Clustering de documentos basado en Big Data

Andrés Felipe García Granados - agarci45@eafit.edu.co
Cristian David Suaza Cárdenas - csuazaca@eafit.edu.co
Stiven Andrés Hurtado Loaiza - shurtad5@eafit.edu.co

EAFIT

Palabras clave— k-means, tf-idf, dataset, cluster, spark.mllib, python, big data, etl, paralelo

I. Resumen

En el presente documento se plantea una solución al Proyecto 4 de la materia Tópicos Especiales en Telemática. Éste consiste en implementar un algoritmo distribuído en Big Data que permita agrupar diferentes documentos, específicamente se hablará del algoritmo ***k-means***, y de ***tf-idf*** que es una medida que ayuda a clasificar documentos relacionados entre sí.

La implementación de este algoritmo distribuído se realizó en Python 3 con Spark 2.1.1

II. Introducción

El mundo de hoy se puede decir que es inconcebible sin la tecnología, pues gracias a dispositivos como los computadores, celulares, entre otros, podemos buscar todo tipo de información que a diario fluye por medio de internet. Es tanta la información que se transmite por este medio, que resulta necesario para los creadores de buscadores como Google, Firefox, Microsoft Edge, entre otros, utilizar algoritmos que agrupen dicha información y de esta manera poder mostrarle al usuario lo más relacionado posible con la búsqueda realizada por éste.

III. Marco teórico

Se cuenta con un conjunto de documentos desordenados, los cuales requieren ser organizados en un número de clusters dependiendo del grado de similitud entre estos documentos. El primer paso a realizar es determinar qué tan similares son cada uno de los documentos del dataset. Existen varios algoritmos que ayudan a determinar la similitud entre dos documentos, entre ellos están: *Distancia Euclidiana, Similitud Coseno, Coeficiente de Jaccard, Coeficiente de Correlación de Pearson, Divergencia de Kullback-Leibler,* ***tf-idf*** *(Term frequency – Inverse document frequency).*

Luego de determinar qué tan similares son los documentos, se debe utilizar un algoritmo de agrupamiento como *k-means* para clasificarlos en un número de k clusters.

IV. Análisis y Diseño mediante ETL

El diseño del algoritmo distribuído se utiliza a través de **ETL**, el cual es un proceso que consiste en Extracción, Transformación y Carga de los datos para analizarlos con un fin específico, en este caso, clasificar documentos en clusters.

La **Extracción** consiste en extraer los datos desde algún sistema origen, ya sea una ruta local, desde HDFS, o cualquier otro sitio válido que contenga un dataset.

En la **Transformación** se aplican una serie de funciones a los datos extraídos para convertirlos en datos que serán cargados.

En la **Carga** se almacenan los datos ya transformados en el sistema destino, por ejemplo, una base de datos o simplemente una ruta válida donde se puedan guardar estos datos.

V. Implementación

A continuación se muestra cada una de las librerías y funciones requeridas para llevar a cabo la implementación del algoritmo distribuído. **Cabe destacar que se hace uso del código proporcionado por Apache Spark en los ejemplos de kmeans y de tf-idf**:

- Se importan las siguientes librerías:
  - *SparkContext* - es el punto de entrada principal para utilizar las funcionalidades que ofrece Spark, es decir, sirve para establecer una conexión a un cluster Spark. Entre las funcionalidades están: crear RDDs, acumuladores y variables de difusión (broadcast) en ese cluster.
  - *randint* - módulo para generar números random.
  - *KMeans* - librería de machine learning que ofrece Spark para organizar en clusters un conjunto de datos.
  - *KMeansModel* - sirve para guardar y cargar el modelo de KMeans.
  - *HashingTF* - mapea un conjunto de términos y retorna su correspondiente frecuencia.
  - *IDF* - computa la frecuencia inversa de documento, es decir, la frecuencia de ocurrencia

del término en la colección de documentos.

- Se crea el Contexto Spark para utilizar las funciones que éste ofrece

```
sc = SparkContext(appName="kmeans")
```

- Se **extrae** el dataset y se almacena en la variable *docs*

```
docs = sc.wholeTextFiles("/home/pipe/
    datasets/gutenberg/*.txt")
```

- Se realiza una **transformación** al dataset para borrar todos los espacios en blanco y sólo trajabar con las palabras

```
words = docs.values().map(lambda line:
     line.split(" "))
```

- Se calcula *tf* con la fórmula:

$tf(t,d) = \frac{f(t,d)}{max\{f(w,d):w \epsilon d\}}$

lo cual quiere decir que se halla el número de veces que el término t ocurre en el documento d.

Algoritmo:

```
hashingTF = HashingTF()
tf = hashingTF.transform(words)
```

- Se calcula *idf* con la fórmula:

$idf(t,D) = \log \frac{|D|}{|\{d \epsilon D: t \epsilon d\}|}$

donde D es el número de documentos en la colección y el denominador representa el número de documentos donde aparece el término t.

Algoritmo:

```
idf = IDF().fit(tf)
tfidf = idf.transform(tf)
```

- Se halla un k aleatorio entre 3 y 6, el cual representa el número de clusters en que serán agrupados los documentos. Luego se pasan los parámetros necesarios para ejecutar la función de *kmeans*. El tfidf es el resultado obtenido de multiplicar tf por idf.

```
k = randint(3,6)

clusters = KMeans.train(tfidf, k,
    maxIterations=10,
    initializationMode="random")
```

- La variable **docsArray** guarda la ruta completa de cada uno de los documentos del dataset, **centroidsArray** almacena los diferentes centroides que son la base para clasificar los documentos.

```
docsArray = docs.keys().collect()
centroidsArray = clusters.predict(
    tfidf).collect()
```

- Se realiza un ciclo para mostrar cada uno de los clusters con sus respectivos documentos

```
for x in range(k):
    print("en el cluster " + str(x) +
        " estan:")
    for j in range(centroidsArray.
        __len__()):
        if centroidsArray[j]==x:
            print(docsArray[j])
    print("")
```

- Por último se realiza las operaciones de guardado y **carga** de los datos en la ruta especificada

```
clusters.save(sc, "target/org/apache/
    spark/PythonKMeansExample/
    KMeansModel")
sameModel = KMeansModel.load(sc, "
    target/org/apache/spark/
    PythonKMeansExample/KMeansModel")
```

VI. Implementación en Cluster Spark en Producción

Se clona el repositorio y se cambia la ruta desde donde se va a extraer el dataset

```
docs = sc.wholeTextFiles("hdfs:///
    datasets/gutenberg-txt-es/*.txt")
```

Luego, se ejecuta con la instrucción:

spark-submit –master yarn –deploy-mode cluster –executor-memory 2G –num-executors 4 kmeans.py

VII. Análisis de resultados

El siguiente cuadro muestra el resultado obtenido al ejecutar el algoritmo de *k-means* tanto en HPC como en Big Data. Cada una de las columnas se interpreta asi:

- Dataset - Conjunto de documentos utilizados
- Docs - Número de documentos en el dataset
- Paralelo - Tiempo de ejecución (número entero) en segundos utilizando MPI. (HPC)
- Distribuído - Tiempo de ejecución (número entero) en segundos utilizando SparkML. (Big Data)

Dataset	Docs	Paralelo(s)	Distribuído(s)
Gutenberg	100	1091	–
gutenberg-txt-es	461	–	180

* La ejecución del algotirmo paralelo se realizó con 7 cores y un k aleatorio entre 8 y 12.

VIII. Conclusiones

- Se logra tener un acercamiento a los modelos de programación en Big Data, y se logra comprender la ventaja de utilizar Spark por encima de MapReduce, ya que al ejecutar el mismo algoritmo en ambos motores, el primero se comporta mejor dando como resultado una ejecución mucho más rápida.

- Comparando HPC con Big Data se observan limitaciones principalmente sobre hardware, ya que si se desea escalar respecto a ello, resulta más costoso, es decir, se necesitan más computadores, mientras que con Big Data no necesariamente ocurre ésto, ya que con un sólo computador con mejores especificaciones de memoria, almacenamiento, etc, se podría ejecutar de manera óptima el algoritmo.

- Se logra comprender la ventaja de utilizar el proceso de ETL durante el análisis y diseño de algoritmos distribuídos, puesto que observamos que este método es de gran importancia para mejorar el rendimiento en general cuando se trabajan con grandes volúmenes de datos, es decir, ésto ayuda a devidir un archivo secuencial en pequeños archivos de datos para así proporcionar acceso paralelo.

- Se pudo observar la mejora de un algoritmo distribuído frente a uno paralelo, pues el tiempo de ejecución se redujo de manera drástica.

Referencias

[1] Anna Huang. *Similarity Measures for Text Document Clustering.* Department of Computer Science. The University of Waikato, Hamilton, New Zealand.

[2] Clustering - RDD-based API - Spark 2.1.1 Documentation. *K-means* Noviembre 17, 2017
https://spark.apache.org/docs/2.1.1/mllib-clustering.html

[3] Apache Spark. *spark/k_means_example.py at master apache/spark.* Noviembre 17, 2017
https://github.com/apache/spark/blob/master/examples/src/main/python/mllib/k_means_example.py

[4] Apache Spark. *spark/tf_idf_example.py at master apache/spark.* Noviembre 17, 2017
https://github.com/apache/spark/blob/master/examples/src/main/python/mllib/tf_idf_example.py

[5] Mike Bernico *Using TF-IDF to convert unstructured text to useful features* Febrero 1, 2015
https://www.youtube.com/watch?v=hXNbFNCgPfY

[6] misssushiyan *super power - doing k-means clustering using Spark ML library* Diciembre 5, 2015
https://www.youtube.com/watch?v=Lm1c2U8BmoA

[7] Extract, transform and load - Wikipedia, la enciclopedia libre. *Extract, transform and load* Octubre 19, 2017
https://es.wikipedia.org/wiki/Extract,_transform_and_load

[8] Tf-idf - Wikipedia, la enciclopedia libre. *Tf-idf* Agosto 27, 2017
https://es.wikipedia.org/wiki/Tf-idf