# GraphX

Table des matières

[GraphX 1](#_Toc187588583)

[Introduction 1](#_Toc187588584)

[Méthodologie 1](#_Toc187588585)

[Préparation de l'environnement Spark 1](#_Toc187588586)

[Chargement et nettoyage des données 2](#_Toc187588587)

[Construction du graphe 3](#_Toc187588588)

[Analyse du graphe 3](#_Toc187588589)

[PageRank : Détection des nœuds influents 3](#_Toc187588590)

[Problèmes rencontrés 4](#_Toc187588591)

[Améliorations possibles 4](#_Toc187588592)

[Conclusion 4](#_Toc187588593)

[Annexes 5](#_Toc187588594)

## **Introduction**

Dans ce projet, nous avons exploré l’analyse des tweets en utilisant **Apache Spark** et sa bibliothèque **GraphX**, dédiée au traitement de graphes distribués. Les données Twitter, riches en relations implicites (mentions, hashtags, etc.), ont été modélisées sous forme de graphes pour identifier les utilisateurs influents et les communautés.

**Objectifs principaux :**

1. Nettoyer et préparer les données brutes.
2. Construire un graphe des relations.
3. Appliquer des algorithmes d’analyse comme **PageRank** et **Connected Components**.
4. Extraire des insights exploitables tout en documentant les défis rencontrés.

## **Méthodologie**

### **Préparation de l'environnement Spark**

Nous avons configuré Spark pour une exécution locale en mode distribué :

val spark = SparkSession.builder()

.appName("Analyse des Tweets avec GraphX")

.master("local[\*]") // Utilise tous les cœurs disponibles

.getOrCreate()

* **SparkSession** : Point d’entrée pour Spark SQL.
* **local[\*]** : Utilise tous les cœurs pour maximiser l’utilisation des ressources.

### **Chargement et nettoyage des données**

Les données brutes ont été chargées dans un **DataFrame** grâce à l’API Spark SQL :

val tweetsDF = spark.read

.option("header", "true")

.option("inferSchema", "true")

.csv("twitter\_comment.csv")

Pour nettoyer les données, nous avons utilisé des expressions régulières pour extraire :

1. **Mentions** (@user) : Les utilisateurs mentionnés dans les tweets.
2. **Hashtags** (#hashtag) : Les thématiques ou mots-clés utilisés.

Cette extraction a été effectuée à l’aide de la fonction UDF suivante basé sur les expressions régulières :

def regexp\_extract\_all(column: org.apache.spark.sql.Column, pattern: String): org.apache.spark.sql.Column = {

val extractAll = udf((text: String) => {

if (text == null) Seq.empty[String]

else pattern.r.findAllIn(text).toSeq

})

extractAll(column)

Le DataFrame nettoyé contient les colonnes suivantes :

* mentions : Tableau des utilisateurs mentionnés.
* hashtags : Tableau des hashtags présents.

Exemple après nettoyage :

+----------+--------------------+---------------+------------------+--------+

| tweet\_id | date | user\_name | mentions | hashtags |

+----------+--------------------+---------------+------------------+--------+

| 1467810369 | Mon Apr 06 ... | \_TheSpecialOne\_ | [@user] | [#topic] |

### **Construction du graphe**

Le graphe est construit à partir de deux éléments :

**Les sommets (vertices)** : Représentent les utilisateurs, mentions et hashtags. Chaque sommet est identifié par un VertexId basé sur le **hashcode** du nom.

val vertices = tweetsCleanedDF.rdd.flatMap(row => {

val user = row.getAs[String]("user\_name")

val mentions = row.getAs[Seq[String]]("mentions")

val hashtags = row.getAs[Seq[String]]("hashtags")

(Seq(user) ++ mentions ++ hashtags).distinct.map(name => (name.hashCode.toLong, name))

})

**Les arêtes (edges)** : Représentent les relations entre les sommets :

* + mention entre un utilisateur et les utilisateurs mentionnés.
  + hashtag entre un utilisateur et les hashtags associés.

val edges = tweetsCleanedDF.rdd.flatMap(row => {

val user = row.getAs[String]("user\_name")

val mentions = row.getAs[Seq[String]]("mentions")

val hashtags = row.getAs[Seq[String]]("hashtags")

mentions.map(mention => Edge(user.hashCode.toLong, mention.hashCode.toLong, "mention")) ++

hashtags.map(hashtag => Edge(user.hashCode.toLong, hashtag.hashCode.toLong, "hashtag"))

})

### Analyse du graphe

### **PageRank : Détection des nœuds influents**

L’algorithme **PageRank** a été appliqué pour attribuer un score à chaque nœud, basé sur son importance dans le réseau :

val ranks = graph.pageRank(0.0001).vertices

* **Score élevé** : Les nœuds fréquemment reliés par des mentions ou hashtags.
* Les 10 nœuds les plus influents sont triés et affichés :

ranks.join(vertices)

.map { case (id, (rank, name)) => (name, rank) }

.reduceByKey(Math.max) // Retirer les doublons

.sortBy(-\_.\_2) // Trier par score décroissant

.take(10)

*Exemple de sortie :*

@mileycyrus Score: 838.36079

@ddlovato Score: 749.21369

#### **4.2. Connected Components : Groupes connectés**

L’algorithme **Connected Components** identifie les sous-graphes où chaque nœud est relié à d’autres directement ou indirectement. Cela permet de découvrir des communautés ou clusters d’utilisateurs.

val connectedComponents = graph.connectedComponents().vertices

Exemple de composants connectés :

Composant 144522792 :

- kayesheys

- @toofache

## **Problèmes rencontrés**

1. **Gestion des collections immuables** : Spark SQL retourne souvent des collections mutable.ArraySeq, incompatibles avec certaines opérations immuables en Scala. Nous avons dû les convertir explicitement en Seq.
2. **Format des dates** : Les dates au format EEE MMM dd HH:mm:ss zzz yyyy n’étaient pas reconnues par Spark 3.0+. Nous avons configuré la politique de parsing en mode **LEGACY** :
3. spark.conf.set("spark.sql.legacy.timeParserPolicy", "LEGACY")
4. **Doublons dans les résultats** : Des noms identiques apparaissaient dans les résultats (par exemple, des mentions ou hashtags répétés). Une réduction a été appliquée pour supprimer les doublons après les calculs :
5. .reduceByKey(Math.max)

## **Améliorations possibles**

1. **Traitement des données en amont** :
   * Retirer les doublons directement lors de la création des sommets et des arêtes.
   * Ajouter un nettoyage NLP (suppression des stopwords, etc.).
2. **Utilisation de GraphFrames** : GraphFrames, une extension de Spark SQL, offre des performances supérieures et une syntaxe SQL-friendly pour les graphes.
3. **Ajout d’analyses avancées** :
   * **Shortest Paths** pour calculer les distances entre utilisateurs.
   * **Motifs dans les graphes** pour détecter des modèles spécifiques (e.g., triangles).

## **Conclusion**

Ce projet a permis de :

* Comprendre et manipuler les notions fondamentales de **Spark** et **GraphX**.
* Modéliser des relations implicites entre utilisateurs et hashtags.
* Utiliser des algorithmes avancés pour extraire des insights exploitables.

### **Annexes**

1. **Code source complet** : Voir fichier joint.
2. **ChatGPT**
3. **Liens utiles** :
   * [Apache Spark](https://spark.apache.org/docs/latest/)
   * [GraphX Documentation](https://spark.apache.org/graphx/)