**Log clustering tool**

Imagen que contiene imágenes prediseñadas

Descripción generada con confianza muy alta

Trabajo de Fin de Máster para acceder al

MÁSTER EN CIENCIA DE DATOS

Autor: Javier Villar Raba

Directora: Cristina Tirnauca

Co-Director: Raúl Uría

Septiembre 2018

# Resumen

En el presente trabajo de fin de máster, se realiza un análisis completo de dos algoritmos que segmentan líneas de log: Simple Logfile Clustering Tool (SLCT) y LogCluster. Para ello, se detallan los principios básicos de su funcionamiento y se estudia cómo afecta implementar estos algoritmos con tecnología big data como es apache spark.

Se estudia como estos algoritmos dependen del usuario y como es importante tener un mínimo conocimiento de los logs a analizar. Se lleva a cabo un estudio de como varían los resultados en función de los datos introducido por el usuario y como cambian los clústeres obtenidos.

Por último, se obtienen modelos que clasifican nuevas líneas en los clusteres obtenidos o bien en outlier con lo que quiere esto decir que esa línea puede venir porque se está produciendo un error.

**Contenido**

[**Resumen i**](#_Toc524443238)

[**1. Introducción 1**](#_Toc524443239)

[**2. Herramientas y metodología. 2**](#_Toc524443240)

[2.1 Python. 2](#_Toc524443241)

[2.2 Elasticsearch. 3](#_Toc524443242)

[2.3 Logstash. 4](#_Toc524443243)

[2.4 Kibana. 5](#_Toc524443244)

[2.5 Apache Spark. 6](#_Toc524443245)

[**2.5.1 Transformaciones y acciones utilizadas. 7**](#_Toc524443246)

[2.6 Metodología. 9](#_Toc524443247)

[**3. Algoritmos. 9**](#_Toc524443248)

[3.1 Algoritmo Simple Logfile Clustering Tool (SLCT). 10](#_Toc524443249)

[3.2 Algoritmo LogCluster. 12](#_Toc524443250)

[**4. Implementación. 14**](#_Toc524443251)

[4.1 Implementación del módulo para el Algoritmo LogCluster secuencial. 15](#_Toc524443252)

[4.2 Implementación del Algoritmo LogCluster usando Apache Spark. 19](#_Toc524443253)

[4.3 Implementación del Algoritmo SLCT usando Apache Spark. 22](#_Toc524443254)

[4.4 Obtención de los modelos. 23](#_Toc524443255)

[4.5 Primeros pasos. 24](#_Toc524443256)

[**5. Ejemplos de uso. 27**](#_Toc524443257)

[5.1 Fichero de logs de un sistema operativo. 27](#_Toc524443258)

[5.2 Fichero de logs de envió de datos de plataformas. 31](#_Toc524443259)

[**6. Conclusiones. 39**](#_Toc524443260)

[**Anexo I. 39**](#_Toc524443261)

[**Anexo II. 41**](#_Toc524443262)

[**7. Bibliografía 44**](#_Toc524443263)

# **1. Introducción**

Hoy en día se genera una gran cantidad de archivos de logs en campos como la industria y la ciencia, ya que éstos son una fuente de información muy importante y útil en muchas situaciones. Sin embargo, a medida que aumenta la complejidad de los sistemas, el análisis de los archivos de logs es cada vez más exigente y dificultoso, ya que hay que realizar un gran esfuerzo para recopilar, almacenar e indexar una gran cantidad de logs que se agrava más cuando éstos no son heterogéneos[1].

Toda esta producción de logs hace que, debido al tamaño de estos conjuntos de datos, las soluciones que devuelven las bases de datos convencionales no sean las adecuadas para el análisis de la información y en su lugar se consideren más apropiados bases de datos virtuales (que no son más que contenedores que integran datos de múltiples fuentes de datos, de modo que se puede acceder a ellas de una manera integrada a través de una API) combinadas con sistemas de procesamiento distribuidos y paralelo.

Con todo esto, se puede suponer que el análisis de logs es un caso de uso de big data y por lo tanto es un gran desafío como procesar, almacenar o gestionar estos logs con los recursos disponibles. Además, hay que añadir que cuando los logs provienen de múltiples fuentes surgen problemas con la extracción de contenido significativo y su correlación. Por todo esto, hacen falta varias soluciones eficientes para tratar, reconocer y almacenar la información importante y que se puedan recuperar o migrar fácilmente entre los diferentes centros de datos.

En definitiva, el tratamiento de logs hoy en día tiene una gran importancia y por esta razón se han desarrollado muchos algoritmos para ello. Esto implica que tienen que existir computadoras muy potentes para dicho tratamiento, ya que una máquina genera muchos registros de logs.

Con la monitorización de los archivos de logs se pueden detectar errores y/o anomalías en el funcionamiento de la máquina que genera los logs. Cuando se manifiesta un error, tiene que haber un experto que haya almacenado dicho error en una base de datos o algo similar, sin embargo, si el error que se ha generado no está catalogado no puede ser detectado, pero llevar todo este trabajo a cabo lleva mucho tiempo y esfuerzo y además es propenso a que se cometan errores [2].

El objetivo de este trabajo es analizar logs a través de varios algoritmos de segmentación (que agrupa los logs en grupos y según sus patrones). Para llevar a cabo estas tareas se trabajará con una base de datos no relacional (elasticsearch), con una herramienta de extracción, transformación y carga (logstash), con un framework de computación en clúster (apache spark) y con el lenguaje de programación Python y sus librerías. Todas estas herramientas son open-source y están indicadas para trabajar con gran cantidad de datos.

# **2. Herramientas y metodología.**

Para la realización de este trabajo se van a utilizar las herramientas que se describen a continuación.

## **2.1 Python.**

Python [3] es un lenguaje de programación que surgió en 1991 con la idea de que su sintaxis haga que el código sea legible más fácilmente y que en la actualidad tiene dos versiones estables que son la 3.7 y la 2.7 (que es la que se utiliza en este trabajo). La razón por la que aún se está dando soporte a la versión 2.7 es porque de esta versión se pasa a la 3.0 y hay un gran cambio cuando uno se pone a desarrollar su código o quiere realizar una migración.

Python es un lenguaje interpretado (no hay que compilar el código antes de su ejecución) y multiplataforma por lo que se puede usar en varios sistemas operativos distintos como pueden ser Windows, Ubuntu o Mac. Además, se pueden crear todo tipo de programas ya que no está diseñado para un único propósito. Asimismo, soporta la programación orientada a objetos y en muchos casos ofrece una manera sencilla de crear programas con componentes reutilizables.

Por último, dispone de muchas funciones incorporadas en el propio lenguaje y además existen muchas librerías que podemos importar en los programas para tratar temas específicos como es la búsqueda de patrones usando expresiones regulares o hacer gráficas de los datos obtenidos o disponibles. Las tres librerías principales que se utilizarán en este proyecto son:

* Matplotlib: librería que genera figuras de calidad en una gran variedad de formatos a partir de datos contenidos en listas o arrays.
* Re: librería que permite verificar si una expresión regular dada coincide con una cadena en particular.
* Pyspark: librería que permite utilizar todas las funciones de apache spark utilizando el lenguaje Python.

Por último, cabe señalar que para desarrollar todo el código se hizo uso de un entorno de desarrollo integrado, en inglés Integrated Development Environment (IDE), que es una aplicación informática que tiene muchas funcionalidades y servicios que facilitan el desarrollo del programa a realizar. Para este proyecto se hizo uso de dos IDE diferentes que fueron.

* Microsoft Visual Studio.
* Wing Python IDE.

## **2.2 Elasticsearch.**

Elasticsearch [4] es una herramienta open-source desarrollada por la compañía Elastic que nos permite indexar una gran cantidad de datos para, posteriormente, realizar consultas sobre ellos, ya sea realizando búsquedas aproximadas o un texto completo, ya que al estar la información almacenada indexada los resultados se obtienen de forma rápida. Elasticsearch funciona mediante una interfaz REST recibiendo y enviando datos en formato JSON y permite que pueda ser usada por varias plataformas como pueden ser Java, Python, .Net o un navegador con JavaScript. Además, la información que se almacena es persistente.

A continuación, se muestra un ejemplo de cómo se puede añadir un log y visualizar la información que se ha añadido en elasticsearch. Para ver este ejemplo se hará uso del siguiente log:

*Jul 26 09:36:29 RUE3 anacron[1102]: Job `cron.daily' terminated*

Para insertar este log en elasticsearch basta con usar un método post y éste se insertará de forma directa. Si ahora se hace una consulta a esta base de datos se puede ver como permanece almacenada la información. Esta consulta se puede realizar de varias formas diferentes y para este caso se ha usado el navegador web para conectarse a elasticsearch con la siguiente url [*http://localhost:9200/trabajo\_master/\_search/?pretty*](http://localhost:9200/trabajo_master/_search/?pretty). El resultado obtenido se muestra en la figura 1.

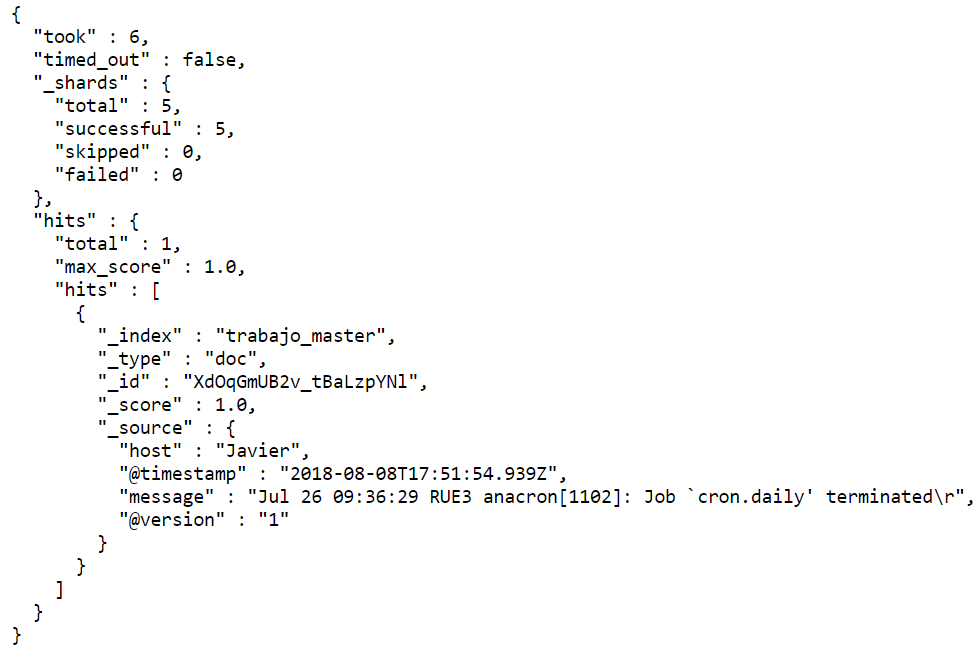


Figura 1. Resultado de como se almacenó el log anterior en elasticsearch.

Como puede apreciarse en la figura 1 tras insertar el log en elasticsearch se ha creado de forma automática un archivo JSON que indexa y almacena la información. Además, se puede ver como añade de forma automática varios campos como son: *@timestamp* en el que almacena la hora en la que se ha insertado el evento y *@version* que almacena la versión del documento. Por último, hay que destacar que también se añade un campo *\_id* que es un identificador único para cada documento.

## **2.3 Logstash.**

Logstash [5] es una herramienta open-source desarrollada por la compañía Elastic que permite extraer, transformar y cargar la información en elasticsearch. Para poder realizar todo esto soporta varias entradas, códecs, filtros y salidas. La fuente donde se encuentra la información de datos es la entrada, los códecs sirven para cambiar formatos de entrada y/o salida. Para transformar la información se utilizan los filtros, de esta forma se procesan los eventos. Finalmente, las salidas son los destinos a los cuales se quieren enviar los datos tras ser procesados.

Vamos a ver un ejemplo del uso de logstash con el log definido anteriormente, en el cual puede apreciarse de una forma evidente que empieza por un mes seguido del día y de la hora. A la vista de esto, se deduce que se puede obtener un campo extra en el que se almacene la fecha y hora en la que se generó el log (antes de insertarlo en elasticsearch). Para su realización se hace uso de las funcionalidades disponibles y se escribe la secuencia tal como se muestra en la figura 2.

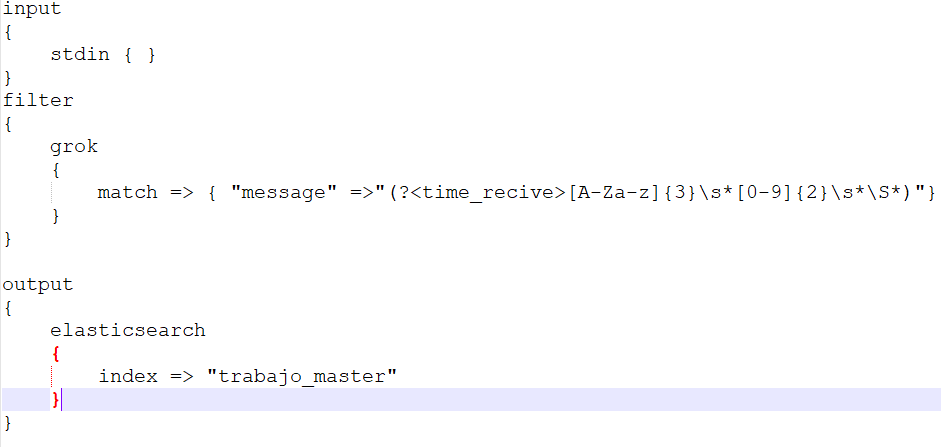


Figura 2. Código usado para transformar y obtener un campo extra del log de ejemplo.

En la figura 2 se observa como la entrada del log es por pantalla y a éste se le aplica un filtro que almacena en la variable *time\_recive* una fecha y hora tras hacer uso de una expresión regular. Por último, se almacena toda la información en elasticsearch con el índice trabajo\_master.

El documento que se ha insertado tendrá el mismo formato que el de la [figura 1](#Figura1) salvo porque la variable *\_id* ha cambiado (recordemos que es un identificador único) y porque se ha añadido un nuevo campo *time\_recive* que contiene la información de la fecha del mensaje original tras haber parseado la información usando logstash. Esto se puede apreciar en la [figura 3](#Figura03).

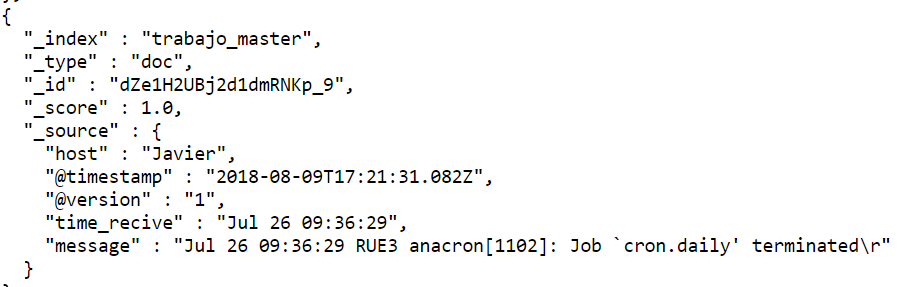


Figura 3. Resultado de como se almacenó el log en elasticsearch tras usar logstash.

## **2.4 Kibana.**

Kibana [6] es una herramienta open-source desarrollada por la compañía Elastic que permite realizar exploraciones visuales y análisis en tiempo real de los datos almacenados en Elasticsearch, ya que se pueden diseñar visualizaciones y dashboards.

A continuación, en la [figura 4](#Figura04), vemos un ejemplo en el que en un dashboards se muestran dos visualizaciones. En la primera de ellas, se puede ver un histograma que muestra cuando se han ido recibiendo los logs en el tiempo y en la segunda visualización se muestra el log.

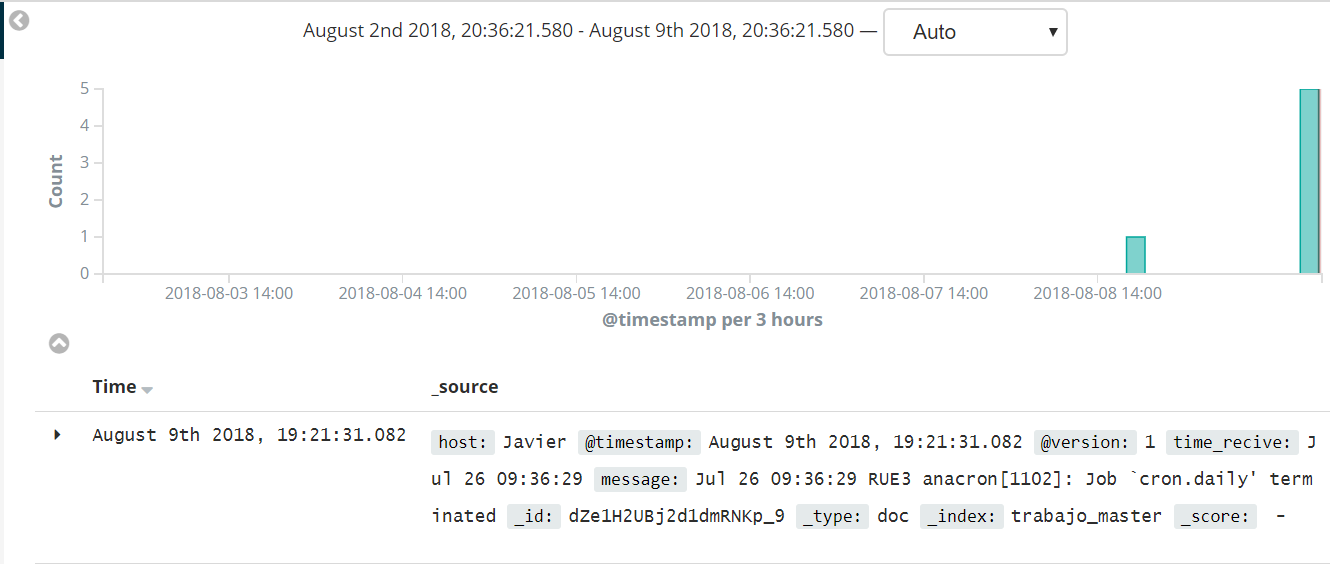


Figura 4. Dashboard que contiene dos visualizaciones sobre los logs.

## **2.5 Apache Spark.**

Apache Spark [7] es una infraestructura informática de código abierto usado para trabajar con gran volumen de datos ya que gestiona el uso de éstos en memoria. Surgió en el año 2009 dentro de un proyecto de investigación en la Universidad de Berkeley y que en el año 2013 fue donado a la fundación Apache Software Foundation. En agosto de 2018 se encuentra por la versión 2.3.1 (en este proyecto se usa la versión 2.3.0) .

Las aplicaciones que usan Spark son realizadas de forma independiente y son coordinadas por el objeto SparkContext que se encuentra en el programa principal. Además, es capaz de conectarse a gestores de clúster que se encargaran de asignar los recursos que hay en el sistema para el mejor funcionamiento de la aplicación.

Para trabajar con Spark hay que conocer el concepto básico de Resilient Distribut Dataset (RDD) que son grupos de datos de lectura que están cargados en memoria (y que se pueden dividir para ser tratados de forma paralela) para realizar dos tipos diferentes de operaciones: acciones y/o transformaciones.

* **Acciones**: transmite el valor de un RDD a la aplicación, la función **count()** es un ejemplo de acción sobre un RDD ya que cuenta los elementos que posee el mismo, también la función **take(n)** devuelve un array con los primeros **n** elementos del RDD. Por último, está la función **collect()** que también devuelve en un array todos los elementos de un RDD.
* **Transformaciones**: radica en obtener un nuevo RDD tras modificar el original. Se pueden definir dos tipos diferentes de operaciones de transformación ya que lo más probable es que los datos se encuentren en más de un RDD.
  + **Narrow:** este tipo de operación se utiliza cuando los datos que se quieren tratar están en la misma distribución del RDD y no hace falta mezclarlos entre ellos. Algunos ejemplos son **map(func)** que crea un nuevo RDD a partir de otro aplicando una transformación a cada elemento original o **filter(func)** que crea un RDD nuevo manteniendo solo los elementos del RDD original que cumplen una determinada condición.
  + **Wide:** este tipo de operación se utiliza cuando los datos a tratar están situados en diferentes particiones de un RDD y es necesario que se mezclen estas particiones. Algunos ejemplos son **groupByKey()** que agrupa los RDD o **reduceByKey()** que los reduce.

De esta forma, se pueden realizar operaciones de gran cantidad de datos de forma rápida y flexible a los fallos, además Spark cuenta con una API que permite realizar conexiones con repositorios de datos como Elasticsearch, Hadoop, Cassandra, SQL y también ser usado con otros lenguajes de programación como Python, R o Java.

### **2.5.1 Transformaciones y acciones utilizadas.**

Spark utiliza un mecanismo de “evaluación perezosa” esto quiere decir que no se ejecuta una transformación en un RDD hasta que no se realiza alguna acción sobre el mismo.

A continuación, se muestran las operaciones de transformaciones y acciones utilizadas en este trabajo.

* **Transformación map()**: aplica una transformación a cada elemento del RDD original, de forma que convierte un RDD de tamaño *n* en otro RDD también de tamaño *n*. A continuación, se muestra un ejemplo del funcionamiento.

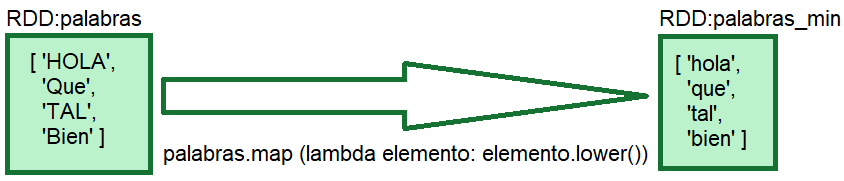


Figura 5. Ejemplo de funcionamiento de la transformación map(func) en un RDD primario.

* **Transformación flatMap():** aplica una transformación a cada elemento del RDD original pero cada elemento puede crear cero o más elementos. A continuación, se muestra un ejemplo en el que se le aplica la función split (que devuelve una lista con las palabras de una cadena) a cada elemento del RDD.

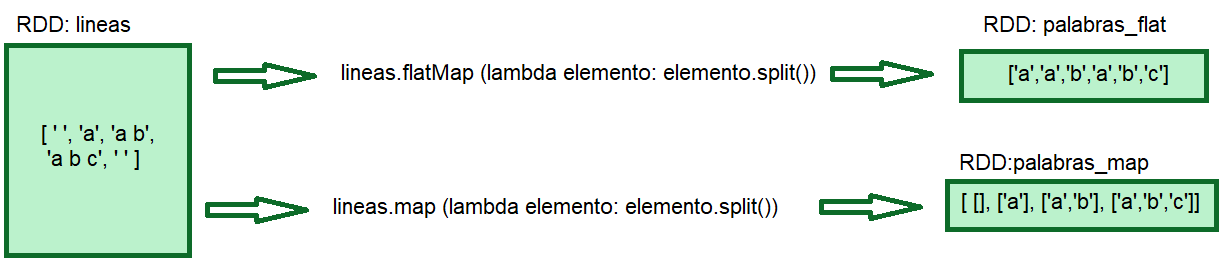


Figura 6. Ejemplo de funcionamiento de la transformación flatMap(func) y map(fun) en un RDD para comparar el funcionamiento de ambas.

* **Transformación filter():** filtra un RDD manteniendo solo los elementos que cumplen una condición.
* **Transformación union():** une dos RDD en uno solo. A continuación, se muestra un ejemplo en el que se hace uso de los métodos filter y union para transformar un RDD inicial.

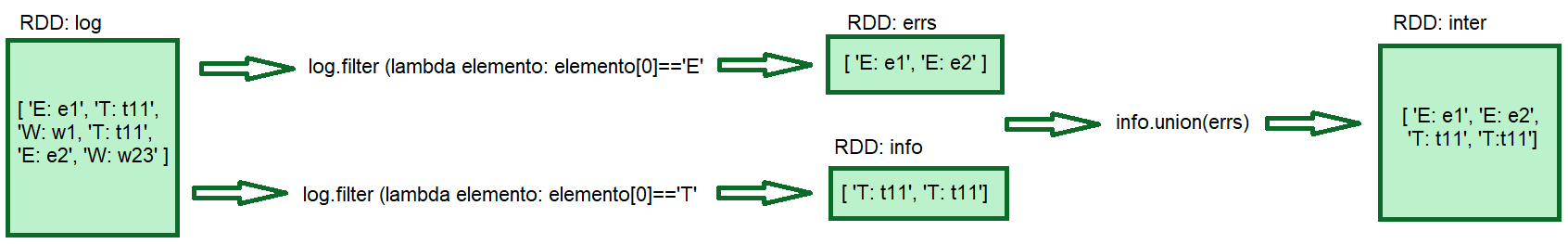


Figura 7. Ejemplo de funcionamiento de los métodos filter y union para transformar un RDD.

* **Transformación reduceByKey()**: Agrega todos los elementos del RDD hasta obtener un único valor por clave. A continuación, se muestra un ejemplo en el que se utiliza esta función para contar palabras.

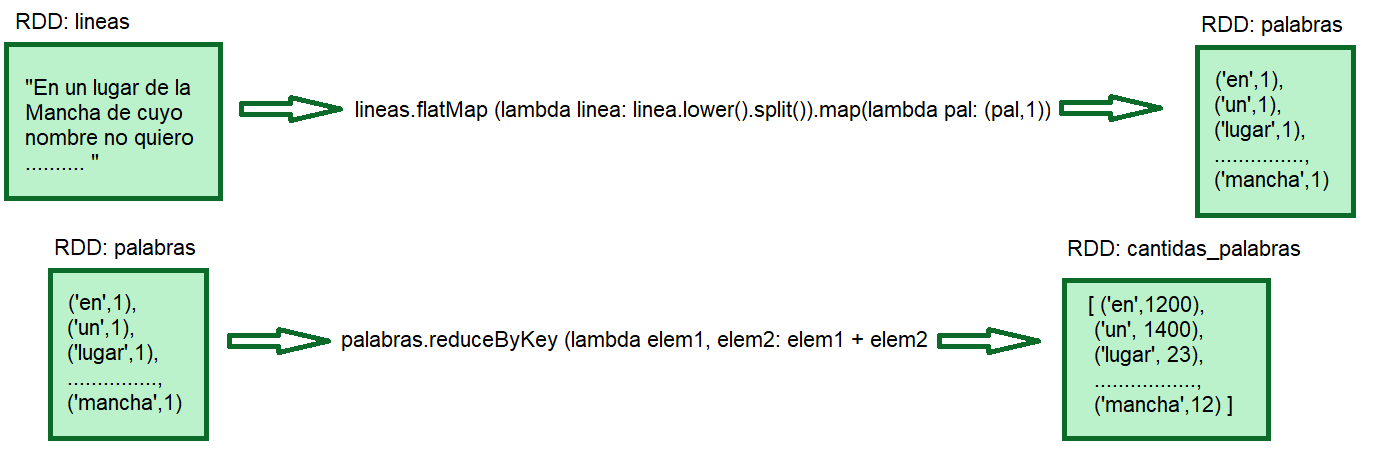


Figura 8. Ejemplo uso del método reduceByKey() para contar palabras.

* **Acción count()**: devuelve el número de elementos de un RDD.

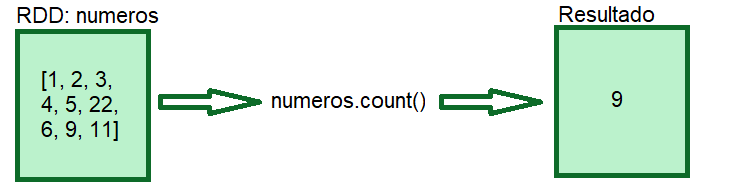


Figura 9. Ejemplo del método count() sobre un RDD.

* **Acción take(n)**: devuelve una lista con los primeros n elementos del RDD.

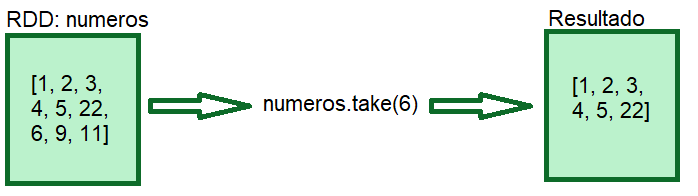


Figura 10. Ejemplo de aplicar la acción take(6) sobre un RDD.

* **Accion collect()**: devuelve en una lista todos los elementos del RDD.

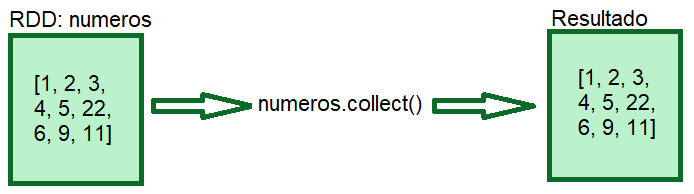


Figura 11. Ejemplo de aplicar la acción collect() sobre un RDD.

Para finalizar hay que destacar una de las características más importantes de Apache Spark: la persistencia o cacheo de un dataset en memoria. Esto quiere decir que cuando se persiste un RDD cada nodo almacena en memoria todas las particiones que posee para poder reutilizarlas al ejecutar otras acciones en dicho dataset, de esta forma las futuras acciones que se ejecuten serán más rápidas (hasta 10 veces más). Para poder convertir un RDD en persistente, hay que usar los métodos persist() o cache() y gracias a que Spark es tolerante a fallos si se pierde alguna partición de un RDD, ésta se recalcula automáticamente utilizando las transformaciones que lo crearon originalmente.

## **2.6 Metodología.**

Este proyecto, como ya se ha comentado anteriormente, trata de intentar segmentar los logs que generan las aplicaciones o sistemas operativos en diferentes clústeres para que de esta forma sea más fácil de identificar los errores y los diferentes tipos de logs que se generan. Además, con los clústeres se puede ver la estructura que tienen los logs.

Para comenzar, se implementan dos algoritmos diferentes de clustering que se aplicaran a varios registros de logs diferentes para, de esta forma comprobar los resultados obtenidos en función de cada algoritmo. Además, un algoritmo se implementará de dos formas diferentes, una primera forma utilizando el lenguaje Python y Apache Spark y una segunda forma utilizando solamente el lenguaje Python, mientras el otro algoritmo solo se implementará utilizando Python y Apache Spark. De esta forma conseguimos comparar el tiempo de ejecución entre los dos algoritmos y entre dos implementaciones diferentes para el mismo algoritmo. Además, se obtendrá para cada algoritmo el número de clústeres o el porcentaje de líneas totales que son segmentadas entre otras.

Para finalizar, se determinarán los modelos de cada clúster que en este caso serán expresiones regulares de cada clúster para que cuando venga un nuevo log clasificar este log en un clúster de nuestro modelo o bien como outlier en caso de que no cumpla ninguna expresión regular.

# **3. Algoritmos.**

Durante los últimos años, varios algoritmos han sido desarrollados para agrupar archivos de logs como CLIQUE, CACTUS Y MAFIA [8] pero en este proyecto nos centraremos en dos algoritmos del mismo autor que son **LogCluster** y Simple Logfile Clustering Tool también llamado **SLCT**.

Para estos dos algoritmos hay que explicar el concepto de soporte, *s*, ya que sin este concepto cada línea del fichero formaría un clúster por sí misma, por lo que podemos definir el soporte como el mínimo número de líneas que cumplen un cierto formato para que este clúster sea considerado patrón.

Por otra parte, hay que decir que estos dos algoritmos necesitan un diccionario de palabras frecuentes que o bien se pueden pasar como parámetros o, como se hará en este proyecto, se pueden contar las palabras que hay y aplicar un soporte que en este caso sería el mínimo número de cada palabra que debe aparecer en el archivo.

Sea un registro de eventos textuales que consiste en líneas y cada línea es una representación completa de algún evento e es un identificador único de cada línea. Suponiendo que cada línea es una secuencia de palabras de forma que . LogCluster y SLCT toman el umbral de soporte como un parámetro de entrada dado por el usuario y divide las líneas de registro de los eventos en grupos o clústeres de forma que hay al menos líneas en cada cluster y el resto de líneas serán consideradas outliers con lo que se puede escribir el registro de eventos como, .

A continuación, se pasa a explicar los dos algoritmos implementados para la realización de este trabajo.

## **3.1 Algoritmo Simple Logfile Clustering Tool (SLCT).**

El algoritmo Simple LogFile Clustering Tool (SLCT) [9] fue desarrollado por Risto Vaarandi en el año 2003 y fue uno de los primeros algoritmos diseñado para la búsqueda de patrones en logs. Este algoritmo tiene dos parámetros de entrada, que son el nombre del archivo con los logs almacenados y el soporte.

El algoritmo consta de tres pasos, el primer paso consiste en pasar por todos los datos y hacer un resumen de ellos, el segundo paso consiste en volver a pasar por todos los datos y utilizando la información recogida en el paso uno generar los candidatos a clúster. Por último, el tercer paso consiste en seleccionar los clústeres finales del conjunto de candidatos. A continuación, se explica de forma más detallado cada paso.

Durante el primer paso del algoritmo (en el que se resumen las palabras) el algoritmo identifica todas las palabras con la posición que ocupa dicha palabra en la línea del log, es decir, esto es equivalente a la extracción de palabras frecuentes (junto con su posición) del conjunto de datos. Para que una palabra sea considerada frecuente debe aparecer N veces en el conjunto de datos, donde N es el valor de umbral de soporte especificado como parámetro de entrada.

Tras haber identificado las palabras frecuentes, se pasa al segundo paso del algoritmo en el cual se construyen todos los candidatos de clúster durante una nueva iteración sobre los datos. Los candidatos a clúster se mantienen en una tabla de candidatos que inicialmente está vacía. El conjunto de datos se procesa línea por línea y cuando se encuentra que una línea tiene una o más de las palabras frecuentes se forma un candidato a clúster agrupando las mismas. Si el candidato a clúster no está en la tabla de candidatos, este nuevo candidato se insertará en la tabla con el valor de soporte 1, en caso de que si existiera el candidato se incrementa en uno su valor de soporte. Un candidato a clúster se forma de la siguiente manera: si la línea tiene *m* palabras frecuentes entonces el candidato a clúster tendrá *m* atributos fijos . Por ejemplo, si una línea es *Conexión desde 193.133.1.1* y existen las palabras frecuentes *(1, ‘Conexión’)* y *(2, ‘desde’)* entonces tendremos un candidato a clúster con los siguientes atributos {*(1, ‘Conexión’), (2, ‘desde’)*}.

Durante la etapa final del algoritmo, se inspecciona la tabla de candidatos a clúster y todos los candidatos con soporte igual o mayor que el umbral de soporte introducido por el usuario son propuestos por el algoritmo como clúster y construye el patrón. Por ejemplo, si hay un clúster que tiene los atributos {(1,’Contraseña’), (2,’del’), (3,’usuario’), (5,‘aceptada’)} el patrón correspondiente sería Contraseña del usuario \* aceptada.

A continuación, se muestra un ejemplo del funcionamiento de este algoritmo. Para esto definimos el soporte como y el conjunto de datos con las siguientes líneas:

Interface eth0 down

Interface eth1 down

Interface eth2 up

luego las palabras (1, ‘Interfaz’) y (3, ‘down’) aparecen en tres y dos líneas respectivamente y son identificadas como palabras frecuentes por el algoritmo. Posteriormente el algoritmo hará otra pasada sobre el conjunto de datos y creará los posibles clústeres obteniéndose, para este ejemplo, los siguientes candidatos.

{(1, ‘Interfaz’), (3, ‘down’)} {s=2}

{(1, ‘Interfaz’)} {s=1}

Por lo tanto, como el soporte es dos se tiene que hay un único clúster {(1,‘Interfaz’), (3,‘down’)} devolviendo al usuario el patrón.

Interfaz \* down

De forma que en este clúster están las dos primeras líneas, mientras que la última línea es considerada como outlier.

Este algoritmo tiene algunos fallos que se señalan a continuación.

* No es capaz de detectar comodines después de la última palabra en un patrón. Por ejemplo, si para las tres líneas del ejemplo anterior, el clúster resultante seria {(1, ‘Interfaz’)} y el usuario recibiría un patrón solo con la palabra y es de esperar que la mayoría de los usuarios preferirían el patrón Interfaz \* \*.
* Este algoritmo no es sensible a los cambios en las posiciones de las palabras dado que las posiciones de las palabras están codificadas. Si en el ejemplo anterior tenemos, además, la línea Interface HQ Link Down no se asignaría al candidato a cluster que contiene Interface y Down, sino que generaría un nuevo candidato.
* Por último, un soporte bajo puede conducir a overfitting ya que los grupos más grandes se dividen y los patrones resultantes son demasiado específicos.

## **3.2 Algoritmo LogCluster.**

El algoritmo LogCluster [10] fue diseñado por Risto Vaarandi y Mauno Phielgas en el año 2015 y es una evolución del algoritmo SLCT. Al igual que el algoritmo anterior también tiene dos parámetros de entrada, que son el nombre del archivo con los logs almacenados y el soporte.

LogCluster ve el problema de clusterización de log como un problema de minería de patrones, cada clúster se identifica de forma única por el patrón de línea que coincide con todas las líneas del clúster. LogCluster extrae los patrones de cada línea que forma el registro de datos. Cada patrón consta de palabras y comodines. Por ejemplo, el patrón **Interfaz \*{1,3} down** indica que la línea tiene las palabras Interfaz y down y el comodín \*{1,3} que indica que hay líneas con al menos 1 palabra y como máximo 3 palabras.

LogCluster encontrará patrones que tengan un soporte s o superior y se basa en que todas las palabras de dichos patrones deben ocurrir al menos *s* veces en las líneas de registro de logs. Por lo tanto, LogCluster comienza su trabajo con la identificación de tales palabras, para lo cual el algoritmo examina cada palabra sin su posición en la línea del evento del log y considera como palabras frecuentes aquellas que tenga un soporte igual o superior al dado por el usuario. Normalmente los registros de eventos grandes contienen millones de palabras y la gran mayoría de estas palabras aparecen pocas veces en los registros de eventos, por lo que la inmensa mayoría de palabras serán ignoradas una vez se pasen por el umbral de soporte.

Tras tener identificadas las palabras frecuentes, LogCluster realiza otra pasada sobre el registro de eventos y crea los candidatos a clúster. En esta segunda pasada LogCluster extrae todas las palabras frecuentes de cada línea y organiza las palabras como una tupla, conservando su posición original en la línea, de esta forma la tupla servirá como un identificador del candidato a clúster y la línea será asignada a este candidato. Si el candidato dado no existe, se inicializa con el contador de soporte establecido en 1 y su patrón de línea se crea a partir de dicha línea. Si el candidato existe, el contador de soporte incrementa en uno su valor y ajusta el patrón de la línea para que quede cubierta la línea actual.

Por ejemplo, si la línea de registro de eventos es *Interface DMZ-link down at node router2* y las palabras Interface, down, at y node son frecuentes, la línea se asigna al candidato identificado por la tupla (Interfaz, down, at, node). Si este candidato no existe, se inicializará configurando su patrón de línea como **Interface \*{1,1} down at node \*{1,1}** y su contador de soporte en 1 (recordemos que \*{1,1} indica que hay una palabra). Si la siguiente línea que analiza el algoritmo produce el mismo identificador, pero la línea es *Interface HQ link down at node router2* el contador de soporte se incrementa a 2 y el patrón de la línea se actualiza a **Interface \*{1,2} down at node \*{1,1}**, indicando que entre Interface y down hay al menos una palabra, pero no más de dos palabras.

Una vez se ha completado el análisis de todos los datos para generar los candidatos a clúster, el algoritmo descarta todos los candidatos con un valor de contador menor que el umbral de soporte introducido por el usuario e informa a éste de los candidatos restantes como clústeres finales.

A continuación, se muestra un ejemplo del funcionamiento de este algoritmo. Para ello se define el soporte como y el conjunto de datos con las siguientes líneas:

Interface eth0 down

Interface eth1 down at 11:00

Interface eth2 up

Interface HQ link down

Tras el primer paso del algoritmo se obtienen las palabras frecuentes y la cantidad de veces que aparecen (Interface, 4) y (down, 3). El algoritmo vuelve a recorrer el conjunto de datos para obtener los candidatos a clúster de la siguiente forma para este ejemplo. La primera línea tiene las dos palabras frecuentes por lo que crea un identificador con un contador {(Interface, down),1} y escribe el patrón de la línea *Interface \*{1,1} down*, luego recibe la segunda línea y vuelve a tener las dos palabras frecuentes por lo que actualiza el contador del identificador {(Interface, down),2} actualizando el primer patrón asociado a *Interface \*{1,1} down \*{2,2}*. Luego recibe la tercera línea que solo tiene la palabra frecuente Interface por lo que crea un nuevo identificador con el contador a uno {(Interface),1} y escribe el patrón de la línea *Interface \*{2,2}*. Por último, recibe la cuarta línea que contiene las palabras frecuentes Interface y down por lo que renueva el identificador con el contador {(Interface, down),3} y actualiza el patrón a *Interface \*{1,2} down \*{2,2}* de forma que al finalizar hay dos candidatos a clúster que son:

*Interface \*{1,2} down \*{2,2}*

*Interface \*{2,2}*

Con soporte 3 y 1 respectivamente. Por lo tanto, tras aplicar la condición de soporte para que un clúster sea considerado como tal el algoritmo da como resultado final un único clúster el cual es:

*Interface \*{1,2} down \*{2,2}*

Por último, si el soporte hubiera sido 4 en vez de 3, el algoritmo solo hubiera detectado la palabra Interface como frecuente y hubiese devuelto como clúster final:

*Interface \*{2,4}*

# **4. Implementación.**

En esta sección se explica cómo se van a implementar los algoritmos anteriormente descritos y la obtención de los modelos para su posterior uso en la asignación a un clúster existente de nuevas líneas de logs.

Para comenzar se debe definir qué significa el término log en el campo de la informática. Un log no es más que un registro grabado de forma secuencial en un archivo o en una base de datos de todos los acontecimientos (eventos o acciones) que afectan a un proceso particular (aplicación, actividad de una red informática, etc.) constituyendo una evidencia del comportamiento del sistema.

Generalmente en un log viene anotado el momento exacto (fecha, hora, minuto y segundo) en el que ocurrió la acción que generó dicho log, con lo que se puede analizar paso a paso lo sucedido. Por esta razón si se quiere obtener un modelo que clasifique las líneas de logs en los diferentes clústeres hay que realizar un preprocesado de los datos para que cuando encuentre una fecha diferente no lo clasifique como outlier. Consideremos por ejemplo una aplicación que cambia las contraseñas de cada usuario una vez al mes y que habrá generado la siguiente línea de log, **Jul 27 11:17:49 contraseña cambiada**, con lo que un posible patrón que el algoritmo logCluster podría determinar sería el que sigue **\*{3,3} contraseña cambiada**, pero si se realiza un preprocesado de las líneas sustituyendo en dicha línea las fechas por la palabra FECHA y la hora por la palabra HORA un posible patrón para la línea anterior podría ser [**FECHA] [HORA] contraseña cambiada** de forma que al obtener los modelos a partir de dichos patrones nuevas líneas que siguen el estándar de esta línea (la fecha y la hora serán diferentes) serían clasificadas en un clúster ya existente, no siendo depositadas como outlier.

Tras lo explicado anteriormente el ciclo del programa implementado será un preprocesado de los datos sobre los cuales posteriormente se aplicará el algoritmo que los clusteriza y por último obtener los modelos tal como se muestra en la [figura 12](#Figura12).

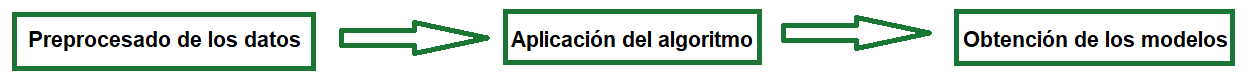


Figura 12. Ciclo de trabajo del programa implementado

Por otra parte, se trabajará con porcentajes de soporte, es decir, como parámetros de entrada se pasará un valor de 0 a 100 que corresponde con el tanto por ciento de las líneas que se quieren segmentar.

Para comenzar con el diseño del programa se creará un módulo llamado ExpresionesRegulares.py el cual se utilizará para preprocesar los datos tal y como se comentó anteriormente, de forma que se puedan añadir todos los métodos que se quieran para realizar diferentes búsquedas en las líneas de logs. En este trabajo solamente se va a preprocesar las fechas y horas que pueden aparecer en cada línea de log, pero también se podría preprocesar otros campos como pueden ser las IP.

Una vez se tiene diseñado el módulo del preprocesado se pasa a implementar el algoritmo LogCluster de dos formas diferentes, una usando apache spark y otra de forma secuencial y el algoritmo SLCT utilizando apache spark de forma que podamos comparar los resultados obtenidos entre algoritmos.

## **4.1 Implementación del módulo para el Algoritmo LogCluster secuencial.**

A continuación, se procederá a describir cómo se ha implementado el algoritmo LogCluster de forma secuencial. Esta implementación puede dividirse en tres etapas o métodos. Un primer método que determina las palabras frecuentes, un segundo método que obtiene los candidatos a clúster y un tercer método que saca los clústeres finales y sus patrones.

Comenzamos mostrando el pseudocódigo de cada método e incorporando un ejemplo que clarifica su ejecución. Se utilizará el mismo ejemplo que anteriormente.

Interface eth0 down

Interface eth1 down at 11:00

Interface eth2 up

Interface HQ link down

Empecemos con el pseudocódigo de la implementación realizada para obtener las palabras frecuentes.

1: Import modulo time

2: Import modulo preprocesado

3: Inputs: logs, , preprocesado (true o false)

4: tiempoInicial = time ()

5: numLineasTotales = 0

6: **Mientras** conexión logs abierta

7: contador = dict ()

8: **Para cada** línea de logs

9: numLineasTotales +1

10: **Si** preprocesado == True

11: línea = línea.preprocesada

12: palabras = línea.split()

13: **Para cada** palabra en palabras

14: **Si** palabra not in contador

15: contador[palabra]=1

16: **Else**

17: contador[palabra]=+1

18: **Cerrar** conexión logs

19: soporte = (s/100)\*numLineasTotales

20: palabrasFrecuentes = [ ]

21: **Para cada** palabra en contador

22: **Si** contador [palabra] >= soporte

23: palabrasFrecuentes.append (palabra)

24: **return** palabrasFrecuentes

Utilizando el ejemplo de logs anteriormente definido se pasa a explicar cómo funciona el código para . En un primer paso del algoritmo se hace un preprocesado de las líneas de logs si el usuario final quiere y en este ejemplo solo cambiaría el 11:00 por la palabra [HORA]. Luego se aplica la función split a cada línea separando el contenido de la línea cada vez que encuentra un espacio en blanco y almacenando esta información en una lista. Posteriormente, para cada palabra de la línea se comprueba si ya existe dicha palabra en el contador de palabras, si existe se suma en uno la cantidad de veces que aparece, sino existe se añade la palabra al contador y un uno. Una vez realizado esto se filtran las palabras que aparecen las mismas o más veces que el umbral introducido por el usuario. Finalmente se devuelve la lista con las palabras frecuentes. Para este ejemplo se devuelve [‘Interface’, ‘down’].

A continuación, se muestra el pseudocódigo para determinar los candidatos a clúster.

1: Inputs: logs, preprocesado (true o false), palabrasFrecuentes

2: clusterCandidatos = [ ]

3: **Mientras** conexión logs abierta

4: **Para cada** línea de logs

5: **Si** preprocesado == True

6: línea = línea.preprocesada

7: tupla = [ ]

8: varrs = [ ]

9: v = 0

10: palabras = línea.split()

11**: Para cada** palabra en palabras

12: **Si** palabra en palabrasFrecuentes

13: tupla.append(palabra)

14: varrs.append(v)

15: v = 0

16: **Else**:

17: v=+1

18: varrs.append(v)

19: k=len(tupla)

20: **Si** k>0

21: existe=False

22: **Para cada** j en rango(len(clusterCandidatos))

23: **Si** (clusterCandidatos[ j ][‘tupla’]=tupla

24: cluster=clusterCandidatos[ j ]

25: cluster[‘soporte’]=+1

26: clusterCandidatos[ j ] = cluster

27: **Para cada** t in rango (k+1)

28: **Si** cluster[‘minimo’][ t ] > varrs[ t ]

29: cluster[‘minimo’][ t ] = varrs[ t ]

30: **Si** cluster[‘maximo][ t ] < varrs[ t ]

31: cluster[‘maximo][ t ] = varrs[ t ]

32: existe = True

33: **break**

34: **Si** existe == False

35: varmin = [ ]

36: varmax = [ ]

37: **Para cada** j en rango (k+1)

38: varmin.append(varrs[ j ])

39: varmax.append (varrs[ j ])

40: cluster={‘tupla’:tupla,’soporte’:1,’minimo’:varmin,’maximo’:varmax}

41: clusterCandidatos.append(cluster)

42: **Cerrar** conexión logs

43: **return** clusterCandidatos

Vamos a explicar el pseudocódigo anterior. Tras introducir el usuario los inputs solicitados, el programa define una lista para almacenar los candidatos a clúster y mientras existan líneas de logs se hará lo siguiente para cada línea. Primero se preprocesan las líneas si el usuario así lo ha decidido. Tras esto se definen dos listas, una variable llamada **tupla** que almacena las palabras frecuentes que hay en cada línea y otra llamada **varr** que almacena un número que indica cuantas palabras hay entre cada palabra frecuente. Además, se utiliza una variable auxiliar **v** que almacena la cantidad de palabras que hay entre cada palabra frecuente. Continuamos separando las palabras de cada línea y comprobando si dicha palabra esta en las palabras frecuentes, si ésta se encuentra en las palabras frecuentes se añade a la variable tupla y a la variable varr se añade el valor de v reinicializándose el valor a cero. En caso de que la palabra no sea una palabra frecuente se suma un uno al valor de v. Una vez se acaben de analizar todas las palabras de la línea se añade a la lista varr el valor de v. Se comprueba si la tupla no está vacía, si es así se declara una variable auxiliar que almacena si existe o no esta tupla ya. Para esto se recorren los candidatos a clúster y si la tupla obtenida ya existe en algún candidato se incrementa el valor de soporte en uno y se actualiza el valor mínimo y máximo de palabras que hay entre cada palabra frecuente. En caso de que la tupla no existiese en los candidatos a clúster se añade ésta con soporte uno y dos listas que contienen el valor mínimo y máximo de palabras que hay entre las palabras frecuentes de la tupla. Para finalizar se retornan los candidatos a clúster. Para el ejemplo que se esta usando, el algoritmo devolvería una lista con los candidatos a clúster tal como se muestra a continuación.

[{'minimo': [0, 1, 0], 'maximo': [0, 2, 2], 'tupla': ['Interface', 'down'], 'soporte': 3}, {'minimo': [0, 2], 'maximo': [0, 2], 'tupla': ['Interface'], 'soporte': 1}]

Por último, se muestra el pseudocódigo para la tercera etapa en la que se determinan los patrones de los clústeres finales.

1: Inputs: , clusterCandidatos

2: patrones=[ ]

3: **Para cada** i en rango (len(clusterCandidatos))

4: patron=""

5: cluster = clusterCandidatos[ i ]

6: **Si** cluster[‘soporte’]>=s

5: k = len(cluster[‘tupla’])

6: **Para cada** j en rango(k)

49: **Si** cluster[‘maximo’][ j ] > 0

50: mínimo = cluster[‘minimo’][ j ]

51: máximo = cluster[‘maximo’][ j ]

52: patron = patron + "\*{" + mínimo + "," + máximo + " }"

53: patron = patron + cluster[‘tupla’][ j ] + " "

54: **Si** cluster[‘maximo’][ k ] > 0

55: mínimo = cluster[‘minimo’][ k ]

56: máximo = clister[‘maximo’][ k ]

57: patron = patron + "\*{" + mínimo + "," + máximo + "}"

58: patrones.append(patron)

60: **return** patrones

En este tercer y último paso se obtienen los patrones de los clústeres finales. Se le pasa como parámetros de entrada los candidatos a clúster y el soporte. El programa recorre todos los candidatos a clúster y si un candidato tiene soporte igual o mayor que el introducido por el usuario determina el patrón para ese clúster. Para obtener el patrón se obtiene la longitud de la tupla del clúster y se comprueba el valor mínimo y máximo de palabras que hay antes de cada palabra frecuente añadiendo el valor obtenido en el patrón. Posteriormente, se añade la palabra de la tupla. Finalmente se devuelven los patrones obtenidos. Para el ejemplo que se esta utilizando se obtiene solamente un patrón.

['Interface \*{1,2} down \*{0,2}']

## **4.2 Implementación del Algoritmo LogCluster usando Apache Spark.**

A continuación, se explica como se ha implementado este algoritmo usando Apache Spark ya que el desarrollo para llevar a cabo esta implementación es diferente a que si fuera secuencial. Para explicar los pasos seguidos vamos a hacer uso del ejemplo anterior, pero añadiendo una fecha al principio de cada línea.

Jul 26 Interface eth0 down

Jul 26 Interface eth1 down at 11:00

Jul 27 Interface eth2 up

Jul 27 Interface HQ link down

En este caso los inputs de entrada de la función serán: los logs, , preprocesado (true o false) y el contexto para que trabaje spark. Para explicar el funcionamiento vamos a usar un soporte del 50% que corresponde con 2 o más líneas y un preprocesado de los datos.

Para comenzar se utiliza la función map de apache spark a la cual se le pasa como argumento el método que transforma los datos (siendo x la línea de log):

**archivo=archivo.map(lambda x:transformarTexto(x))**

De forma que si aplicamos una acción sobre esta transformación obtendríamos un array de 4 componentes:

[FECHA] Interface eth0 down', '[FECHA] Interface eth1 down at [HORA]', '[FECHA] Interface eth2 up', '[FECHA] Interface HQ link down']

Una vez realizada esta transformación hay que contar las palabras frecuentes para ello primero hay que separar las palabras de la primera transformación. Posteriormente se añade un uno a cada palabra y se agrupa el resultado obtenido por palabra. Para contar las palabras se hace uso de la función flatMap pasándose como parámetro la función split aplicada a cada línea. Posteriormente, se le aplica otra transformación map de forma que añada un uno a cada palabra de la línea tal como se muestra a continuación:

**archivo.flatMap(lambda x:x.split()).map(lambda x:(x,1))**

De esta forma si se aplica la acción collect sobre el resultado anterior se obtienen todas las palabras en un array que a su vez está formado por tuplas donde cada tupla contiene la palabra y un uno.

[('[FECHA]', 1), ('Interface', 1), ('eth0', 1), ('down', 1), ('[FECHA]', 1), ('Interface', 1), ('eth1', 1), ('down', 1), ('at', 1), ('[HORA]', 1), ('[FECHA]', 1), ('Interface', 1), ('eth2', 1), ('up', 1), ('[FECHA]', 1), (‘Interface', 1), ('HQ', 1), ('link', 1), ('down', 1)]

Ya solo queda contar las palabras para lo cual se hace uso de la transformación reduceByKey que sumará todas las palabras.

**sumaPalabras=pasoTokenizar.reduceByKey(lambda x,y: x + y)**

Quedando tal como se muestra a continuación:

[('down', 3), ('Interface', 4), ('[FECHA]', 4), ('link', 1), ('eth1', 1), ('HQ', 1), ('up', 1), ('at', 1), ('[HORA]', 1), ('eth2', 1), ('eth0', 1)]

Una vez se tienen las palabras contadas hay que eliminar aquellas que aparecen menos veces que el umbral de soporte. Para esto apache spark dispone de una transformación llamada filter que se aplica tal como se muestra a continuación:

**palabrasUmbral=sumaPalabras.filter(lambda (x,y): y >= soporte)**

Con esto conseguimos obtener aquellas palabras que aparecen más veces que el umbral de soporte y además la cantidad de veces que aparece cada palabra.

[('down', 3), ('Interface', 4), ('[FECHA]', 4)]

Para terminar solo nos interesan las palabras por lo que aplicamos una transformación map para obtenerlas y almacenarlas en memoria.

**palabras=palabrasUmbral.map(lambda x: x[0]).collect()**

De esta forma se ha implementado la primera parte del algoritmo que es la de contar las palabras frecuentes, que para este ejemplo son:

['down', 'Interface', '[FECHA]']

Una vez se han determinado las palabras frecuentes se pasa a la segunda parte del algoritmo donde se determinan las palabras frecuentes que hay en cada línea, así como cuantas palabras hay entre cada palabra frecuente. Para ello se crea una función a la que se la pasa como parámetros de entrada las palabras frecuentes y la línea de log. Esta función devolverá las palabras frecuentes que hay en cada línea, así como el número de palabras que hay entre ellas. Para realizar esto se utiliza la transformación flatMap de apache spark tal como se muestra a continuación.

**obtenerPalabras=archivo.flatMap(lambda x:((encontrarPalabrasLogCluster(palabras,x))))**

Vemos que se obtiene un array compuesto por tuplas y que estas tuplas a su vez están compuestas por otras dos. Una encargada de almacenar las palabras y otra que almacena el número de palabras que hay entre las palabras frecuentes.

[(('[FECHA]', 'Interface', 'down'), (0, 0, 1, 0, 1)), (('[FECHA]', 'Interface', 'down'), (0, 0, 1, 2, 1)), (('[FECHA]', 'Interface'), (0, 0, 2, 1)), (('[FECHA]', 'Interface', 'down'), (0, 0, 2, 0, 1))]

Tras aplicar un filtro para eliminar los campos vacíos (en este caso todas las líneas tenían alguna palabra frecuente) hay que unificar cuando las tuplas de palabras son iguales y a la vista de lo anterior se observa que solo existen dos tuplas de palabras diferentes por lo que se aplica una transformación reduceByKey que agrupa el array según las tuplas. Además, hay que calcular el máximo y mínimo de la otra tupla, obteniendo de esta forma el número de palabras mínimo y máximo. Para crear esto se crea una función llamada buscarMinMax que será la encargada de transformar las tuplas que contienen la cantidad de palabras entre palabras frecuentes. Los candidatos a clúster vendrían dados tal como se muestra:

**candidatosCluster=lineasFiltradas.reduceByKey(lambda v1,v2:buscarMinMax(v1,v2))**

Si hacemos un collect de la transformación anterior vemos que el array se ha transformado en otro de dos componentes con tres tuplas por componente. La primera tupla contiene las palabras frecuentes, la segunda tupla contiene el mínimo y máximo que hay antes de cada palabra frecuente y la tercera tupla contiene las veces que ha aparecido la primera tupla.

[(('[FECHA]', 'Interface'), (0, 0, 2, 1)), (('[FECHA]', 'Interface', 'down'), ((0, 0), (0, 0), (1, 2), (0, 2), 3))]

Con esto ya tenemos los candidatos a clúster, ahora basta con aplicar un filtro para aceptar aquellos candidatos con el soporte mayor o igual que el introducido por el usuario. En este caso tenemos que hay únicamente un clúster candidato.

[(('[FECHA]', 'Interface', 'down'), ((0, 0), (0, 0), (1, 2), (0, 2)))]

Para finalizar se obtienen los patrones de los clústeres finales y para este ejemplo en el que solo hay un clúster final el patrón devuelto sería:

['[FECHA] Interface \*{1,2} down \*{0,2}']

## **4.3 Implementación del Algoritmo SLCT usando Apache Spark.**

En este apartado al igual que en el anterior vamos a pasar a explicar cómo se ha implementado el algoritmo Simple Logfile Clustering Tool (SLCT) haciendo uso de la tecnología apache spark. Para explicar esto se hará uso del ejemplo anterior. Lo primero que hay que hacer es contar las palabras, pero teniendo en cuenta la posición que ocupan en la línea, es decir hay que contar cuantas palabras iguales hay en la misma posición.

Tras leer los datos los pasos a seguir para contar las palabras son prácticamente los mismos que en el algoritmo anterior, salvo al comenzar con las transformaciones. La primera transformación consiste en añadir un índice a las palabras usando la función flatMap e index que permite separar las palabras e indexarlas.

**lineaTokenizada =archivo.flatMap(lambda x: x.split()).indext() )**

De esta forma se indexan las palabras según la posición que ocupan en su línea.

[(1, '[FECHA]'), (2, 'Interface'), (3, 'eth0'), (4, 'down'), (1, '[FECHA]'), (2, 'Interface'), (3, 'eth1'), (4, u'down'), (5, 'at'), (6, '[FECHA]'), (1, '[FECHA]'), (2, 'Interface'), (3, 'eth2'), (4, 'up'), (1, '[FECHA]'), (2, 'Interface'), (3, 'HQ'), (4, 'link'), (5, 'down')]

A partir de aquí se procede de la misma forma que en el algoritmo anterior aplicándose una transformación map para añadir unos a todos los campos. Posteriormente se hace una transformación reduceByKey donde la key es la posición que ocupa la palabra y la palabra. Para finalizar se aplica un filtro para solamente aceptar las palabras que aparezcan más o igual veces que el soporte, con lo que para el ejemplo anterior obtendríamos las palabras con su posición.

[(4, 'down'), (2, 'Interface'), (1, '[FECHA]')]

Ahora obtenemos los candidatos a clúster. Para esto se hace uso de la función map a la que se le pasa como argumento una función que encuentra las palabras junto con su posición.

**obtenerPalabras=lineaTokenizada.map(lambda x:((encontrarPalabras(palabras,x))))**

Esta transformación almacena en un array tuplas que contienen cada una de las palabras y su posición para cada línea tal como se muestra a continuación.

[((1, '[FECHA]'), (2, 'Interface'), (4, 'down')), ((1, '[FECHA]'), (2, 'Interface'), (4, 'down')), ((1, '[FECHA]'), (2, 'Interface')), ((1, '[FECHA]'), (2, 'Interface'))]

Ahora hay que hacer que los campos tengan clave valor por lo que se añade un uno a cada campo y posteriormente se aplica un groupByKey que agrupará las palabras según su clave.

**agrupacion=obtenerPalabras.groupByKey().mapValues(len)**

Almacenando además las veces que se repiten en el conjunto de datos, esto se muestra a continuación.

[(((1, '[FECHA]'), (2, 'Interface'), (4, 'down')), 2), (((1, '[FECHA]'), (2, 'Interface')), 2)]

Tras aplicar el umbral de soporte se obtiene que de los dos candidatos a clústeres pasa a ser clústeres finales. A continuación, se escriben los patrones haciendo uso de la posición de la palabra y dicha palabra. Para este ejemplo los patrones devueltos son:

['[FECHA] Interface \* down ', '[FECHA] Interface ']

Con lo que este algoritmo queda implementado.

## **4.4 Obtención de los modelos.**

Una vez se han determinado los patrones de los clústeres hay que obtener los modelos que identifican dichos patrones, es decir, obtener las expresiones regulares que cumplen dichos patrones. Para esto hay que pasar como parámetros los patrones y simplemente tener claro con cuál de los dos algoritmos se han obtenido dichos patrones.

* **Algoritmo SLCT:**
  + Cuando aparece un \* en el patrón quiere decir que ahí puede ir cualquier carácter que no sea un espacio en blanco de longitud uno o mayor por lo que en el modelo hay que sustituirlo por un \S\*.
  + Cuando aparecen caracteres especiales como pueden ser los corchetes, los paréntesis o un punto entre otros se pone la barra \ de escape delante de ellos para indicar que es el carácter que indica su forma.
  + Por último, se le indica que entre palabras en el patrón puede existir un hueco en blanco o más.
* **Algoritmo LogCluster:**
  + Cuando aparecen caracteres especiales como pueden ser los corchetes, los paréntesis o un punto entre otros se pone la barra \ de escape delante de ellos para indicar que es el carácter que indica su forma.
  + Cuando se encuentra que en el patrón aparece el formato \*{min,max} obtiene el conjunto de datos mediante la expresión (\s\*\S\*) que tiene que aparecer obligatoriamente el número de veces que indica el mínimo. Cuando ya pasa este valor mínimo se le indica que puede o no aparecer hasta el número máximo de veces (\s\*\S\*)? donde ? indica que puede aparecer como mucho una vez.
  + Por último, se le indica que entre palabras en el patrón puede existir un hueco en blanco o más.

Vamos a utilizar los patrones obtenidos en el ejemplo anterior utilizando los dos algoritmos para ver como serían los modelos obtenidos finalmente.

* + - **Algoritmo SLCT:**

Recordemos que los patrones obtenidos utilizando este algoritmo fueron dos '[FECHA] Interface \* down ' y ‘[FECHA] Interface' por lo que nuestro modelo tendrá dos expresiones regulares que serán:

* ^\[FECHA\]\s\*Interface\s\*\S\*\s\*down
* ^\[FECHA\\]\s\*Interface

Con estos modelos podríamos clasificar nuevos logs e identificar si son outlier.

* + - **Algoritmo LogCluster:**

Recordemos que el patrón obtenido utilizando este algoritmo fue '[FECHA] Interface \*{1,2} down \*{0,2}' por lo que nuestro modelo tendrá una única expresión regular que será:

* ^\s\*\[FECHA\]\s\*Interface(\s\*\S\*)(\s\*\S\*)?\s\*down(\s\*\S\*)?(\s\*\S\*)?

Como se ve, este modelo tendrá una única expresión regular la cual indica que una línea de logs debe empezar por un espacio en blanco o una fecha para posteriormente encontrarse con un espacio en blanco seguido de la palabra Interface. Luego un conjunto de espacios en blanco y caracteres cualesquiera juntos, pudiendo aparecer otra vez este formato. Posteriormente debe encontrarse la palabra down pudiendo aparecer dos palabras más.

Por otra parte, puede que el algoritmo encuentre patrones más generales que engloban a otros más específicos con lo que una nueva línea de logs podría pertenecer a más de un patrón, por lo que quizá si queremos que una línea pertenezca a un único clúster hay que dar un peso a cada patrón para obtener los que más información contienen como más importantes, aunque esto no se ha implementado en este trabajo.

## **4.5 Primeros pasos.**

Para poder trabajar con las implementaciones anteriormente descritas hay que ingresar los datos que están almacenados en ficheros de texto en elasticsearch para lo cual se hará uso de logstash.

Este fichero consta de tres partes, dos obligatorias (input y output) y una opcional (filter). En este caso vamos a dejar la opción del filter sin configurar ya que se va a introducir toda la información tal como se recibe. La parte input recibirá la información de los logs, es decir, donde están almacenados que en este caso es un fichero de texto (file) y aquí se le indica la ruta de los datos y desde donde empieza a leer entre otras opciones.

Para el output se le indica que se quiere guardar la información en elasticsearch y indicandole el índice para que se guarde y el host donde se encuentra corriendo.

A continuación, se muestra el archivo de configuración utilizado.

input

{

file

{

path => "/home/jvillar/Escritorio/Trabajo/syslog.txt"

sincedb\_path => "/dev/null"

#start\_position => "beginning"

}

}

# The fillter part ofj this file is commented out to indicate that it is optional.

# filter

#{

#

# }

output

{

elasticsearch

{

index => "TFM"

}

}

Ya solo falta crear y conformar un contexto de Apache Spark para conseguir configurar Apache Spark de manera interna. Para esto se asignará un nodo master que será el encargado de gestionar toda la aplicación y los nodos que se quiera que sean los worker para lo cual se usa la siguiente instrucción:

*conf = SparkConf().setAppName("Algoritmos").setMaster("local")*

*sc = SparkContext(conf=conf)*

Por último, ya solo falta realizar la conexión con elasticsearch y apache spark directamente sin tener que almacenar la información que contiene la base de datos en ninguna variable extra. Para ello se usa la siguiente configuración.

*es\_read\_conf = { "es.nodes" : 'localhost', "es.port" : '9200', "es.resource" : index/doc-type}*

Siendo el parámetro es.resource diferente para cada documento de datos que se analiza y es.nodes y es.port comunes para todos ya que es donde se encuentra elasticsearch.

* es.nodes: especifica el nodo al que estamos enviando los datos
* es.port: especifica el puerto en el que se encuentra elasticsearch.
* es.resource: especifica un recurso en la forma ‘índice/tipo de documento’

Ya solo queda configurar el contexto para que funcione correctamente para lo cual hay que bajarse un archivo .jar y configurarlo como variable de entorno, tal como se muestra:

*os.environ['PYSPARK\_SUBMIT\_ARGS'] = '--jars /home/jvillar/spark-2.3.0-bin-hadoop2.7/jars/elasticsearch-hadoop-5.5.3.jar pyspark-shell'*

Finalmente, solo queda utilizar un RDD para los datos almacenados para lo cual se hace uso de la documentación de elastic y que para nuestro caso queda de la siguiente forma.

*es\_rdd = sc.newAPIHadoopRDD(*

*inputFormatClass="org.elasticsearch.hadoop.mr.EsInputFormat",*

*keyClass="org.apache.hadoop.io.NullWritable",*

*valueClass="org.elasticsearch.hadoop.mr.LinkedMapWritable",*

*conf=es\_read\_conf )*

Para analizar los archivos de configuración y estudiar los algoritmos mejor vamos a realizar representaciones graficas para comparar estos mismos. Para esto hemos recogido la información siguiente para poder hacer las representaciones correctas.

* Patrones: Son los clústeres finales que han sido aceptados por el algoritmo, pero ya modificados para ver los patrones.
* Soporte de líneas: Es la cantidad de líneas que pertenecen a cada clúster.
* Tiempo empleado: Es el tiempo total que el algoritmo ha estado ejecutándose para obtener los clústeres finales.
* Número de candidatos a clúster: es el número de todos los clústeres que se han formado utilizando el algoritmo.
* Numero de clúster finales: son los clústeres finales tras aplicar el valor de soporte.
* Porcentaje de líneas clusterizadas: es el tanto por ciento del total de líneas que pertenecen a algún clúster.

# **5. Ejemplos de uso.**

Estos dos algoritmos se pueden utilizar para cualquier archivo de logs, pero para mostrar un ejemplo de cómo funcionan se van a probar en dos archivos totalmente opuestos entre sí. Para realizar estas pruebas se utilizará una maquina con Ubuntu 16.04 y un único nodo y se probaran en un fichero que contiene solamente 2633 líneas de un sistema operativo y el otro que contiene 55116 líneas que tendrá información sobre el envió de datos de temperatura de varias estaciones.

Para ejecutar el programa diseñado se utilizarán los siguientes parámetros:

* Soporte: desde el 5% hasta el 80%
* Sc: como el contexto de apache spark.
* Preprocesado: True

A continuación, pasamos a tratar los dos archivos.

## **5.1 Fichero de logs de un sistema operativo.**

Para comenzar a analizar este fichero veamos primero un ejemplo de una línea de dicho fichero *Jul 26 09:36:29 RUE3 anacron[1102]: Job `cron.daily' terminateed*. Como vemos viene una fecha y una hora seguida de palabras. A continuación, veamos cuánto tarda en ejecutarse el algoritmo LogCluster y SLCT.

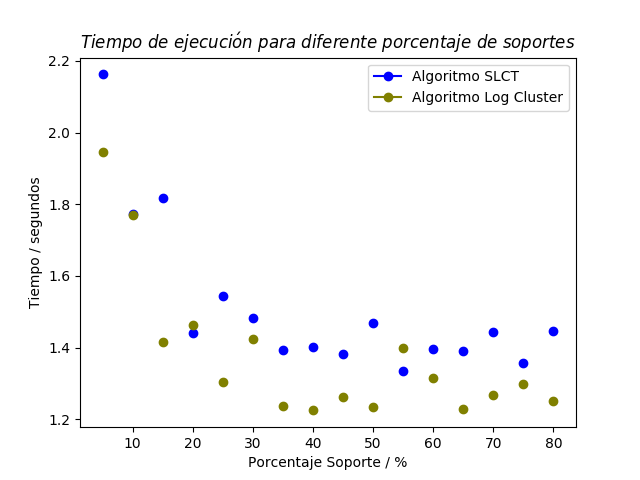


Figura 12. Tiempo de ejecución para diferentes porcentajes de soportes.

En la [figura 12](#Figura12) puede verse que los tiempos de ejecución son muy parecidos entre sí y como según se incrementa el soporte el tiempo de ejecución disminuye, esto es algo esperado ya que se supone que a mayor soporte menos palabras frecuentes debe encontrar por lo tanto el algoritmo debe trabajar menos. También se observa que el algoritmo LogCluster en general tarda menos tiempo aunque esto no podemos decir que simplemente sea por el contexto de spark ya que seguramente tarde más en crear todos los RDD que en ejecutarse.

Para continuar con el análisis examinamos cuantos candidatos a clúster obtiene cada algoritmo y para esto podemos ver la [figura 13](#Figura13) que muestra el número de candidatos a clúster en función del porcentaje de soporte. Como se puede ver el algoritmo LogCluster siempre propone más candidatos a clúster que el algoritmo SLCT. Esto seguramente se debe a que hay palabras en las líneas de logs que se repiten y como en el algoritmo SLCT se tiene en cuenta la posición de la palabra, seguramente ésta fue rechazada como palabra frecuente. Además, se puede ver como para un soporte comprendido entre el 15% y el 70 % se obtiene el mismo número de clústeres candidatos, esto quiere decir que no se están perdiendo candidatos a clúster al aumentar el soporte.

Imagen que contiene captura de pantalla

Descripción generada con confianza muy alta

Figura 13. Número de candidatos a clúster para diferentes porcentajes de soportes

Pero lo que en verdad recibe el usuario final son los clústeres que superan el soporte. Éstos se pueden ver en la [figura 14](#Figura14). A la vista de esta figura puede apreciarse como el algoritmo SLCT siempre ofrece al menos un clúster sin embargo el algoritmo LogCluster no siempre como puede verse para un soporte del 70% esto se debe simplemente al soporte ya que ningún candidato a clúster ha sido aceptado por lo que todos son rechazados.

Imagen que contiene captura de pantalla

Descripción generada con confianza muy alta

Figura 14. Número de clúster finales para diferentes porcentajes de soportes.

Para finalizar con la comparación vamos a mostrar en la [figura 15](#Figura15) el porcentaje de líneas que pertenecen a algún clúster y que no serían consideradas outlier. Como puede apreciarse en dicha [figura 15](#Figura15) vemos que para soportes del 5% y desde el 30% hasta el 65% se clusterizan el mismo porcentaje de líneas. Además también se observa que el algoritmo SLCT consigue asignar más líneas a los clústeres finales para todos los soportes, llegando este algoritmo incluso a asignar todas las líneas.

Imagen que contiene captura de pantalla

Descripción generada con confianza muy alta

Figura 15. Porcentaje de líneas clusterizadas para diferentes porcentajes de soportes.

Para continuar con nuestro análisis vamos a comparar el tiempo de ejecución para las dos implementaciones realizadas para el algoritmo LogCluster. Este análisis se puede ver en la [figura 16](#Figura16) que muestra el tiempo de ejecución del algoritmo para ambos casos. Como puede apreciarse el algoritmo LogCluster secuencial tiene un tiempo de ejecución prácticamente lineal mientras que utilizando apache spark tiene saltos. Esto se debe prácticamente a que en el LogCluster implementado usando apache spark tiene que crear el contexto y el RDD y por eso hay esa diferencia de tiempos ya que en algunos casos tarda más que en otros, pero como puede verse cuando se tiene un archivo con pocas líneas da igual que implementación usar porque apenas hay diferencia.

Imagen que contiene captura de pantalla

Descripción generada con confianza alta

Figura 16. Tiempo de ejecución para dos implementaciones diferentes del algoritmo log clúster para diferentes porcentajes de soportes.

Para finalizar el análisis nos fijamos en los patrones retornados y en el número de líneas que siguen dichos patrones. Estos resultados se pueden ver en el [Anexo I](#_Anexo_I.). A la vista de los resultados mostrados anteriormente en las gráficas y en el [Anexo I](#_Anexo_I.) se puede concluir que ambos algoritmos, salvo para soportes del 5%, 10% y 70%, encuentran los mismos patrones salvo en el final de éstos ya que el algoritmo LogCluster indica las palabras que van al final del patrón. Por otra parte, también vemos que hay patrones que pueden estar dentro de otros más generales como puede ser para el caso de soporte del 20% en el que se encuentran dos patrones [FECHA] [HORA] RUE3 \*{1,21}' y '[FECHA] [HORA] RUE3 kernel: [ \*{2,23} para el algoritmo LogCluster y en los cuales se puede ver como el primer patrón engloba a la mayoría de las líneas del segundo. De hecho, si en el primer patrón se cambia el 21 por un 25 estarían cubiertas todas las líneas asignadas al segundo patrón.

Examinando toda esta información se llega a la conclusión de que no hay un mejor soporte, ya que para este caso tenemos tres soportes como son del 15%, 20% y 25% que son igual de buenos y con los que se obtienen los mismos resultados. Por lo que con cualquiera de estos resultados se obtienen los modelos (recordemos que los modelos no son más que expresiones regulares que cumplen dichos patrones). Se obtienen los siguientes modelos:

**Modelos para los patrones obtenidos con el algoritmo LogCluster y con soporte 15%, 20% o 25%**

Patrón:

[FECHA] [HORA] RUE3 \*{1,21}

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*RUE3(\s\*\S\*)(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?

Patrón:

[FECHA] [HORA] RUE3 kernel: [ \*{2,23}

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*RUE3\s\*kernel:\s\*\[(\s\*\S\*)(\s\*\S\*)(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?(\s\*\S\*)?

Ahora cada línea nueva que llegue se le pasa por este modelo y si no cumple ninguno de los dos patrones se clasifica como un outlier.

**Modelos para los patrones obtenidos con el algoritmo SLCT y con soporte 15%, 20% o 25%**

Patrón:

[FECHA] [HORA] RUE3

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*RUE3

Patrón:

[FECHA] [HORA] RUE3 kernel: [

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*RUE3\s\*kernel:\s\*\[

## **5.2 Fichero de logs de envió de datos de plataformas.**

Aplicamos este algoritmo ahora a un fichero que almacena logs de una plataforma que envía datos a otra plataforma para su análisis posterior. Para entender estos logs vamos a ver un ejemplo de lo que se envía. Normalmente esta plataforma envía los datos cada media hora por lo que cuando comienza a enviar un dato se genera un log de la siguiente forma:

2018-08-05 00:00:48.886 +02:00 [Debug] Sending (current) E=50; PW=DebodFSMLR2016; F=2018-08-04; H=23:36; ID=6400; V=0.0000; ID=6401; V=0.0000;

Pero también ocurre que el dato se puede enviar más tarde de lo debido ya que se han podido producir fallos de comunicación. Entonces si se consigue recuperar la conexión y se envían los datos en el fichero de log se escribirá algo como lo que sigue:

2018-08-05 00:00:48.886 +02:00 [Debug] Sending (late) E=50; PW=DebodFSMLR2016; F=2018-08-04; H=23:36; ID=6400; V=0.0000; ID=6401; V=0.0000;

Obsérvese que la línea que se escribe es la misma salvo cambiando late por current, en principio, deberían existir más líneas que se envían con current, por lo que cabe esperar que el algoritmo encuentre patrones con esta forma y por consiguiente poder obtener un modelo para que cuando no cumpla éste se pueda avisar de que pueden existir errores.

Para comenzar vamos a realizar el mismo análisis que en el caso anterior, pero teniendo en cuenta que ahora estamos ante 55.116 líneas.

Se empieza ejecutando ambos algoritmos y examinando cual es el tiempo de ejecución de cada uno.

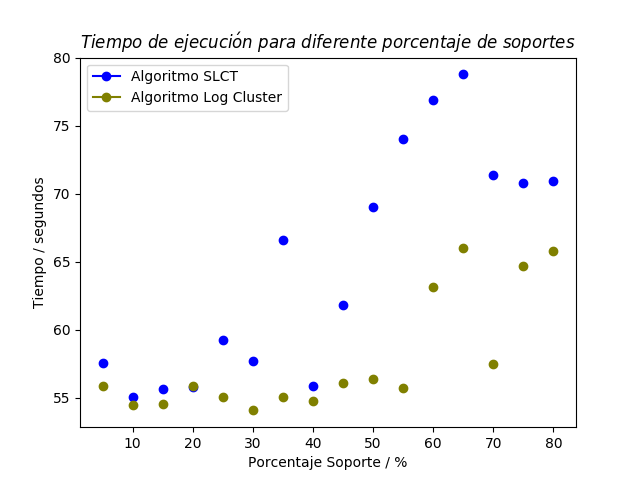


Figura 17. Tiempo de ejecución para diferentes porcentajes de soportes.

En la [figura 17](#Figura17) puede apreciarse el tiempo de ejecución de los dos algoritmos implementados. En esta ocasión se puede ver como se ha tardado más en analizar todos los datos disponibles que para el caso de uso anterior, y además también vemos como el algoritmo SLCT en general tarda más en ejecutarse completamente que el LogCluster y como el primero dispara su tiempo de ejecución a mayores soportes. Seguramente esto se deba al tiempo máquina, es decir, es posible que en ese momento el nodo de trabajo no solo estuviese dedicado a trabajar para dicho algoritmo, sino que estaría trabajando en otros procesos, pero a la vista de los resultados se puede concluir que el algoritmo LogCluster en líneas generales tardaría entre 55 y 60 segundos en devolver un resultado, mientras el algoritmo SLCT no tiene un término medio.

En la [figura 18](#Figura18) se puede apreciar el número de clústeres candidatos y se puede observar que para un soporte bajo hay muchos candidatos a clúster sin embargo a partir de un soporte del 20% los dos algoritmos obtienen 3 candidatos a clúster para el resto de los soportes lo que quiere decir que no se están reduciendo los candidatos a clúster seguramente debido a que las palabras frecuentes que se encuentran son las mismas a partir del 20% sin desaparecer palabras. Por lo que podemos concluir que no es necesario aplicar soportes superiores y como se puede ver en la [figura 17](#Figura17) gastar menos tiempo en la ejecución.

Imagen que contiene captura de pantalla

Descripción generada con confianza muy alta

Figura 18. Número de candidatos a clúster para diferentes porcentajes de soportes

En la [figura 19](#Figura19) se puede ver el número final de clústeres y como puede apreciarse a partir de un soporte del 20% que ambos algoritmos ofrecen un único clúster final como solución con lo que se cumplen las sospechas descritas anteriormente de que a partir de dicho soporte no se pierde información. Por último, cabe señalar como el algoritmo LogCluster a pesar de tener más candidatos a clúster siempre acaba teniendo el mismo número de clústeres finales que el algoritmo SLCT.

Imagen que contiene captura de pantalla

Descripción generada con confianza muy alta

Figura 19. Número de clúster finales para diferentes porcentajes de soportes

Para tomar una decisión final, hay que ver qué porcentaje de líneas acaban siendo distribuidas en los diferentes clústeres y por lo tanto ver cuántos outlier podrían existir. Esto se muestra en la figura siguiente.

Imagen que contiene captura de pantalla

Descripción generada con confianza muy alta

Figura 20. Porcentaje de líneas clusterizadas para diferentes porcentajes de soportes.

A la vista de los resultados mostrados en la [figura 20](#Figura20) se puede ver todo lo comentado ya anteriormente y es que a partir de un soporte del 20% en adelante no se pierde información ni se obtiene un clúster más general que englobe a más líneas. Lo que sí que se ve es que ambos algoritmos consiguen distribuir en los diferentes clústeres finales el mismo porcentaje de líneas indicando esto que la única diferencia entre los patrones de ambos algoritmos será únicamente la parte final ya que en el LogCluster aparecerán el número de palabras finales mientras que en el SLCT no.

Tras realizar todo este análisis nos queda preguntarnos si de verdad con apache spark para el algoritmo LogCluster se gana tiempo respecto a una implementación clásica. Ya se ha visto en la [figura 16](#Figura16) que para pocas líneas es más rápido no hacer uso de una implementación usando apache spark, pero para este caso en el que tenemos más de 55.000 líneas quizás sí que sea más veloz utilizar apache spark. Esto lo comprobamos en la [figura 21](#Figura21) en la que se muestra el tiempo de ejecución para ambas implementaciones del algoritmo LogCluster y como puede apreciarse se ve claramente que llevar a cabo la implementación utilizando apache spark reduce claramente el tiempo de ejecución y a todo esto hay que recordar que el código se está ejecutando para un único nodo. Con esto se hace hincapié en que si hubiese más de un nodo el tiempo de ejecución seria incluso menor, por lo que se puede concluir que es conveniente utilizar apache spark cuando se tiene un gran número de líneas ya que en realidad se suelen tener muchas más líneas de las que se han usado para este trabajo.

Imagen que contiene captura de pantalla

Descripción generada con confianza alta

Figura 21. Tiempo de ejecución para dos implementaciones diferentes del algoritmo log clúster para diferentes porcentajes de soportes.

Finalizamos el análisis fijándonos en los patrones retornados y en el número de líneas que siguen dichos patrones (ver [Anexo II](#_Anexo_II.)). A la vista de estos resultados se observa que los patrones a partir de un soporte del 20% son muy generales y además excluyen demasiadas líneas.

Analizando todo lo descrito anteriormente y el fichero de logs, el mejor soporte para ambos algoritmos será un soporte del 5% ya que clasifica más líneas en diferentes clústeres.

Finalmente se obtienen los modelos que nos clasificarán nuevas líneas en los diferentes clústeres existentes o bien como outlier.

**ALGORITMO LOG CLUSTER**

Modelos para los patrones obtenidos con el algoritmo LogCluster y con soporte 5%

Patrón:

[FECHA] [HORA] [Debug] Enqueue \*{1,1} in current queue.

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Enqueue(\s\*\S\*)\s\*in\s\*current\s\*queue\.

Patrón:

[FECHA] [HORA] [Debug] Number of bytes read: \*{1,1}

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Number\s\*of\s\*bytes\s\*read:(\s\*\S\*)

Patrón:

[FECHA] [HORA] [Debug] step 2

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*step\s\*2

Patrón:

[FECHA] [HORA] [Debug] Pending messages en queue (late=False): 1

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Pending\s\*messages\s\*en\s\*queue\s\*\(late=False\):\s\*1

Patrón:

[FECHA] [HORA] [Debug] myCompleteMessage: \*{1,1}

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*myCompleteMessage:(\s\*\S\*)

Patrón:

[FECHA] [HORA] [Debug] Sent: \*{1,1}

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Sent:(\s\*\S\*)

Patrón:

[FECHA] [HORA] [Debug] Enqued: \*{1,1}

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Enqued:(\s\*\S\*)

Patrón:

[FECHA] [HORA] [Information] Client connected \*{1,1} Waiting for data.

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Information\]\s\*Client\s\*connected(\s\*\S\*)\s\*Waiting\s\*for\s\*data\.

Patrón:

[FECHA] [HORA] [Debug] myData: \*{1,1}

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*myData:(\s\*\S\*)

Patrón:

[FECHA] [HORA] [Information] Waiting for client...

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Information\]\s\*Waiting\s\*for\s\*client\.\.\.

Patrón:

[FECHA] [HORA] [Debug] step 3

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*step\s\*3

Patrón:

[FECHA] [HORA] [Debug] Sending (current) \*{1,1}

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Sending\s\*\(current\)(\s\*\S\*)

**ALGORITMO SLCT**

Modelos para los patrones obtenidos con el algoritmo Slct y con soporte 5%

Patrón:

[FECHA] [HORA] [Information] Client connected \* Waiting for data.

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Information\]\s\*Client\s\*connected\s\*\\*\s\*Waiting\s\*for\s\*data\.

Patrón:

[FECHA] [HORA] [Debug] myCompleteMessage:

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*myCompleteMessage:

Patrón:

[FECHA] [HORA] [Debug] Sending (current)

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Sending\s\*\(current\)

Patrón:

[FECHA] [HORA] [Debug] Number of bytes read:

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Number\s\*of\s\*bytes\s\*read:

Patrón:

[FECHA] [HORA] [Information] Waiting for client...

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Information\]\s\*Waiting\s\*for\s\*client\.\.\.

Patrón:

[FECHA] [HORA] [Debug] step 3

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*step\s\*3

Patrón:

[FECHA] [HORA] [Debug] step 2

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*step\s\*2

Patrón:

[FECHA] [HORA] [Debug] Pending messages en queue (late=False): 1

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Pending\s\*messages\s\*en\s\*queue\s\*\(late=False\):\s\*1

Patrón:

[FECHA] [HORA] [Debug] Sent:

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Sent:

Patrón:

[FECHA] [HORA] [Debug] Enqueue \* in current queue.

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Enqueue\s\*\\*\s\*in\s\*current\s\*queue\.

Patrón:

[FECHA] [HORA] [Debug] Enqued:

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*Enqued:

Patrón:

[FECHA] [HORA] [Debug] myData:

Modelo:

^\s\*\[FECHA\]\s\*\[HORA\]\s\*\[Debug\]\s\*myData:

Como puede apreciarse los modelos en su parte inicial son iguales para ambos algoritmos por lo que el modelo para el algoritmo SLCT podría ser perfectamente descartable ya que el algoritmo LogCluster es más completo. Si por ejemplo usamos la línea de logs del principio que indica que la señal ha llegado tarde y la pasamos por nuestro modelo esta línea quedará etiquetada como outlier y el usuario podría comprobar si existen errores de conexión entre otros posibles errores.

# **6. Conclusiones.**

Se han implementado dos algoritmos del mismo autor utilizando tecnología big data como es apache spark combinada con Python, además también se ha utilizado tecnología para el almacenamiento de logs con elasticsearch, su visualización con Kibana y su transformación y carga con logstash.

Para entender bien como esta tecnología big data mejora los tiempos de ejecución con grandes volúmenes de datos se han utilizado dos conjuntos de datos diferentes uno con muy poca cantidad de datos y el otro conjunto con un gran volumen de datos para así poder comparar los diferentes tiempos entre el uso de esta tecnología y el no uso. Los resultados obtenidos para un gran volumen de datos muestran como se reduce el tiempo de ejecución con esta tecnología.

Por otro lado, queda en evidencia que al aplicar estos dos algoritmos a un conjunto de datos es conveniente conocer y analizar este conjunto de datos, ya que el usuario tiene que decidir si quiere preprocesar los datos o no y lo más importante que soporte utilizar ya que esta decisión es crítica porque se pueden obtener patrones/modelos muy generales o en caso contrario muy representativos. Es importante buscar un término medio y esto solo se consigue si el usuario conoce un poco la información a tratar.

Para finalizar se observa que el algoritmo SLCT da resultados muy parecidos que LogCluster, pero sin dar la información de las palabras que pueden existir al final de cada línea.

Como trabajo futuro hay que destacar una posible mejora en la obtención de los modelos, ya que estos algoritmos pueden obtener un clúster más general y otros más específicos que para soportes mayores pueden estar dentro de este cluster general, la idea es asignar pesos a los patrones y ordenarlos de mayor a menor peso y de esta forma una línea nueva sea solo asignada a un único clúster.

El grueso de este trabajo puede consultarse en github [11] en el que está todo el código utilizado, así como las gráficas y documentos de texto.

# **Anexo I.**

**Nombre del fichero: syslog.txt, numero de lineas totales: 2633**

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 5 %

('[FECHA] [HORA] RUE3 kernel: [ 0.000000] ', 268)

('[FECHA] [HORA] RUE3 ', 456)

('[FECHA] [HORA] RUE3 kernel: [ ', 1184)

**Log cluster apache spark**, Soporte: 5 %

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1067)

('[FECHA] [HORA] RUE3 kernel: [ 0.000000] \*{0,4} [mem \*{1,5}', 154)

('[FECHA] [HORA] RUE3 \*{1,16}', 397)

('[FECHA] [HORA] RUE3 kernel: [ 0.000000] \*{2,16}', 160)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 10 %

('[FECHA] [HORA] RUE3 ', 714)

('[FECHA] [HORA] RUE3 kernel: [ 0.000000] ', 384)

('[FECHA] [HORA] RUE3 kernel: [ ', 1504)

**Log cluster apache spark**, Soporte: 10 %

('[FECHA] [HORA] RUE3 \*{1,21}', 704)

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1306)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 15 %

('[FECHA] [HORA] RUE3 ', 714)

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 15 %

('[FECHA] [HORA] RUE3 \*{1,21}', 704)

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 20 %

('[FECHA] [HORA] RUE3 ', 714)

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 20 %

('[FECHA] [HORA] RUE3 \*{1,21}', 704)

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 25 %

('[FECHA] [HORA] RUE3 ', 714)

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 25 %

('[FECHA] [HORA] RUE3 \*{1,21}', 704)

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark)

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 30 %

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 35 %

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 35 %

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 40 %

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 40 %

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 45 %

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 45 %

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 50 %

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 50 %

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 55 %

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 55 %

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 60 %

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 60 %

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 65 %

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 65 %

('[FECHA] [HORA] RUE3 kernel: [ \*{2,23}', 1810)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 70 %

('[FECHA] [HORA] RUE3 kernel: [ ', 1888)

**Log cluster apache spark**, Soporte: 70 %

-

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 75 %

('[FECHA] [HORA] RUE3 ', 2633)

**Log cluster apache spark**, Soporte: 75 %

('[FECHA] [HORA] RUE3 \*{1,25}', 2619)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 80 %

('[FECHA] [HORA] RUE3 ', 2633)

**Log cluster apache spark**, Soporte: 80 %

('[FECHA] [HORA] RUE3 \*{1,25}', 2619)

# **Anexo II.**

**Nombre del fichero: ShipperServer.txt, numero de lineas totales: 55116**

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 5 %

('[FECHA] [HORA] [Information] Client connected \* Waiting for data. ', 4597)

('[FECHA] [HORA] [Debug] myCompleteMessage: ', 4592)

('[FECHA] [HORA] [Debug] Sending (current) ', 4045)

('[FECHA] [HORA] [Debug] Number of bytes read: ', 4592)

('[FECHA] [HORA] [Information] Waiting for client... ', 4599)

('[FECHA] [HORA] [Debug] step 3 ', 4592)

('[FECHA] [HORA] [Debug] step 2 ', 4590)

('[FECHA] [HORA] [Debug] Pending messages en queue (late=False): 1 ', 4045)

('[FECHA] [HORA] [Debug] Sent: ', 4045)

('[FECHA] [HORA] [Debug] Enqueue \* in current queue. ', 4045)

('[FECHA] [HORA] [Debug] Enqued: ', 4590)

('[FECHA] [HORA] [Debug] myData: ', 4592)

**Log cluster apache spark**, Soporte: 5 %

('[FECHA] [HORA] [Debug] Enqueue \*{1,1} in current queue. ', 4045)

('[FECHA] [HORA] [Debug] Number of bytes read: \*{1,1}', 4592)

('[FECHA] [HORA] [Debug] step 2 ', 4590)

('[FECHA] [HORA] [Debug] Pending messages en queue (late=False): 1 ', 4045)

('[FECHA] [HORA] [Debug] myCompleteMessage: \*{1,1}', 4592)

('[FECHA] [HORA] [Debug] Sent: \*{1,1}', 4045)

('[FECHA] [HORA] [Debug] Enqued: \*{1,1}', 4590)

('[FECHA] [HORA] [Information] Client connected \*{1,1} Waiting for data. ', 4597)

('[FECHA] [HORA] [Debug] myData: \*{1,1}', 4592)

('[FECHA] [HORA] [Information] Waiting for client... ', 4599)

('[FECHA] [HORA] [Debug] step 3 ', 4592)

('[FECHA] [HORA] [Debug] Sending (current) \*{1,1}', 4045)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 10 %

('[FECHA] [HORA] [Information] ', 9198)

('[FECHA] [HORA] [Debug] step ', 9182)

('[FECHA] [HORA] [Debug] ', 36726)

**Log cluster apache spark**, Soporte: 10 %

('[FECHA] [HORA] [Information] \*{0,3} Waiting for \*{1,1}', 9196)

('[FECHA] [HORA] [Debug] step \*{1,1}', 9182)

('[FECHA] [HORA] [Debug] \*{2,6}', 36726)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 15 %

('[FECHA] [HORA] [Information] ', 9198)

('[FECHA] [HORA] [Debug] step ', 9182)

('[FECHA] [HORA] [Debug] ', 36726)

**Log cluster apache spark**, Soporte: 15 %

('[FECHA] [HORA] [Information] \*{0,3} Waiting for \*{1,1}', 9196)

('[FECHA] [HORA] [Debug] step \*{1,1}', 9182)

('[FECHA] [HORA] [Debug] \*{2,6}', 36726)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 20 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 20 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 25 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 25 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 30 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 30 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 35 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 35 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 40 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 40 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 45 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 45 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 50 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 50 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 55 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 55 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 60 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 60 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 65 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 65 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 70 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 70 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 75 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 75 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

**Simple logfile clustering tool** (slct\_apache\_spark), Soporte: 80 %

('[FECHA] [HORA] [Debug] ', 45908)

**Log cluster apache spark**, Soporte: 80 %

('[FECHA] [HORA] [Debug] \*{2,6}', 45908)

# **7. Bibliografía**

|  |  |
| --- | --- |
| [1] | Ilias Mavridis y Helen Karatza, «Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark,» *The Journal of Systems and Software,* pp. 133-151, 2017. |
| [2] | T. C. C. A. M. y A. A. K. , «Scalability and Performance of Web Applications in a Compute Cloud,» *IEEE 8th International Conference on e-Business Engineering,* pp. 317-323, 2011. |
| [3] | «Python,» [En línea]. Available: https://www.python.org/. |
| [4] | «ElasticSearch,» [En línea]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html. |
| [5] | «Logstash,» [En línea]. Available: https://www.elastic.co/guide/en/logstash/current/index.html. |
| [6] | «Kibana,» [En línea]. Available: https://www.elastic.co/guide/en/kibana/current/index.html. |
| [7] | «Apache Spark,» [En línea]. Available: https://spark.apache.org/. |
| [8] | H. N. a. A. C. Sanjay Goil, «Mafia efficient and scalable subspace clustering for very large data sets». |
| [9] | R. Varaandi, «A Data Clustering and Pattern Mining,» 2003. |
| [10] | R. V. a. M. Pihelgas, «LogCluster - A Data Clustering and Pattern Minig Algorithm for Wvent Logs,» 2015. |
| [11] | «Github,» [En línea]. Available: https://github.com/villarj/TFM-LogCluster-for-Event-Logs. |
|  |  |