Programmation fonctionnelle

Projet Spark

Aurélie FERAUD
Brenda SEPIERE
Alexandre ABDELATIF
Franklin BRASA
Camille LABORDE

ING2 GMA S1
2024 - 2025

Sommaire

Glossaire

RDD	Resilient Distributed Dataset (Jeu de données distribué résilient)
JDK	Java Development Kit (Kit de développement Java)
SDKMAN	Software Development Kit Manager (Gestionnaire de kits de développement logiciel)
IDE	Integrated Development Environment (Environnement de développement intégré)
DAG	Directed Acyclic Graph (Graphe acyclique orienté)
SVM	Support Vector Machine (Machine à vecteurs de support)
GLoVe	Global Vectors for Word Representation (Vecteurs globaux pour la représentation des mots)
VertexRDD[VD]	Vertex Resilient Distributed Dataset [Données du sommet]
EdgeRDD[ED]	Edge Resilient Distributed Dataset [Données de l'arête]
VertexId	Identifiant de sommet
API	Application Programming Interface (Interface de programmation d'applications)
GraphX	Une bibliothèque dans Apache Spark pour le traitement des graphes
PageRank	Un algorithme utilisé par Google Search pour classer les pages web
Spark Streaming	Un composant de Spark qui traite les flux de données en temps réel

Synthèse de l’installation de Spark

Dans le cadre de notre projet, nous avons dû installer Apache Spark sur nos ordinateurs afin de disposer d’un environnement performant pour traiter et analyser des données massives. L’installation de Spark n’est pas immédiate et nécessite de suivre une série d’étapes pour garantir son bon fonctionnement. Dans ce rapport, nous détaillons comment nous avons procédé, depuis la préparation de l’environnement jusqu’à la vérification de l’installation et la configuration de notre environnement de développement. Afin d’effectuer cette installation, nous avons dû procéder à un certain nombre d’étapes.

Pour commencer, la première étape a consisté à nous assurer que nos ordinateurs possédaient les prérequis nécessaires pour l’installation de Spark. Étant donné que Spark repose sur Java, nous avons vérifié la présence du Java Development Kit (JDK) en exécutant la commande “java -version” dans le terminal. Dans notre cas, certains d’entre nous ont dû installer Java, tandis que d’autres avaient déjà une version récente et compatible avec Spark, ce qui leur a permis de passer directement à l’étape suivante sans avoir à installer un JDK.

Ensuite, comme nous prévoyons d’utiliser Spark principalement avec Scala, nous avons également installé Scala. Pour simplifier cette tâche, nous avons utilisé SDKMAN, un gestionnaire pratique pour l’installation de langages de programmation. Enfin, pour préparer l’utilisation de PySpark, nous avons confirmé que Python était correctement installé sur nos machines, en exécutant “python --version”. Nous avons ainsi assuré une compatibilité avec les bibliothèques Spark.

Après avoir vérifié les prérequis, nous sommes passés au téléchargement d’Apache Spark. Nous avons téléchargé la version la plus récente depuis le site officiel de Spark (spark.apache.org) et avons sélectionné une version compatible avec notre environnement.

Après le téléchargement, nous avons extrait les fichiers compressés dans des répertoires dédiés sur nos machines, par exemple C:\spark pour ceux sous Windows. Ce dossier est devenu le point central pour toutes les potentielles futures configurations.

Comme nous étions tous sous Windows, la configuration des variables d’environnement a été réalisée via les paramètres système. Nous avons commencé par définir la variable SPARK_HOME, qui pointait vers le dossier où Spark avait été extrait, par exemple C:\spark. Cette variable permet de référencer l’emplacement exact de l’installation de Spark sur nos machines.

Ensuite, nous avons ajouté le dossier des binaires de Spark à la variable système PATH. Cela nous a permis d’exécuter des commandes Spark depuis n’importe quel répertoire dans l’invite de commandes. Pour effectuer cette configuration, nous sommes allés dans les "Propriétés système avancées", puis dans "Variables d’environnement". Nous avons ajouté une nouvelle variable utilisateur pour SPARK_HOME et modifié la variable PATH en y ajoutant le chemin suivant : %SPARK_HOME%\bin.

Une fois les variables d'environnement configurées, nous avons vérifié que Spark fonctionnait correctement sur nos ordinateurs. Pour cette étape, nous avons ouvert l'invite de commandes et exécuté la commande "spark-shell". Cette commande nous a permis de lancer l'interface interactive de Spark pour Scala. La console Spark s'est ouverte avec succès, confirmant que Spark était bien installé et fonctionnel.

Pour optimiser notre travail avec Spark, nous avons décidé d'adopter un IDE, et nous avons choisi IntelliJ IDEA. Nous y avons donc installé le plugin Scala, ce qui nous a permis de créer des projets spécifiquement conçus pour le développement avec Spark. Nous avons ensuite configuré IntelliJ afin qu'il utilise les versions de Java et Scala installées précédemment. Une fois cette étape terminée, nous avons ajouté les dépendances nécessaires pour Spark afin que le projet soit opérationnel, ce qui nous a permis de compléter l'installation de Spark.

Les RDDs dans Apache Spark

Aujourd'hui, la quantité de données croît de manière exponentielle, et le besoin de technologies capables de traiter efficacement de vastes volumes d'informations est devenu crucial. Apache Spark, avec ses Resilient Distributed Datasets (RDDs), offre une solution robuste pour manipuler des données distribuées à grande échelle.

Définition des RDDs

Les RDDs sont des collections immuables et distribuées de données, partitionnées sur les nœuds (plusieurs ordinateurs) d'un cluster Spark (Master et workers). Ils représentent la structure de données de base de Spark et constituent les éléments fondamentaux de certains types de données tels que les DataFrames et les Datasets.

Les RDDs sont particulièrement utiles pour traiter des données non structurées (par exemple des fichiers texte). Ils permettent d'appliquer des transformations comme le filtrage ou le tri grâce à des fonctions intégrées de Spark.

Caractéristiques des RDDs

- Résilience : En cas de panne, Spark peut recréer un RDD à partir de sa lignée (historique des transformations).
- Immutabilité : Une fois créé, un RDD ne peut être modifié. Les transformations génèrent un nouveau RDD.
- Distribution : Les données sont réparties sur plusieurs machines, permettant un traitement parallèle rapide et efficace.

- Lazy Evaluation : Les transformations ne sont exécutées qu'au moment d'une action (par exemple, collect() ou count()).
- Calcul en mémoire : Les données peuvent être stockées en RAM, ce qui améliore considérablement les performances.

Schéma : RDD distribué dans un cluster

RDD : ["chien", "chat", "grenouille", "cheval"]

Partition 1 : ["chien", "chat"] ==> Machine 1
Partition 2 : ["grenouille"] ==> Machine 2
Partition 3 : ["cheval"] ==> Machine 3
Un RDD est divisé en partitions, chacune étant traitée par une machine différente dans le cluster.

Création d'un RDD

Il existe plusieurs méthodes pour créer un RDD :

Parallélisation d'une collection

Une collection Scala existante peut être transformée en RDD à l'aide de la méthode parallelize :

```
val animaux = List("chien", "chat", "grenouille", "cheval")
val animauxRDD = sc.parallelize(animaux)
```

Lecture de fichiers externes

Les données peuvent être chargées directement depuis un fichier texte :

```
val data = sc.textFile("data.txt")
```

Conversion depuis un DataFrame ou un Dataset

Un DataFrame peut être converti en RDD via la méthode .rdd :

```
val dataDF = spark.read.csv("data.csv")
val rdd = dataDF.rdd
```

Opérations sur les RDDs

Les opérations sur les RDDs se divisent en deux catégories principales : les transformations et les actions.

Les transformations

Les transformations créent de nouveaux RDDs à partir d'un RDD existant. Ces transformations sont lazy, c'est-à-dire qu'elles ne sont évaluées que lorsqu'une action est appelée.

Exemples de transformations :
map()
filter()
flatMap()

Exemple de code pour ajouter la longueur de chaque mot :

```
val animaux = List("chien", "chat", "grenouille", "cheval")
val animauxRDD = sc.parallelize(animaux)

// Transformation : ajouter la longueur de chaque mot
val longueurRDD = animauxRDD.map(animal => (animal, animal.length))
```

Les actions

Les actions exécutent les calculs définis par les transformations et renvoient les résultats au programme principal.

Exemples d'actions :
collect()
count()
reduce()

Exemple de code pour calculer la somme :

```
val nombres = sc.parallelize(List(1, 2, 3, 4, 5))
val somme = nombres.reduce(_ + _)
```

RDDs à paires clé-valeur

Les RDDs peuvent également représenter des paires clé-valeur, très utiles pour grouper ou agréger des données. Voici un exemple avec une jointure :

```
val rddUn = sc.parallelize(List((1, "chat"), (2, "chien"), (3, "grenouille")))
val rddDeux = sc.parallelize(List((1, "mammifère"), (3, "amphibien")))
```

Jointure sur la clé :

```
val rddJoint = rddUn.join(rddDeux)
```

Performances et optimisation

Les RDDs permettent de suivre leur plan d'exécution grâce à une lignée (DAG - Directed Acyclic Graph), qui enregistre toutes les transformations avant leur calcul réel. Cela permet d'optimiser les performances et de rejouer les étapes en cas de panne.

Exemple d'utilisation du DAG avec des transformations :

```
val data = sc.parallelize(List(1, 2, 3, 4, 5))
val filteredRdd = data.filter(_ % 2 == 0)
val mappedRdd = filteredRdd.map(_ * 2)

// Affiche la lignée des transformations
println(mappedRdd.toDebugString)
```

Sortie :

```
(8) MapPartitionsRDD[2] at map at <console>:26 []
| MapPartitionsRDD[1] at filter at <console>:25 []
| ParallelCollectionRDD[0] at parallelize at <console>:24 []
```

Conclusion

Les RDDs constituent la base d'Apache Spark et offrent une flexibilité inégalée pour manipuler des données non structurées. Leur résilience, immutabilité et évaluation paresseuse en font un outil puissant pour traiter des volumes massifs de données dans un environnement distribué. Toutefois, pour des cas nécessitant des performances optimales, les DataFrames et Datasets sont souvent préférés.

GraphX dans Apache Spark

Avec l'augmentation rapide des données connectées, telles que celles générées par les réseaux sociaux, les systèmes de recommandation et les analyses de relations complexes, il est essentiel de disposer d'outils capables de traiter efficacement ces informations sous forme de graphes. Apache Spark propose GraphX, un module puissant dédié au traitement et à l'analyse des graphes distribués.

Définition de GraphX

GraphX est un composant de Spark permettant de représenter et de traiter des données relationnelles sous forme de graphes orientés, y compris des multigraphes. Dans GraphX, chaque sommet (vertex) et arête (edge) du graphe peut contenir des propriétés associées, ce qui permet d'enrichir les relations modélisées.

GraphX repose sur deux structures principales :

- VertexRDD[VD] : Représente un RDD de sommets avec leurs propriétés (ID, valeur).
- EdgeRDD[ED] : Représente un RDD d'arêtes avec leurs propriétés (source, destination, valeur).

Caractéristiques de GraphX

- Multigraphe orienté : Permet de gérer des graphes dirigés avec plusieurs types d'arêtes entre deux sommets, idéal pour des applications comme les réseaux sociaux ou les recommandations.
- Flexibilité et optimisation : Supporte des transformations et jointures basées sur les RDDs, avec des optimisations pour le traitement parallèle.
- Traitement distribué : Exploite la puissance de Spark pour effectuer des calculs distribués à grande échelle, tout en offrant une tolérance aux pannes.
- Algorithmes intégrés : Inclut des algorithmes courants pour les graphes comme PageRank, les composants connectés et le comptage de triangles.
- API Pregel : Permet l'écriture d'algorithmes itératifs personnalisés, avec une API spécifique pour les messages échangés entre les sommets.

Création d'un Graphe

GraphX permet de créer des graphes à partir de collections ou de fichiers. Voici un exemple pour créer un graphe avec des sommets et des arêtes :

```
val utilisateurs: RDD[(VertexId, (String, String))] = sc.parallelize(Seq(
  (3L, ("rxin", "étudiant")),
  (7L, ("jgonzal", "postdoc")),
  (5L, ("franklin", "prof")),
  (2L, ("istoica", "prof"))
))

val relations: RDD[Edge[String]] = sc.parallelize(Seq(
  Edge(3L, 7L, "collab"),
  Edge(5L, 3L, "conseiller"),
  Edge(2L, 5L, "collègue"),
  Edge(5L, 7L, "pi")
))

val defaultUser = ("John Doe", "Manquant")
val graph = Graph(utilisateurs, relations, defaultUser)
```

Opérateurs dans GraphX

Opérateurs structurels

inverse() : Inverse les arêtes d'un graphe (source ↔ destination).
subgraph() : Filtre les sommets et arêtes répondant à des critères spécifiques.
mask() : Crée un sous-graphe à partir d'un masque basé sur un autre graphe.
Exemple pour extraire un sous-graphe des "professeurs" :

```
val sousGraphe = graph.subgraph(vpred = (id, attr) => attr._2 == "prof")
```

Opérateurs de jointure

joinVertices() : Fusionne des propriétés supplémentaires aux sommets existants.
outerJoinVertices() : Jointure externe pour ajouter des informations même si elles sont partielles.
Exemple pour enrichir les sommets avec des données supplémentaires :

```
val extraData: RDD[(VertexId, String)] = sc.parallelize(Seq((3L, "Paris"), (7L, "Lyon")))
val enrichi = graph.joinVertices(extraData)((id, user, location) => (user._1, user._2, location))
```

Messages agrégés

aggregateMessages() : Envoie et agrège des messages personnalisés entre les sommets.
Exemple pour envoyer un message à tous les voisins et calculer la somme des messages :

```
val voisins = graph.aggregateMessages[Int](
  triplet => triplet.sendToSrc(1),
  _ + _
)
```

Algorithmes intégrés

GraphX propose plusieurs algorithmes optimisés pour l'analyse des graphes :

- PageRank : Classe l'importance des sommets.
- Connected Components : Identifie les groupes connectés.
- Triangle Count : Compte les triangles formés par les arêtes.

Exemple pour calculer le PageRank :

```
val ranks = graph.pageRank(0.15).vertices
```

Optimisation et performances

GraphX exploite les optimisations de Spark pour le traitement des graphes :

- Partitionnement intelligent : Réduit les déplacements de données entre les machines.

- Caching : Stocke en mémoire les données fréquemment utilisées.
- Checkpointing : Sauvegarde l'état du graphe pour assurer la tolérance aux pannes.

Exemple pour analyser le plan d'exécution du graphe :

```
println(graph.vertices.toDebugString)
```

Conclusion

GraphX est un module puissant d'Apache Spark qui facilite le traitement et l'analyse des graphes distribués. Grâce à sa flexibilité, sa compatibilité avec les RDDs et ses algorithmes intégrés, il permet d'exploiter efficacement des données relationnelles complexes. Il est particulièrement adapté aux cas d'usage tels que l'analyse des réseaux sociaux, les systèmes de recommandation et la détection de fraudes. Toutefois, pour des performances optimales, il peut être intéressant d'explorer des alternatives comme GraphFrames ou des bases de données spécialisées dans les graphes.

Explication détaillée du projet

Objectifs du Projet

Le projet vise à développer une application capable d'analyser les sentiments exprimés sur Twitter en temps réel en utilisant Apache Spark et Spark Streaming. L'objectif principal est de se connecter à l'API Twitter pour récupérer un flux continu de tweets concernant un sujet spécifique. Une fois les données collectées, le projet inclut plusieurs étapes de traitement : nettoyage des données, analyse des sentiments, et présentation des résultats sous forme de statistiques agrégées.

Pour y parvenir, le projet se concentre sur l'utilisation de modèles de machine learning pour classifier les tweets en catégories de sentiments (positif, négatif ou neutre). En outre, l'application doit permettre de visualiser l'évolution des sentiments sur une période donnée à l'aide de fenêtres temporelles, et les résultats doivent être stockés dans une base de données en temps réel, comme Cassandra.

L'application doit aussi être optimisée pour garantir des performances élevées et capable de gérer un grand volume de données, notamment grâce aux capacités de mise à l'échelle de Spark.

Fichiers du Projet

Le projet comprend plusieurs fichiers Scala, chacun étant responsable d'une partie spécifique du processus. Ces fichiers sont cruciaux pour le bon fonctionnement de l'application.

Le fichier *ParameterTuning.scala* joue un rôle clé dans le projet en réglant les paramètres des modèles de machine learning utilisés pour l'analyse des sentiments. Ce fichier permet d'ajuster les paramètres des modèles comme la régression logistique et la forêt aléatoire, ce qui est essentiel pour optimiser les performances et garantir des résultats précis. L'ajustement des paramètres se fait généralement par validation croisée, une technique qui permet de tester différentes configurations de modèles et de choisir celle qui donne les meilleurs résultats.

Le fichier *SentimentAnalyzer.scala* est au cœur de l'analyse des sentiments. Il charge plusieurs modèles de machine learning, notamment la régression logistique, le SVM, et la forêt aléatoire, et les applique aux tweets collectés en temps réel. Ce fichier permet de classifier chaque tweet en fonction de son sentiment, qu'il soit positif, négatif ou neutre. Il joue également un rôle important dans la gestion des modèles en définissant une carte associant chaque modèle à son étiquette respective.

Le fichier *TextClassifierTrain.scala* est responsable de l'entraînement des modèles de machine learning avant qu'ils ne soient utilisés dans l'analyse des tweets. Il utilise des jeux de données d'entraînement pour apprendre aux modèles à classer les sentiments des textes. Ce fichier prend en charge des algorithmes tels que la régression logistique ou le SVM, en les entraînant sur des données préparées.

Le fichier *TrainModels.scala* est destiné à la création et l'entraînement des différents modèles de classification. Il intègre toutes les étapes nécessaires dans un pipeline Spark, y compris la tokenisation et la normalisation des textes, qui sont des étapes cruciales pour préparer les données avant de les soumettre à un modèle de machine learning. Ce fichier permet donc de mettre en place le pipeline de machine learning et d'entraîner les modèles en vue de leur utilisation en temps réel.

Le dossier model contient un fichier appelé *TextClassifier.scala*, qui définit le modèle de classification utilisé pour analyser les tweets. Ce fichier utilise des embeddings pré-entraînés de GloVe, une technique permettant de transformer des mots en vecteurs numériques, ce qui est essentiel pour l'analyse de texte. En utilisant ces embeddings, le modèle peut représenter chaque mot sous forme de vecteur dense et utiliser ces représentations pour effectuer des prédictions sur les sentiments des tweets.

Enfin, dans le dossier utilities, le fichier *Cleaner.scala* définit un transformateur personnalisé qui est utilisé pour nettoyer les tweets avant leur analyse. Ce nettoyage comprend la suppression des mentions, hashtags, URLs et autres éléments non pertinents qui pourraient nuire à l'analyse des sentiments. Ce fichier est essentiel pour préparer les données avant qu'elles ne soient soumises au modèle.

Points Clés à Clarifier

L'utilisation de Apache Spark et de Spark Streaming pour traiter les tweets en temps réel est un élément fondamental du projet. Spark permet de traiter des volumes massifs de données de manière distribuée et parallèle, ce qui est crucial pour le traitement en temps réel des tweets. Spark Streaming facilite la gestion des flux de données entrants, en permettant à l'application d'analyser les tweets dès leur arrivée.

Les embeddings GloVe sont une autre composante importante du projet. Ces embeddings pré-entraînés permettent de convertir des mots en vecteurs denses, ce qui rend les textes compréhensibles par les modèles de machine learning. L'utilisation de GloVe permet de profiter d'un modèle déjà optimisé, évitant ainsi de devoir entraîner ces embeddings depuis zéro. Cependant, cela nécessite une gestion adéquate de ces vecteurs de mots, ce qui peut être complexe pour ceux qui ne sont pas familiers avec les concepts de représentation du texte.

L'ajustement des paramètres des modèles est également une étape essentielle. Cela permet de maximiser la précision des modèles de machine learning. L'une des techniques utilisées est la validation croisée, qui consiste à diviser les données en plusieurs sous-ensembles pour tester

différentes configurations et déterminer laquelle donne les meilleurs résultats. Ce processus d'optimisation est essentiel pour éviter le sur-apprentissage et garantir des performances robustes.

Résumé du Cheminement du Projet

Le projet commence par la collecte des données à l'aide de Spark Streaming, qui se connecte à l'API Twitter pour récupérer un flux continu de tweets en temps réel. Une fois les tweets récupérés, ils sont traités dans Cleaner.scala pour éliminer les informations inutiles. Le texte nettoyé est ensuite tokenisé, et les mots vides sont supprimés, avant d'être soumis aux modèles d'analyse de sentiments.

Les modèles de machine learning sont entraînés à l'avance dans TextClassifierTrain.scala et TrainModels.scala, utilisant des jeux de données d'entraînement pour apprendre à classifier les tweets. Ces modèles sont ensuite utilisés en temps réel dans SentimentAnalyzer.scala pour prédire le sentiment de chaque tweet.

Les résultats sont agrégés et visualisés sur un tableau de bord interactif, tandis que les résultats sont stockés dans une base de données en temps réel pour une gestion continue. Les performances du système sont optimisées tout au long du projet pour garantir une exécution fluide et rapide.

Bibliographie

- Contenu de notre programme : https://github.com/Camoht/Spark
- Partie 1 : Call with Camille Laborde-20250112_112213-Meeting Recording.mp4
  Partie 2 :
  https://ucergyfr-my.sharepoint.com/:v:/g/personal/brenda_sepiere_etu_cyu_fr/EYC3bGMZBBdAnqJ9zpLIHQsBQR0eVpRLE7liNVsDv-21vA?e=j5srbb