# Clústering de Documentos a partir de Métricas de Similitud

**Jose D. Sánchez Castrillón - Mayerli A. López Galeano**

**jsanch81@eafit.edu.co - mlopez12@eafit.edu.co**

**Universidad Eafit**

22 de noviembre de 2017

## Resumen

La minería de texto es una de las mejores técnicas de búsqueda en análisis de escritos, textos o documentos. Hoy en día es utilizado en los grandes buscadores web, sistemas de recomendaciones, entre otros. El principal problema a resolver es brindar recomendaciones de libros que tiendan a poseer el mismo contenido, o hablar de un mismo tema. Para realizar esto se utiliza la técnica de agrupamiento de clúster, que permite relacionar un documento con otro parecido, donde es necesario utilizar algoritmos de agrupamiento (kmeans), y a su vez algoritmos de distancias para obtener el parecido de un documento con el otro (jaccard). En nuestro caso utilizamos el Kmeans como algoritmo de agrupamiento, el cual estaba incluido en una biblioteca de python, para la distancia entre los documentos se implementaron algoritmos como el HashingTF y el IDF, los cuales devuelven la matriz que es analizada por el kmeans. Por último, hay que tener en cuenta que los documentos que se analizan se encuentran en un Cluster Hadoop, y para poder acceder a ellos es necesario utilizar el framework de ejecución y procesamiento distribuido basado en Spark.[1]

## Palabras claves

Clustering, text minig, dataset, k-means, Big Data, hadoop, spark, IDF, HashingTF, HPC, paralelo, jaccard.

## 1. Introducción

El siguiente informe muestra la forma de abordar la problemática de búsqueda de documentos a partir de la técnica de análisis (minería de textos) que utiliza una implementación con las tecnologías de BigData.

El algoritmo final fue implementado con una estructura específica que radica en leer los datasets desde el servidor de hadoop, buscar la distancia de los documentos y dividir en subgrupos, con el fin de dar solución a la problemática, y además poder realizar un comparativo de los tiempos de procesamiento con el algoritmo en paralelo.

## 2. Marco teórico

La minería de texto (text mining), es una de las técnicas de análisis de textos que ha permitido implementar una serie de aplicaciones muy novedosas hoy en día. Buscadores en la web (Google, Facebook, Amazon, Spotify, Netflix, entre otros), sistemas de recomendación, procesamiento natural del lenguaje, son algunas de las aplicaciones. Las técnicas de agrupamiento de documentos (clustering) permiten relacionar un documento con otros parecidos de acuerdo a alguna métrica de similaridad. Esto es muy usado en diferentes aplica- ciones como: Clasificación de nuevos documentos entrantes al dataset, búsqueda y recuperación de documentos, ya que cuando se encuentra un documento seleccionado de acuerdo al criterio de búsque- da, el contar con un grupo de documentos relacionados, permite ofrecerle al usuario otros documentos que potencialmente son de interés para él.[2]

## 3. Análisis y diseño

En la implementación del código se realizaron diferentes etapas, la primera de ellas fue el acceso de los documentos, los cuales se encuentran en el servidor de hadoop, y para acceder a ellos era necesario utilizar spark, por lo cual fue necesario realizar una investigación de spark con python, donde nos basamos principalmente en los métodos utilizados en el laboratorio de spark, y utilizamos el método $wholeTextFiles$, el cual se encarga de generar un diccionario, donde su clave es el nombre del documento analizado, y su valor es el texto del archivo, pero inicialmente para poder utilizar estos métodos era necesario iniciar un contexto de spark, el cual se generaba con la función. $SparkContext$.

Ya con los datos obtenidos del servidor, la siguiente etapa básicamente se encarga de realizar un método split con espacios a los valores del diccionario obtenido por el método $wholeTextFiles$, básicamente este split está dividiendo el texto del archivo en un arreglo de palabras, las cuales se cuentan más adelante para poder obtener la frecuencia de cada palabra, y poder comparar con los demás documentos.

En la siguiente etapa empezamos a buscar métodos de similaridad, donde encontramos los métodos HashingTF y el IDF, En primera parte el método HashingTF lo que hace es trasformar conjuntos de términos en vectores de longitud fija, básicamente, este método me devuelve la frecuencia de las palabras de cada documento.

Luego se utilizael método IDF, este es un estimador que se ajusta a un conjunto de datos y produce un modelo IDF. El IDFModel toma vectores de características (generalmente creados a partir de HashingTF o CountVectorizer) y escala cada columna. Intuitivamente, baja las columnas que aparecen con frecuencia en un conjunto de datos de textos. Este método básicamente me devuelve la similaridad de cada documento comparado con los demás, estos valores me los entrega en una matriz, y utiliza las siguientes formulas para sacar estos valores donde D es el número total de documentos.[3]

**Figura 1.** IDF

$$IDF(t, D) = \log \frac{|D| + 1}{DF(t, D) + 1},$$

FUENTE: https://spark.apache.org/docs/2.2.0/ml-features.html

Ya el algoritmo compara los términos de los documentos, y si aparece un término en todos los documentos, su valor IDF se convierte en 0. Se debe de tener en cuenta que se aplica un término de suavizado para evitar dividir por cero los términos fuera del conjunto de textos. La medida TF-IDF es simplemente el producto de TF e IDF.[3]

**Figura 2.** TFIDF

$$TFIDF(t, d, D) = TF(t, d) \cdot IDF(t, D).$$

FUENTE: https://spark.apache.org/docs/2.2.0/ml-features.html

Después de realizar la etapa anterior, procedimos a utilizar el algoritmo kmeans, el cual se encarga de asignar a cada documento a un cluster, dependiendo en principio de su cercanía con los centroides iniciales, los cuales son aleatorios en el comienzo.

Después de esto el kmeans procede a realizar más iteraciones, pero en cada iteración calcula nuevos centroides dependiendo de los valores de las distocias de los documentos que pertenezcan a ese cluster. Con estos nuevos centroides el algoritmo procede a realizar la reagrupación de los documentos.

El kmeans utilizado proviene de una biblioteca de Python, este me devuelve un objeto de tipo $KMeansModel$, el cual es necesario aplicarle un método de esta clase llamado *predict*, y que recibe como parámetros el mapa $TFIDF$. Al aplicar este método al objeto obtenido por el kmeans, obtendremos un arreglo el cual tendrá a que grupo o cluster pertenece cada documento. Ya al tener esta solución en principio decidimos mostrarla en la terminal para verificar los datos, luego se guardaron los datos de salida en el cluster de hadoop con el método $saveAsTextFile$, el cual guarda archivos del tipo $RDD$, por lo cual fue necesario transformar el arreglo de salida en un objeto de tipo $RDD$ con el método *parallelize*.

## 4. Implementación

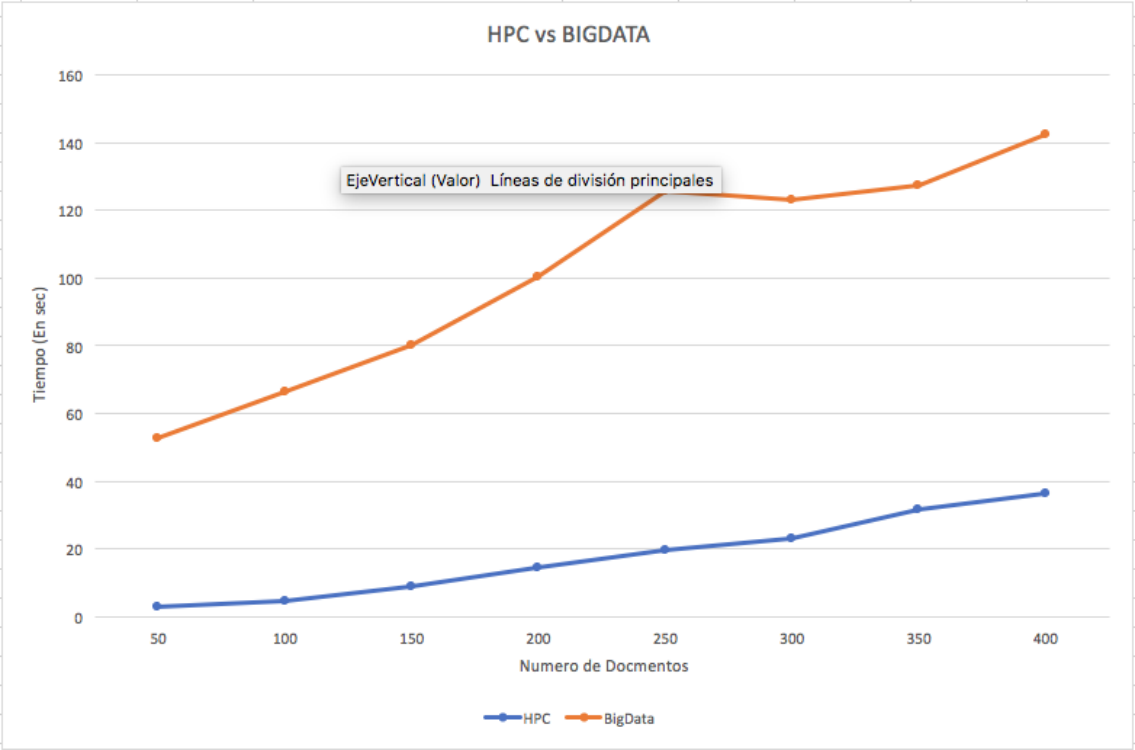
La implementación se inició en primera parte obteniendo los archivos del cluster de hadoop con el método $wholeTextFiles$, luego se procedio a realizar la separación por espacio para obtener las palabras, luego se procedió a sacar la frecuencia de cada palabra en cada documento con el método hashingTF, y de ahí con el método IDF procedemos a tener la distancia de cada documento en comparación con los demás. Ya al obtener los resultados, se envía estos datos al kmeans para su agrupamiento, y al final se realiza el almacenado en el cluster de hadoop.

## 5. Resultados

Las pruebas se realizaron en el DCA que contaba con spark, y se ejecutaron pruebas con una cantidad inicial de 50 documentos, hasta una cantidad de 400 documentos como máximo, en la siguiente tabla se nota que los tiempos arrojados por la programación en paralelo son mucho más rápidos que los tiempos arrojados con la implementación de spark, sin embargo, estos últimos son mejores que los tiempos que se obtienen con la programación serial y no están muy lejos en comparación con HPC.

4

**Figura 3.** Grafica de comparación




FUENTE: Jose David

**Tabla 1.** Tabla de tiempos

| Número de datos | Tiempo Paralelo (seg) | Tiempo BigData (seg) |
|---|---|---|
| 50 | 2,900 155 067 | 50 |
| 100 | 4,588 327 885 | 62 |
| 150 | 9,021 161 079 | 71 |
| 200 | 14,322 097 06 | 86 |
| 250 | 19,471 766 95 | 106 |
| 300 | 23,146 032 09 | 100 |
| 350 | 31,451 109 89 | 96 |
| 400 | 36,209 965 23 | 106 |

Como se logra apreciar en la tabla anterior, los tiempos de HPC vs. BigData, no son muy alejados y hay momentos en que los tiempos de BigData son muy poco cambiantes. Además de esto hay que tener en cuenta que los tiempos no son muy precisos, debido a que las pruebas se realizaron en el momento en que otras personas también estaban realizando sus pruebas, lo cual pudo bajar el rendimiento del DCA, pero en sí, la minería de datos que es trabajada con BigData se considera una gran opción no solo por los tiempos, sino también por la facilidad de la implementación en comparación con la programación en paralelo.

## 6. Conclusiones

1. A partir de los resultados obtenidos, la gráfica y la manera en que se aborda el problema se concluye que la implementación en paralelo logró ser más eficiente que la implementación con spark.

2. Al comparar las implementaciones, se determinó que la programación con spark es mucho más sencilla de aplicar en comparación con el desarrollo en paralelo.

3. En el proceso se observó que la implementación con spark se encuentra entre la programación serial y la programación en paralelo.

## 7. Referencias


1. spark, A. TD IDF., https://spark.apache.org/docs/2.2.0/ml-features.html.
2. Montoya-Múnera Edwin N. Pineda-Cárdenas, J. D. Tópicos Especiales en Telemática: Proyecto 4 – Clústering de Documentos a partir de Métricas de Similitud basado en Big Data - v1.0., 2017, pág. 1.
3. maximerihouey TD-IDF., https://github.com/apache/spark/blob/master/examples/src/main/python/mllib/tf_idf_example.py, 2016.
4. Huang, A. en *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, 2008, págs. 49-56.