Big Data Analytics

Résumé - Séances 4 et 5

Spark ML et Spark Streaming

Pr EL GHALI Btihal

18 janvier 2026

1 Introduction - IA et Big Data

1.1 Principe général

Enjeux de la combinaison IA + Big Data :

- Les techniques de Machine Learning extraient des connaissances à forte valeur ajoutée
- Les données massives présentent des spécificités de traitement et stockage (les V du Big Data)
- Les itérations des algorithmes ML aggravent la situation :
  - Le volume se multiplie
  - La vitesse se dégrade
  - La qualité des données peut se détériorer

1.2 Défis spécifiques

1.2.1 Contraintes techniques

Problème concret :

- Dataset : 500 GB
- Cluster : 200 GB RAM total
- **En ETL** : Pas de problème (traitement chunk par chunk)
- **En ML** : Algorithme de descente de gradient doit voir TOUT le dataset à chaque itération
  - 50 itérations = 25 TB de lecture effective

1.2.2 Contraintes industrielles

1. **Volume massif**
   - Datasets : Giga, Téra, voire Péta-octets
   - Ressources mémoire et CPU limitées
   - Risques : Out of Memory, temps d'attente interminables
2. **Infrastructure critique**
   - Avant les algorithmes IA, s'assurer que l'infrastructure peut tenir la charge
   - Importance du paramétrage correct
3. **Spark sensible au paramétrage**
   - Ne traite pas automatiquement les problèmes de performance
   - Paramètres clés : nombre de partitions, mémoire executors, parallélisme, caching, shuffle, broadcast
   - Mauvaise configuration = échec du job
   - **Tuning Spark = garantir l'efficacité des algorithmes IA**

1.3 Défis à traiter

Pour faire de l'IA sur données massives :

- Contraintes techniques : calculs et mémoire
- Qualité et préparation des données à l'échelle
- Architectures MLOps à adapter

2 Étapes d'un projet Data Science

2.1 Phase 1 : Compréhension du problème

— Définition des besoins métiers

— Détermination des objectifs et utilité de la connaissance

— Évaluation de la situation

2.2 Phase 2 : Approche analytique

— Définition de l'approche de résolution

— Production d'un plan de projet

— Approche statistique

— Sélection des techniques de Machine Learning

2.3 Phase 3 : Compréhension des données

— Collecte des données : Data Warehouse, Data Mart, BD opérationnelle, fichiers

— Vérification de la qualité des données

2.4 Phase 4 : Préparation des données

— Nettoyage des données

— Traitement des données manquantes

— Sélection de données

— Réduction de dimension

2.5 Phase 5 : Analyse des données

— Description et exploration

— Visualisation des données

— Découverte des tendances et patterns

2.6 Phase 6 : Modélisation

— Sélection des techniques de modélisation

— Création de modèle

2.7 Phase 7 : Évaluation

— Visualisation et interprétation des modèles

— Analyse de la connaissance

— Vérification de validité

— Réitération si nécessaire

2.8 Phase 8 : Déploiement

— Gestion de la connaissance découverte
— Mise à disposition des décideurs
— Déploiement en production

3 Spark MLlib

3.1 Définition

— Bibliothèque d'apprentissage automatique de Spark
— Objectif : Rendre le ML pratique, évolutif et facile
— Basé sur les DataFrames

3.2 Outils fournis

1. **Algorithmes ML**
   — Classification
   — Régression
   — Clustering (segmentation)
   — Filtrage collaboratif
2. **Features (Caractéristiques)**
   — Extraction de caractéristiques
   — Transformation
   — Réduction de dimensionnalité
   — Sélection
3. **Pipelines**
   — Construction de modèles ML
   — Évaluation
   — Réglage (tuning)
4. **Persistance**
   — Enregistrement et chargement
   — Algorithmes, modèles et pipelines
5. **Utilitaires**
   — Algèbre linéaire
   — Statistiques
   — Traitement de données

4 Pipelines ML

4.1 Principe

— Standardise les APIs pour faciliter la combinaison d'algorithmes
— Ensemble uniforme d'APIs de haut niveau sur DataFrames
— Enchaîne plusieurs Transformateurs et Estimateurs
— Spécifie un flux de travail ML

4.2 Composants d'un Pipeline

4.2.1 1. Transformateur (Transformer)

Définition :

— Algorithme transformant un DataFrame en un autre DataFrame

— Inclut les transformateurs de caractéristiques et les modèles

— Implémente la méthode `transform()`

Exemple :

— Un modèle ML transforme un DataFrame avec features en DataFrame avec prédictions

`DataFrame1 → Transformer → DataFrame2`

4.2.2 2. Estimateur (Estimator)

Définition :

— Algorithme pouvant être ajusté sur un DataFrame pour produire un Transformer

— Un algorithme d'apprentissage est un Estimator

— S'entraîne sur DataFrame et produit un modèle

— Implémente la méthode `fit()`

`DataFrame1 → Estimator → Transformer`

4.2.3 3. Évaluateur (Evaluator)

Définition :

— Évalue la performance du modèle selon certaines métriques

— Métriques : RMSE, ROC, Accuracy, etc.

— Permet le réglage des paramètres (model tuning)

— Implémente la méthode `evaluate()`

Exemples :

— BinaryClassificationEvaluator

— CrossValidator

— RegressionEvaluator

4.3 Structure complète d'un Pipeline

Un pipeline est un ensemble de plusieurs étapes qui peut persister :

1. Transformations de features
2. Estimateurs (algorithmes d'apprentissage)
3. Transformateurs (modèles entraînés)
4. Évaluateurs (métriques)

4.4 Exemple de Pipeline

Pipeline de Logistic Regression :

```
// Etape 1: Tokenizer (Transformer)
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")

// Etape 2: HashingTF (Transformer)
val hashingTF = new HashingTF()
  .setInputCol("words")
  .setOutputCol("features")

// Etape 3: Logistic Regression (Estimator)
val lr = new LogisticRegression()
  .setMaxIter(10)

// Creation du Pipeline
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF, lr))

// Entrainement
val model = pipeline.fit(trainingData)

// Prediction
val predictions = model.transform(testData)
```

5 Model Tuning (Réglage des paramètres)

5.1 Principe

— Spark MLlib automatise le réglage des paramètres

— Construit le meilleur modèle possible avec les données disponibles

— Utilise des techniques de validation croisée

5.2 Validation croisée (K-Fold Cross-Validation)

5.2.1 Principe du K-Fold

— Divise les données en K partitions (folds)

— Pour chaque fold :

  — Utilise K-1 folds pour l'entraînement

  — Utilise 1 fold pour la validation

— Moyenne les résultats sur les K itérations

— Sélectionne les meilleurs paramètres

5.2.2 Exemple avec K=3

Itération 1 :

— Training : Fold 1 + Fold 2

— Validation : Fold 3

Itération 2 :

— Training : Fold 1 + Fold 3

— Validation : Fold 2

Itération 3 :

— Training : Fold 2 + Fold 3

— Validation : Fold 1

5.3 Implémentation avec CrossValidator

```
import org.apache.spark.ml.tuning.{ParamGridBuilder, CrossValidator}
import org.apache.spark.ml.evaluation.RegressionEvaluator

// Grille de parametres a tester
val paramGrid = new ParamGridBuilder()
  .addGrid(lr.regParam, Array(0.1, 0.01))
  .addGrid(lr.elasticNetParam, Array(0.0, 0.5, 1.0))
  .build()

// Configuration du CrossValidator
val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEvaluator(new RegressionEvaluator())
  .setEstimatorParamMaps(paramGrid)
  .setNumFolds(3)  // K = 3

// Entrainement avec validation croisee
val cvModel = cv.fit(trainingData)

// Meilleur modele
val bestModel = cvModel.bestModel
```

5.4 ParamGridBuilder

— Construit une grille de paramètres à tester

— Teste toutes les combinaisons possibles

— Sélectionne automatiquement la meilleure combinaison

Exemple :

— regParam : [0.1, 0.01] → 2 valeurs

— elasticNetParam : [0.0, 0.5, 1.0] → 3 valeurs

— Total : $2 \times 3 = 6$ combinaisons à tester

6 Spark Streaming

6.1 Définition

Spark Streaming :

— Extension de l'API Spark pour traitement de flux de données

— **Évolutif** : Peut facilement évoluer vers des centaines de nœuds

— **Haut débit et rapide** : Atteint une faible latence

— **Tolérant aux pannes** : Continue le streaming après défaillances

6.2 Sources de données

Les données peuvent provenir de :

— Kafka
— Flume
— Kinesis
— Sockets TCP
— Systèmes de fichiers

6.3 Traitement des données

Opérations disponibles :

— Fonctions de haut niveau : map, reduce, join, window
— Algorithmes complexes
— Algorithmes de Machine Learning
— Traitement de graphes

Destinations :

— Systèmes de fichiers (HDFS, S3)
— Bases de données
— Tableaux de bord dynamiques

6.4 Fonctionnement interne

1. Spark Streaming reçoit des flux de données en direct
2. Divise les données en **lots (batches)**
3. Chaque lot est traité par le moteur Spark
4. Génère le flux final de résultats par lots

Principe clé :

```
1 Batch (lot) = 1 RDD
```

7 Spark Structured Streaming (SSS)

7.1 Définition

— Moteur de traitement de flux basé sur Spark SQL
— Évolutif et tolérant aux pannes
— Calcul de streaming comme calcul sur données statiques
— **Même API pour streaming et batch**

7.2 Architecture

7.2.1 Traitement par micro-lots

Par défaut :

— Traitement par micro-lots
— Traite les flux comme série de petits lots

— Latences : Aussi faibles que 100 millisecondes
— Garantie : **exactly-once delivery**

7.2.2 Mode de traitement continu

Depuis Spark 2.3 :

— Mode continuous processing
— Latences : Aussi faibles qu'1 milliseconde
— Garantie : at-least-once delivery
— Choix du mode selon les besoins

7.3 Modes de tolérance aux pannes

1. **At-most-once (0 or 1)**
   — Message délivré zéro ou une fois
   — Messages peuvent être perdus
2. **At-least-once (1 or more)**
   — Message délivré au moins une fois
   — Multiples tentatives de délivrance
   — Messages peuvent être dupliqués mais pas perdus
3. **Exactly-once (1)** ✓
   — Une seule délivrance au destinataire
   — Message ni perdu ni dupliqué
   — **Mode par défaut en micro-lots**

8 Modèle de programmation SSS

8.1 Concept de table

Principe fondamental :

— Chaque élément de données = nouvelle ligne dans table d'entrée
— Streaming = ajout continu de lignes à la table
— Requête sur l'entrée génère la "Table des résultats"
— Mise à jour incrémentale de la table de résultats

8.2 Tables dans SSS

1. **Table d'entrée** : Flux de données reçues
2. **Table de résultats** : Résultats des requêtes
3. **Stockage externe** : Sauvegarde des résultats

8.3 Modes de sortie (Output Modes)

8.3.1 1. Mode Complet (Complete)

— Ensemble du tableau des résultats écrit au stockage
— Utilisé pour agrégations complètes

— Exemple : Compte total de tous les mots

Exemple :

```
// Chaque d clencheur ecrit TOUT le tableau
Declencheur 1: Cat=1, Dog=3
Declencheur 2: Cat=2, Dog=3, Owl=1
Declencheur 3: Cat=2, Dog=4, Owl=2
```

8.3.2 2. Mode Ajout (Append)

— Seules les nouvelles lignes écrites

— Lignes ajoutées depuis le dernier déclencheur

— Utilisé pour données qui ne changent jamais

Exemple :

```
// Seulement les nouvelles lignes
Declencheur 1: Cat=1, Dog=3
Declencheur 2: Owl=1
Declencheur 3: Dog=1, Owl=1
```

8.3.3 3. Mode Mise à jour (Update)

— Seules les lignes mises à jour écrites

— Disponible depuis Spark 2.1.1

— Optimisé pour agrégations changeantes

Exemple :

```
// Seulement les lignes modifiees
Declencheur 1: Cat=1, Dog=3
Declencheur 2: Cat=2, Owl=1
Declencheur 3: Dog=4, Owl=2
```

8.4 Optimisations

SSS ne matérialise pas la table entière :

— Lit les dernières données disponibles

— Traite de manière incrémentale

— Supprime les données source après traitement

— Conserve uniquement l'état intermédiaire minimal

9 Gestion du temps et données tardives

9.1 Event-time (Heure de l'événement)

— Heure incorporée dans les données

— Reflète quand l'événement s'est réellement produit

— Utilisable pour la plupart des applications

Exemple d'utilisation :

— Compter événements IoT générés chaque minute
— Agrégations basées sur fenêtres de temps
— Type spécial de regroupement sur colonne Event-time

9.2 Données tardives (Late Data)

Gestion naturelle :

— Modèle gère données arrivées en retard
— Basé sur leur heure d'événement
— Spark met à jour la table de résultats
— Contrôle total sur mise à jour des anciens agrégats

9.3 Watermarking

Depuis Spark 2.1 :

— Support du watermarking
— Permet de spécifier le seuil de données en retard
— Supprime les états obsolètes
— Optimise l'utilisation mémoire

```
val windowedCounts = events
  .withWatermark("timestamp", "10␣minutes")
  .groupBy(
    window($"timestamp", "5␣minutes"),
    $"word"
  )
  .count()
```

10 Exemple WordCount en Streaming

10.1 Objectif

— Calculer le nombre d'occurrences des mots en mode streaming
— Écouter le port 9999
— Recevoir des phrases
— Compter les mots en temps réel

10.2 Code exemple

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

// Creation de la SparkSession
val spark = SparkSession
  .builder
  .appName("StructuredNetworkWordCount")
  .getOrCreate()

import spark.implicits._
```

```
// Creation du DataFrame de streaming
val lines = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()

// Transformation: split et count
val words = lines.as[String]
  .flatMap(_.split(" "))

val wordCounts = words
  .groupBy("value")
  .count()

// Ecriture de la sortie
val query = wordCounts.writeStream
  .outputMode("complete")
  .format("console")
  .start()

query.awaitTermination()
```

11 Notions fondamentales de Spark Streaming

11.1 1. Caching

Principe :

- Permet de mettre en cache les données du flux en mémoire
- Utile pour transformations coûteuses réutilisées
- Utilise `persist()` ou `cache()`

Utilité :

- Réutilisation du même DataFrame dans sinks différents
- Calculs lourds répétés dans pipeline

```
val transformed = inputDf
  .withColumn("value_double", col("value") * 2)
  .cache() // ou .persist(StorageLevel.MEMORY_AND_DISK)

val query = transformed.writeStream
  .format("console")
  .start()
```

11.2 2. Accumulateurs

Caractéristiques :

- Implémentent des compteurs ou des sommes
- Suivi sur l'UI de Spark Streaming
- Partagés entre tous les executors
- Mise à jour uniquement par les workers

11.3 3. Variables de diffusion (Broadcast Variables)

Principe :

— Variables diffusées sur chaque machine

— Variable en lecture seule en cache sur chaque machine

— Utile pour transformations larges (avec shuffling)

— Optimise les opérations de lookup

Exemple d'utilisation :

— Diffuser une table de mapping

— Éviter de répéter les données dans chaque tâche

11.4 4. Points de contrôle (Checkpoints)

11.4.1 Objectif

— Application streaming fonctionne 24/7

— Doit être résiliente aux défaillances

— Spark Streaming sauvegarde les informations critiques

— Permet la récupération après panne

11.4.2 Types de checkpoints

1. Points de contrôle des métadonnées :

— Sauvegarde dans stockage tolérant aux pannes (HDFS)

— Configuration de l'application

— Lots incomplets (démarrés mais pas enregistrés)

— Récupération des pannes du driver

2. Points de contrôle des données :

— Sauvegarde des RDDs générés

— Stockage dans système fiable

— Permet de reconstruire l'état

```
// Configuration du checkpoint
streamingContext.checkpoint("hdfs://checkpoint_dir")

val lines = ssc.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split("␣"))
val pairs = words.map(word => (word, 1))

// Checkpoint automatique pour operations avec etat
val wordCounts = pairs.reduceByKey(_ + _)
```

12 Déclencheurs et intervalles

12.1 Principe

— Chaque intervalle de déclenchement (ex : chaque seconde)

— Nouvelles lignes ajoutées à la table d'entrée

— Mise à jour de la table de résultats

— Écriture sur stockage externe selon le mode de sortie

13 Comparaison des approches streaming

13.1 Spark Streaming vs Structured Streaming

Critère	Spark Streaming	Structured Streaming
API de base	DStream (RDD)	DataFrame/Dataset
Abstraction	Flux de RDDs	Table infinie
Optimisation	Manuelle	Catalyst + Tungsten
Latence	Centaines de ms	Aussi faible que 1 ms
API	Bas niveau	Haut niveau
Event-time	Support limité	Support natif
Late data	Difficile	Géré automatiquement

14 Cas d'usage pratiques

14.1 Spark ML

Applications typiques :

— Classification de clients (churn prediction)

— Systèmes de recommandation

— Détection de fraudes

— Analyse de sentiment

— Prédiction de séries temporelles

— Clustering de documents

14.2 Spark Streaming

Applications typiques :

— Analyse de logs en temps réel

— Détection d'anomalies

— Monitoring IoT

— Analyse de réseaux sociaux

— Recommandations en temps réel

— Agrégation de métriques

15 Best Practices

15.1 Spark ML

1. **Préparation des données**
   — Nettoyage et normalisation en amont
   — Gestion appropriée des valeurs manquantes
   — Feature engineering avant le pipeline

2. **Pipelines**
   - Toujours utiliser des pipelines pour reproductibilité
   - Sauvegarder les pipelines complets
   - Versioning des modèles
3. **Tuning**
   - Utiliser CrossValidator pour éviter l'overfitting
   - Tester plusieurs algorithmes
   - Optimiser les hyperparamètres systématiquement
4. **Performance**
   - Cacher les DataFrames intermédiaires
   - Utiliser le bon nombre de partitions
   - Optimiser la mémoire des executors

15.2 Spark Streaming

1. **Configuration**
   - Ajuster la taille des micro-lots
   - Configurer correctement les checkpoints
   - Optimiser le parallélisme
2. **Gestion de l'état**
   - Utiliser watermarking pour données tardives
   - Nettoyer l'état ancien régulièrement
   - Choisir le bon mode de sortie
3. **Tolérance aux pannes**
   - Activer les checkpoints
   - Utiliser exactly-once semantics
   - Tester la récupération après panne
4. **Performance**
   - Minimiser les shuffles
   - Utiliser broadcast pour petites données
   - Monitorer les métriques de latence

16 Points clés à retenir

16.1 Spark ML

1. MLlib standardise les APIs ML sur DataFrames
2. Pipeline = chaîne de Transformers et Estimators
3. Transformer : DataFrame → DataFrame (transform)
4. Estimator : DataFrame → Transformer (fit)
5. Evaluator : Évalue la performance (evaluate)
6. CrossValidator automatise le model tuning
7. ParamGridBuilder teste toutes les combinaisons

8. Toujours valider avec K-Fold cross-validation
9. Persistance des pipelines pour réutilisation
10. Optimisation critique pour Big Data + IA

16.2 Spark Streaming

1. Structured Streaming = streaming sur DataFrames
2. Même API pour batch et streaming
3. Concept de table infinie en croissance
4. 3 modes de sortie : Complete, Append, Update
5. Exactly-once semantics par défaut
6. Event-time support natif avec watermarking
7. Latences : 100ms (micro-lots) à 1ms (continu)
8. Checkpoints essentiels pour tolérance aux pannes
9. Caching pour optimiser les transformations répétées
10. Broadcast variables pour optimiser les shuffles

16.3 Intégration ML + Streaming

— Possibilité d'appliquer ML sur flux de données
— Modèles pré-entraînés utilisables en streaming
— Prédictions en temps réel sur données streaming
— Réentraînement périodique des modèles
— Architecture Lambda : batch + streaming

17 Architecture type Big Data + IA

17.1 Composants

1. **Ingestion**
   — Kafka, Flume pour streaming
   — HDFS, S3 pour batch
2. **Traitement**
   — Spark Streaming pour données temps réel
   — Spark Batch pour données historiques
3. **Machine Learning**
   — MLlib pour entraînement
   — Pipelines pour déploiement
   — Model serving pour prédictions
4. **Stockage**
   — HDFS pour données brutes
   — Hive/Parquet pour données structurées
   — HBase/Cassandra pour requêtes rapides

5. **Visualisation**
   - Tableaux de bord temps réel
   - Outils BI (Tableau, Power BI)
   - APIs de prédiction

17.2 MLOps avec Spark

- Versioning des données et modèles
- CI/CD pour pipelines ML
- Monitoring de la performance des modèles
- Réentraînement automatique
- A/B testing des modèles
- Gestion du drift des données

18 Conclusion

18.1 Défis relevés

- Spark ML : Apprentissage automatique à l'échelle
- Spark Streaming : Traitement temps réel
- Optimisations : Catalyst, Tungsten, caching
- Tolérance aux pannes : Checkpoints, exactly-once
- APIs unifiées : Même code batch/streaming

18.2 Écosystème complet

Spark offre une plateforme unifiée pour :

- Traitement batch (Spark Core, SQL)
- Traitement streaming (Structured Streaming)
- Machine Learning (MLlib)
- Traitement de graphes (GraphX)
- Support multi-langages (Scala, Python, Java, R)

18.3 Prochaines étapes

- Maîtriser le tuning de Spark
- Pratiquer avec des datasets réels
- Comprendre les métriques de performance
- Expérimenter avec différents algorithmes ML
- Implémenter des architectures streaming complètes
- Explorer MLOps et déploiement en production