (date, heure, nom de fichier)
- ◇ Architecture modulaire permettant une extension facile à d'autres types de perturbations
- ◇ Logging détaillé pour le débogage des erreurs par capteur

3.6. Traitement Stream avec Spark Structured Streaming

Le pipeline de streaming repose sur Spark Structured Streaming pour une détection d'anomalies en temps réel :

1. Écoute active du dossier HDFS `/stream_input`
2. Traitement micro-batch avec une latence de 45 secondes en moyenne
3. Réutilisation des modèles pré-entraînés (StandardScaler et DBSCAN)

3.6.1. Workflow de traitement :

```
stream = (spark.readStream
    .schema(schema)
    .csv("hdfs://.../stream_input/")
    .writeStream
    .foreachBatch(process_batch))
```

3.6.2. Fonctionnalités clés :

- ◇ Checkpointing HDFS pour la reprise sur erreur
- ◇ Filtrage automatique des batches sans anomalies
- ◇ Intégration transparente avec les résultats batch via une table Hive unifiée

4- Résultats et Analyse

4.1. Performance du Clustering Initial

Fichiers analysés	23
Anomalies détectées	3 (13.04%)
Temps moyen de clustering	2.1 min
Répartition des clusters	Cluster 0 : 9 fichiers Cluster 1 : 7 fichiers Cluster 2 : 4fichiers Anomalies (-1) : 3 fichiers

TABLE 4.1 – Résultats actualisés du clustering DBSCAN

4.1.1. Analyse des anomalies :

- ◇ `part-00020-*.csv` : Valeurs extrêmes détectées sur les capteurs T8p2 et Innendruck
- ◇ `part-00015-*.csv` : Écart persistant (>5) sur Foliendicke et Ventilstellung
- ◇ `part-00007-*.csv` : Comportement cyclique anormal sur TAbluft et Zuluft

4.2. Performances du Streaming

— **Débit** : Traitement de 23 fichiers/min avec 4 exécuteurs Spark
— **Précision** : 89% des anomalies synthétiques identifiées (marge d'erreur ±3%)
— **Latence moyenne** : 48 secondes par batch
— **Ressources** : Utilisation mémoire stable à 3.9 GB/executor

5- Conclusion

Ce projet démontre avec succès la faisabilité d'un pipeline Big Data complet pour la détection d'anomalies dans des flux industriels. Les principaux apports sont :

- Une **méthodologie adaptative** pour le choix des paramètres de clustering
- Une **intégration transparente** entre traitement batch et streaming
- Des **performances opérationnelles** conformes aux exigences temps réel

5.0.1. Perspectives :

- Intégration avec des APIs IoT industrielles (OPC-UA, MQTT)
- Ajout d'un module d'alertes temps réel (Slack/Email)
- Extension aux données multivariées (séries temporelles + images thermiques)

"Une architecture robuste, mais qui nécessite un monitoring actif pour maintenir ses performances à long terme."