Log clustering tool

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

Contenido

1. Introducción

Hoy en día se genera una gran cantidad de archivos de logs en campos como la industria y la ciencia, ya que estos 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 estos logs no son heterogéneos (1).

Toda esta producción de logs hace, que, debido al tamaño de estos conjuntos de datos, que las soluciones de 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 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 para su procesamiento, almacenamiento, variedad y su gestión 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 surgen 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 tiene que existir computadoras muy potentes para dicho tratamiento, ya que una maquina genera muchos 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 clusterización (que agrupa los logs en grupos y según sus patrón), para llevar a cabo estas tareas se trabajara 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 puede 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 la búsqueda de patrones usando expresiones regulares o hacer graficas de los datos obtenidos o disponibles. Las tres librerías principales que se utilizaran 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.

Solo 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, en nuestro caso hicimos 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 usadas por varias plataformas como puede ser Java, Python, .Net o un navegador con JavaScript, además la información que se almacena es persistente.

A continuación, se muestra un ejemplo de como 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 se inserta de forma directa. Seguidamente se hace una consulta a esta base de datos para ver como se a almacenado 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* y el resultado obtenido ha sido el que se muestra en la figura 1.


```
{
  "took" : 6,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "skipped" : 0,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "trabajo_master",
        "_type" : "doc",
        "_id" : "XdOqGmUB2v_tBaLzpYNl",
        "_score" : 1.0,
        "_source" : {
          "host" : "Javier",
          "@timestamp" : "2018-08-08T17:51:54.939Z",
          "message" : "Jul 26 09:36:29 RUE3 anacron[1102]: Job `cron.daily' terminated\r",
          "@version" : "1"
        }
      }
    ]
  }
}
```


Figura 1. Resultado de como se almaceno 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 un campo, por último destacar que también se añade un campo _id que es un identificador de cada documento Json y que tiene que ser único.

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 realizar esto se hace uso de las funcionalidades disponibles y se escribe la secuencia tal como se muestra en la figura 2.

```
input
{
    stdin { }
}
filter
{
    grok
    {
        match => { "message" =>"(?<time_recive>[A-Za-z]{3}\s*[0-9]{2}\s*\S*)"}
    }
}

output
{
    elasticsearch
    {
        index => "trabajo_master"
    }
}
```

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 este se le aplica un filtro que almacena en la variable time_recive una fecha y hora tras hacer uso de una expresión regular y por último 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 salvo porque el _id ha cambiado (recordemos que es un identificador único) y porque se ha añadió un campo time_recive que contiene la información parseada, esto se puede apreciar en la figura 3.


```
{
  "_index" : "trabajo_master",
  "_type" : "doc",
  "_id" : "dZe1H2UBj2d1dmRNKp_9",
  "_score" : 1.0,
  "_source" : {
    "host" : "Javier",
    "@timestamp" : "2018-08-09T17:21:31.082Z",
    "@version" : "1",
    "time_recive" : "Jul 26 09:36:29",
    "message" : "Jul 26 09:36:29 RUE3 anacron[1102]: Job `cron.daily' terminated\r"
  }
}
```


Figura 3. Resultado de como se almaceno 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, 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, todo esto se puede ver en la figura 4.

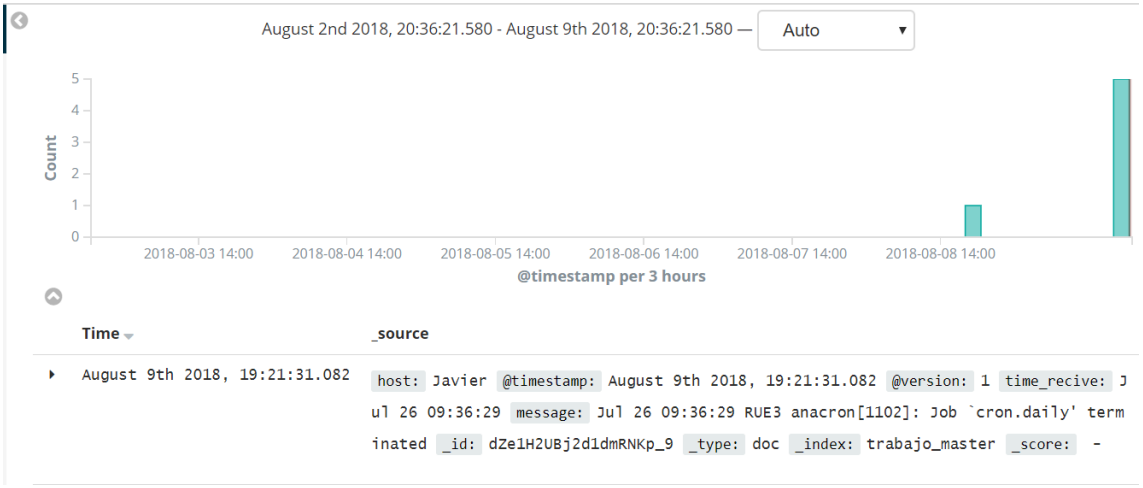

Figura 4. Dashboard que contiene dos visualizaciones sobre los logs.

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 que 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 (este proyecto usa la 2.3.0) .

Las aplicaciones que usan Spark son realizadas de forma independientes y son coordinadas por el objeto SparkContext que se encuentra en el programa principal y que es capaz de conectarse a gestores de cluster 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 primero **n** elementos del RDD y por último esta 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 mas probable es que los datos se encuentren en mas de un RDD, estos son:
    - **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 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.

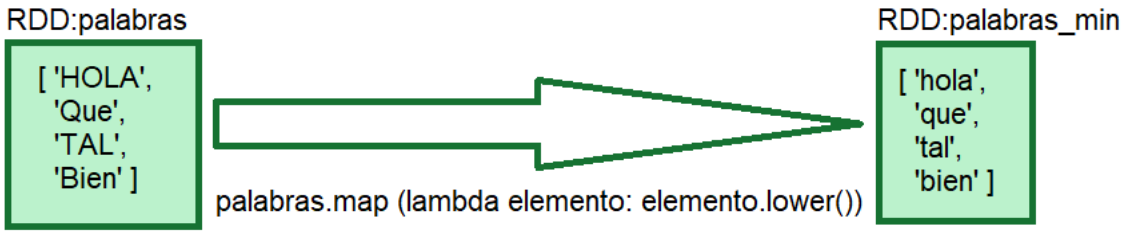

Figura 5. Ejemplo de funcionamiento de la transformación map(func) en un RDD primario.

- **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.

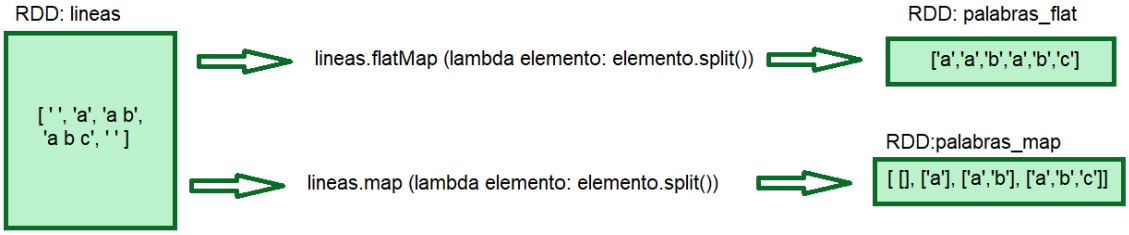

Figura 6. Ejemplo de funcionamiento de la transformación flatMap(func) y map(fun) en un RDD para comparar el funcionamiento de ambas.

- **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.

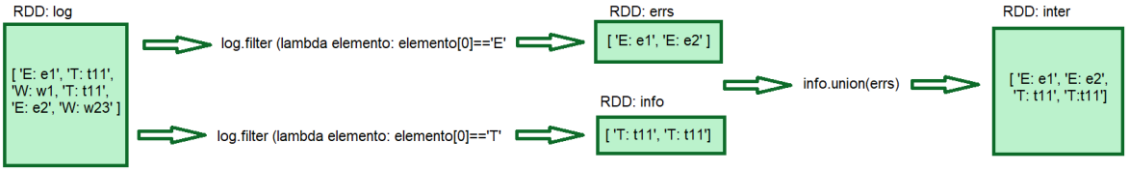

Figura 7. Ejemplo de funcionamiento de los métodos filter y union para transformar un RDD.

- **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.

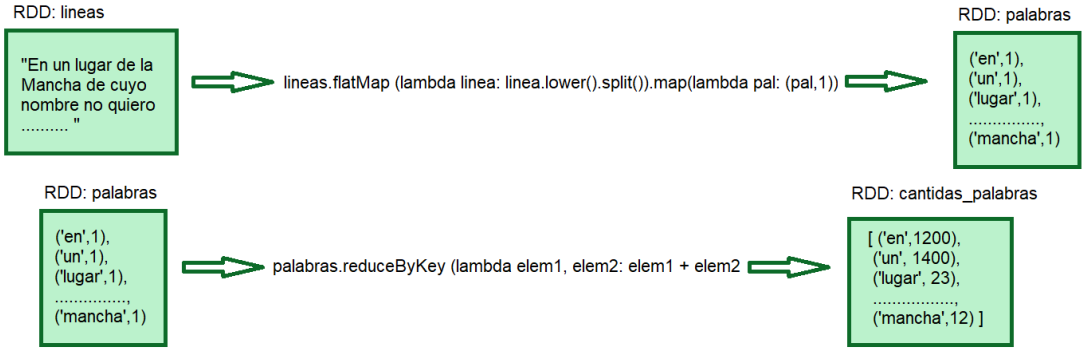

Figura 8. Ejemplo uso del método reduceByKey() para contar palabras.

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

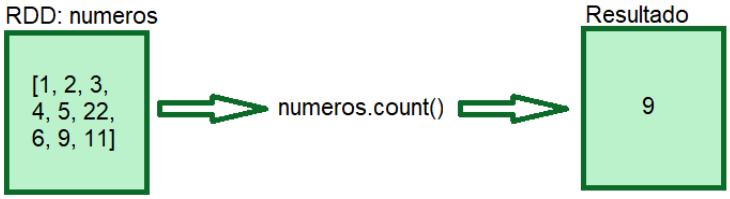

Figura 9. Ejemplo del método count() sobre un RDD.

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

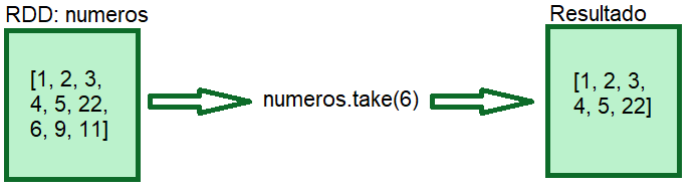

Figura 10. Ejemplo de aplicar la acción take(6) sobre un RDD.

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

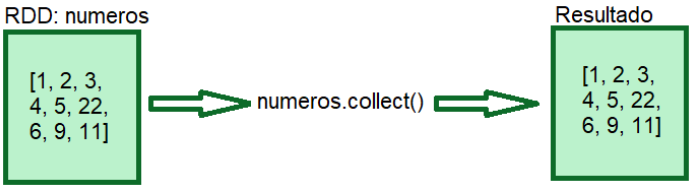

Figura 11. Ejemplo de aplicar la acción collect() sobre un RDD.

Para finalizar hay que destacar una de las características más importantes de Apache Spark y es la persistencia o cacheo de un dataset en memoria, por lo 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, esta se recalcula automáticamente utilizando las transformaciones que lo crearon originalmente.

2.6 Metodología.

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

Para comenzar, se implementan dos algoritmos diferentes que hacen clusteres con los logs y de esta forma comprobar los resultados obtenidos en función de cada algoritmo, además cada algoritmo se implementa de dos formas diferentes, una primera forma utilizando el lenguaje Python y Apache Spark y una segunda forma utilizando solamente el lenguaje Python, con lo que conseguimos comparar el tiempo de ejecución entre los dos algoritmos y las dos implementaciones diferentes de cada algoritmo.

Para finalizar, se determinarán los modelos de cada cluster que en este caso serán expresiones regulares de cada cluster para que cuando venga un nuevo log clasificar este log en un cluster de nuestro modelo o bien como outliers 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 cluster por si misma, por lo que podemos definir el soporte como el mínimo número de líneas que cumplen un cierto formato para que este cluster 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 trabajo, 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 $L = \{l_1, \dots, l_n\}$ un registro de eventos textuales que consiste en n líneas y cada línea l_i $(1 \leq i \leq n)$ es una representación completa de algún evento e i es un identificador único de cada línea. Suponiendo que cada línea $l_i \in L$ es una secuencia de palabras k_i de forma que $l_i = (w_{i,1}, \dots, w_{i,k_i})$. LogCluster y SLCT toman el umbral de soporte s $(1 \leq s \leq n)$ como un parámetro de entrada dado por el usuario y divide las líneas de registro de los eventos en grupos (clusters) $C_1, \dots, C_m$ de forma que hay al menos s líneas

en cada cluster y el resto de líneas serán consideradas Outliers con lo que se puede escribir el registro de eventos como, $L = C_1 \cup \ldots \cup C_m \cup O$.

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 o bien otro tipo diferente donde se almacena la información y el otro parámetro de entrada es 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 cluster, por último, el tercer paso consiste en seleccionar los clusters 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, que 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 pasada 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. 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 $(i_1, v_1), \ldots, (i_m, v_m)$ entonces el candidato a clúster tendrá m atributos fijos $\{(i_1, v_1), \ldots, (i_m, v_m)\}$. 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 son propuestas

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 $s = 2$ 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 $s = 3$ 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 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 o bien otro tipo diferente donde se almacena la información y el otro parámetro de entrada es el soporte.

LogCluster ve el problema de clusterización de log como un problema de minería de patrones, cada clúster C_j se identifica de forma única por el patrón de línea p_j que coinciden 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 encontrara 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 examinara cada palabra sin su posición en la línea del evento del log y considerara 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 se incrementa en uno su valor y se 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 establece en **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 al usuario de los candidatos restantes como clusteres.

A continuación, se muestra un ejemplo del funcionamiento de este algoritmo para lo cual definimos el soporte como $s = 3$ 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} modificando el primer patrón asociado *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}* y por ultimo recibe la 4 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 *Interface *{1,2} down *{2,2}* de forma que al finalizar hay dos candidatos a cluster que son:

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

*Interface *{2,2}*

Con soporte 3 y 1 respectivamente, por lo que tras aplicar la condición de que para ser considerado cluster el soporte tiene que ser 3 o más el algoritmo da como resultado final un único cluster el cual es:

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

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

*Interface *{2,4}*

4. Implementación.

En esta sección se explica como 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 debemos definir qué significa el término log en el campo de la informática. Un log no es mas 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 partícula (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, segundo) en el que ocurrió la acción que genero dicho log con lo que permite 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, si por ejemplo una aplicación que cambia las contraseñas de cada usuario una vez al mes 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 seria 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 líneas (la fecha y la hora serán diferentes) serian clasificadas en un cluster ya existente, no siendo depositadas como outlier.

Tras lo explicado anteriormente el ciclo del programa implementado será un preprocesado de los datos para posteriormente aplicar el algoritmo que los clusteriza y por último obtener los modelos tal como se muestra en la figura 12.

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 clusterizar.

Para comenzar con el diseño del programa se creará un modulo llamado ExpresionesRegulares.py el cual se utilizará para preprocesar los datos tal y como se comentó anteriormente, de forma que se pueda añadir todas 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 tenemos el modulo del preprocesado pasamos a implementar el algoritmo LogCluster usando apache spark y otra de forma secuencial y el algoritmo Slct utilizando apache Spark de forma que podemos comparar los resultados obtenidos entre algoritmos.

4.1 Implementación del módulo para el Algoritmo LogCluster secuencial.

A continuación, procedemos a describir como se ha implementado el algoritmo LogCluster de forma secuencial incorporando un ejemplo que clarifica su ejecución. Para comenzar mostramos el pseudocodigo de la implementación realizada.

```
1: Import modulo time
2: Import modulo preprocesado
3: Inputs: logs, s ∈ [0,100], 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)
```

```
clusterCandidatos = [ ]
Mientras conexión logs abierta
    Para cada línea de logs
        Si preprocesado == True
            línea = línea.preprocesada
        tupla = [ ]
        varrs = [ ]
        i = 0
        v = 0
        palabras = línea.split()
        Para cada palabra en palabras
            Si palabra en palabrasFrecuentes
                tupla.append(palabra)
                varrs.append(v)
                i=+1
                v = 0
            Else:
                v=+1
        varrs.apprend(v)
        k=len(tupla)
        Si k>0
            existe=False
            Para cada j en rango(len(clusterCandidatos))
                Si (clusterCandidatos[ j ][‘tupla’]=tupla
                    cluster=clusterCandidatos[ j ]
                    cluster[‘soporte’]=+1
                    clusterCandidatos[ j ] = cluster
                    Para cada t in rango (k+1)
                        Si cluster[‘minimo’][ t ] > varrs[ t ]
                            cluster[‘minimo’][ t ] = varrs[ t ]
                        Si cluster[‘maximo][ t ] < varrs[ t ]
                            cluster[‘maximo][ t ] = varrs[ t ]
                    existe = True
                    break
            Si existe == False
                varmin = [ ]
                varmax = [ ]
```

```
61:                     Para cada j en rango (k+1)
62:                         varmin.append(varrs[ j ])
63:                         varmax.append (varrs[ j ])
64:                     cluster={‘tupla’:tupla,’soporte’:1,’minimo’:varmin,’maximo’:varmax}
65: Cerrar conexión logs
66: Para cada i en rango (len(clusterCandidatos))
67:     patron=""
68:     cluster = clusterCandidatos[ i ]
69:     k = len(cluster[‘tupla’])
70:     Para cada j en rango(k)
71:         Si cluster[‘maximo’][ j ] > 0
72:             mínimo = cluster[‘minimo’][ j ]
73:             máximo = cluster[‘maximo’][ j ]
74:             patron = patron + "*{" + mínimo + "," + máximo + " }"
75:         patron = patron + cluster[‘tupla’][ j ] + " "
76:     Si cluster[‘maximo’][ k ] > 0
77:         mínimo = cluster[‘minimo’][ k ]
78:         máximo = clister[‘maximo’][ k ]
79:         patron = patron + "*{" + mínimo + "," + máximo + "}"
80:     cluster[‘patron’] = patron
81:     clusterCandidatos[ i ] = cluster
82: numCandidatosCluster = len (clusterCandidatos) #candidatos a cluster
83: patrones = [ ]
84: soporteLineas = [ ]
85: lineasClusterizadas = 0
86: numClusterFinales = 0
87: Para cada i en rango (len(clusterCandidatos))
88:     Si clusterCandidatos[ i ][‘soporte’] > soporte
89:         patrones.append(clusterCandidatos[ i ][‘patron’])
90:         soporteLineas.append(clusterCandidatos[ i ][‘soporte’]
91:         lineasClusterizadas=lineasClusterizadas+clusterCandidatos[i][‘soporte’]
92:         numClusterFinales= +1
93: tiempoFinal=time()
94: tiempoEmpleado = tiempoFinal – tiempoInicial
95:     resultado={‘Patrones’:patrones,‘SoporteLineas’:soporteLineas,‘LineasClusterizadas’:
lineasClusterizadas,‘NumCandidatosCluster’:numCandidatosCluster,  ‘NumFinalesCluster’:
numClusterFinales,’TiempoEmpleado’:tiempoEmpleado}
```

A la vista del seudocódigo anterior se puede ver la implementación de este en 3 pasos diferentes.

Un primer paso que va desde la línea 6 hasta la 23 en el que se hace un preprocesado de las líneas de logs si el usuario final quiere y se cuentan las palabras que aparecen en el archivo a analizar para finalmente solamente aceptar las que tienen un soporte igual o superior al dado por el usuario.

Un segundo paso que va desde la línea 24 hasta la línea 81 en el que se obtienen los candidatos a cluster con sus patrones y soportes.

Y un ultimo tercer paso que va desde la línea 82 hasta la 95 en el que se filtra los candidatos a cluster por el soporte y se obtienen otros resultados interesantes como son los patrones, el numero de candidatos, el numero de cluster totales o el tiempo empleado en obtener el resultado.

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 a la función serán los logs, $s \in [0,100]$, 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.

Para comenzar se utiliza la función map de apache spark al 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 pasamos a contar las palabras para lo cual primero hay que separarlas para posteriormente añadir un 1 y luego sumar las palabras. Para contar las palabras se hace uso de la función flatMap y se le aplica la función Split a cada línea y posteriormente se le aplica otra transformación map de forma que añada un uno a cada línea tal como se muestra:

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

De forma que se si se aplica la acción collect se obtienen todas las palabras en un array que esta formado por tuplan con 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 sumara las palabras.

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

Con lo cual se sumarán las palabras 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 filtrar para rechazar el resto de palabras que son inferiores al 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 mas veces que el umbral y la cantidad de veces que aparecen.

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

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

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

De esta forma ya tendríamos implementado la primera parte del algoritmo que es la de contar las palabras.

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

Una vez se han determinado las palabras frecuentes se pasa a la segunda parte del algoritmo en donde se determina las palabras frecuentes que hay en cada línea así como cuantas palabras hay entre cada palabra frecuente, para lo cual hay que crear

una función a la que se la pasa como parámetros las palabras frecuentes y la línea de log y esta función devolverá las palabras frecuentes que hay en cada línea así como el numero de palabras que hay entre ellas, para realizar esto se utiliza la transformación flatMap de apache spark tal como se muestra.

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

Y vemos que se obtiene un array compuesto por tuplas y estas tuplas a su vez están compuestas por otras dos una que almacena las palabras y la otra que almacena el numero 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 hay que aplicar una transformación reduceByKey donde se reduce mediante las tuplas de palabras y además hay que pasar como parámetro una función que determina el mínimo y el máximo de palabras que hay entre cada palabra frecuente de forma que:

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

Si hacemos un collect de la transformación anterior vemos que el array se ha transformado a un array 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 han aparecido.

[(('[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 cluster ahora basta con aplicar un filtro para aceptar aquellos candidatos con el soporte mayor o igual que el introducido por el usuario, por lo que en este caso tenemos que hay únicamente un cluster candidato.

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

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

['[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 como 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 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 en el que 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 se aplica 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 y para finalizar se aplica un filtro para solamente aceptar las palabras que aparezcan mas 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 cluster, para esto se usa 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 con las palabras con su posición para cada línea tal como se muestra.

[((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 para cada campo y posteriormente se aplica un groupByKey que agrupara 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 obtienen que los dos candidatos a cluster pasan a ser cluster finales y 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 cluster 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 cual de los dos algoritmos se han obtenido los patrones.

- **Algoritmo Slct:**
  - ✓ Cuando aparece un * en el patrón quiere decir que hay 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 numero 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 serian 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 los patrones obtenidos 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 y luego un conjunto de espacio en blanco con caracteres cualesquiera juntos pudiendo aparecer otra vez esto posteriormente debe encontrarse la palabra down pudiendo aparecer dos palabras más.

Por otra parte, puede que el algoritmo encuentre patrones mas generales que engloban a otros más específicos con lo que una nueva línea de logs podría pertenecer a mas de un patrón, por lo que quizá si queremos que una línea pertenezca a una única línea hay que dar un peso a cada patrón para obtener los que mas información contienen como mas 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 que carga esta información, a continuación, se muestra el archivo de configuración utilizado.

Este fichero consta de tres partes, dos obligatoria (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 además hay que indicar el índice para que se guarde y el host donde se encuentra corriendo.

```
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"
        }

   stdout { codec => rubydebug }
}
```

Ya solo falta crear o configurar un contexto de Apache Spark con esto se consigue 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 quiere 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 spart directamente sin tener que almacenar la información que contiene la base de datos en ninguna variable extra, para lo cual se usa la siguiente configuración para establecer la conexión.

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 común 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'

Y ya solo queda utilizar un Rdd para los datos almacenados y para lo cual se hace uso de la documentación de elastic y 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 analizar 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 cluster 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 cluster.
- Tiempo empleado: Es el tiempo total que el algoritmo ha estado ejecutándose para obtener los clústeres finales.
- Numero de candidatos a cluster: es el número de todos los cluster que se han formado utilizando el algoritmo.
- Numero de cluster finales: son los cluster finales tras aplicar el valor de soporte.
- Porcentaje de líneas clusterizadas: es el tanto por ciento del total de líneas que pertenecen a algún cluster.

5. Casos 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. Y para realizar estas pruebas se utilizará una maquina con Ubuntu 16.04 y un único nodo y se proba en un fichero contendrá solamente 2633 líneas de un sistema operativo y el otro contendrá 55116 líneas que tendrá información sobre él 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 veamos a continuación cuanto tarda en ejecutarse el algoritmo LogCluster y el Slct.

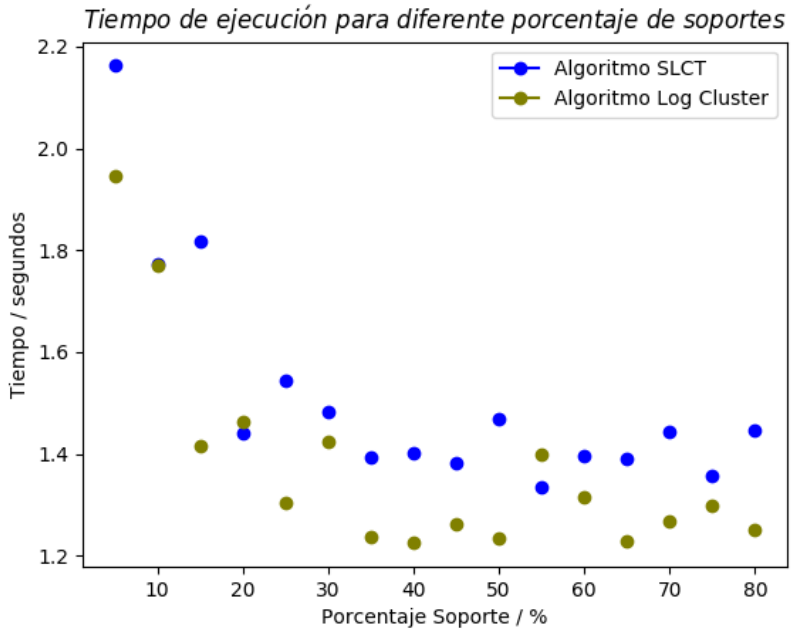

Figura 12. Tiempo de ejecución para diferentes porcentajes de soportes.

En la figura 12 puede verse que los tiempos de ejecución son muy parecidos entre sí y como según estos avanzan va bajando dicho tiempo de ejecución, 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 vemos 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 mas en crear todos los RDD que en ejecutarse.

Para continuar con el análisis examinamos cuantos candidatos a cluster obtiene cada algoritmo y para esto podemos ver la figura 13 que muestra el numero de candidatos a cluster en función del porcentaje de soporte y como se puede ver el algoritmo LogCluster siempre propone mas candidatos a cluster 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 tienen en cuenta su posición también seguramente fue rechazada como palabra frecuente. Además, se puede ver como para un soporte desde el 15% hasta el 70 % se obtiene el mismo número de clúster candidatos, esto quiere decir que no se están perdiendo candidatos a cluster al aumentar el soporte.

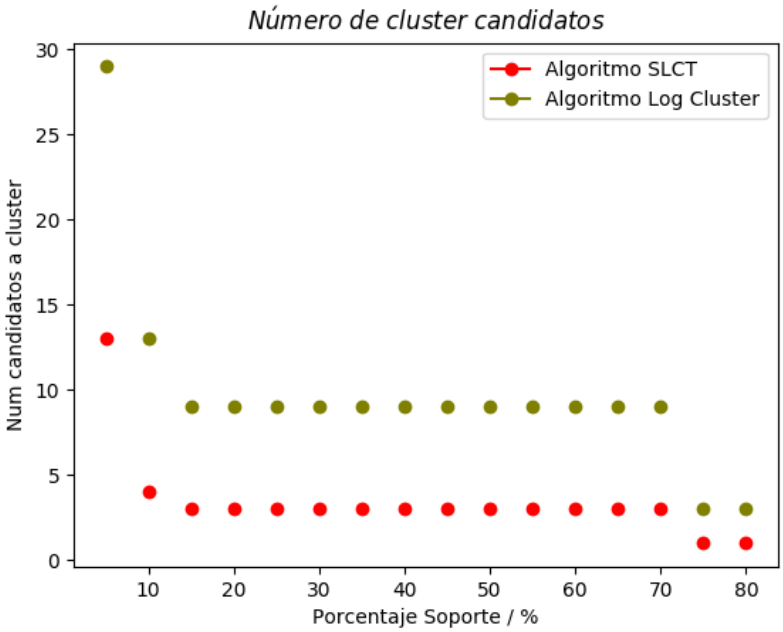

Figura 13. Número de candidatos a clúster para diferentes porcentajes de soportes

Pero lo que en verdad recibe el usuario final son los cluster que superan el soporte, estos se pueden ver en la figura 14, a la vista de esta figura puede apreciarse como el algoritmo SLCT siempre ofrece al menos un cluster sin embargo el algoritmo Log Cluster no siempre como puede verse para un soporte del 70% esto se debe simplemente al soporte ya que ningún candidato a cluster ha sido aceptado por lo que se rechazan.

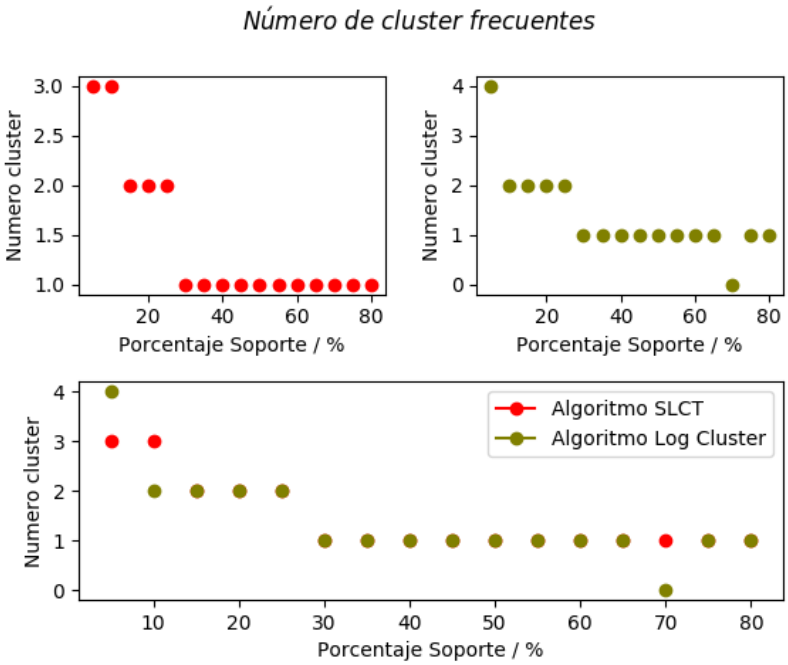

Figura 14. Número de clúster finales para diferentes porcentajes de soportes.

Para finalizar con la comparación vamos a mostrar en la figura 15 el porcentaje de líneas que pertenecen a algún cluster y que no serian consideradas outlier. Como puede apreciarse en dicha figura 15 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 cluster finales para todos los soportes llegando este algoritmo incluso a asignar todas las líneas.

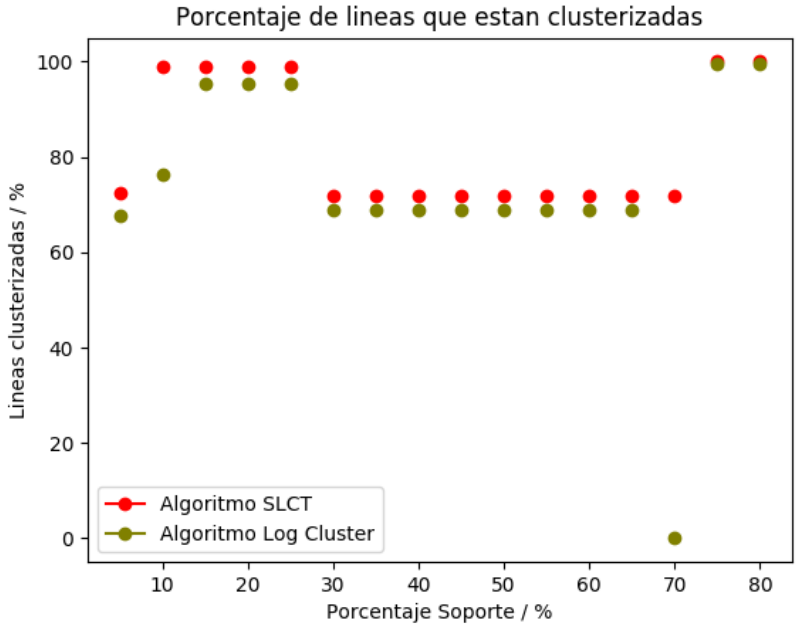

Figura 15. Porcentaje de líneas clusterizadas para diferentes porcentajes de soportes.

Para continuar con nuestro análisis vamos a comparar el tiempo de ejecución para las dos implementaciones realizadas para el algoritmo log cluster, este análisis se puede ver en la figura 16 que muestra el tiempo de ejecución del algoritmo para ambos casos y como puede apreciarse el algoritmo log cluster secuencial tiene un tiempo de ejecución prácticamente lineal mientras que utilizando apache spark tiene saltos, esto se debe prácticamente a que en el log cluster 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 mas que en otros, pero como puede verse cuando se tiene un archivo con pocas líneas da igual que implementación usar porque apenas hay diferencia.

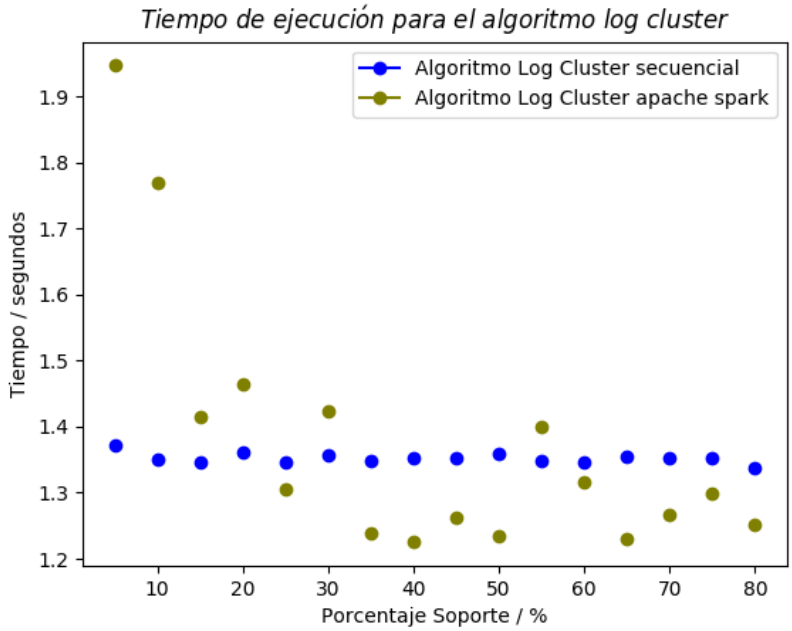

Figura 16. Tiempo de ejecución para dos implementaciones diferentes del algoritmo log clúster para diferentes porcentajes de soportes.

Para finalizar el análisis nos fijamos en los patrones retornados y en el número de líneas que siguen dicho patrón y a partir de toda esta información intentamos concluir con que soporte y por consiguiente patrón deberíamos quedarnos para obtener nuestro modelo.

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)

A la vista de los patrones presentados para ambos algoritmos desde un soporte del 5% hasta del 80% podemos ver como prácticamente ambos algoritmos encuentran los mismos patrones salvo que el log cluster indica las palabras que van al final. Por otra parte, también vemos que hay patrones que pueden estar dentro de otros patrones mas generales como puede ser para el caso de soporte 25% para el algoritmo log cluster donde hay dos patrones '[FECHA] [HORA] RUE3 *{1,21}' y '[FECHA] [HORA] RUE3 kernel: [*{2,23}', en los cuales se puede ver como el primer patrón engloba a la mayoría de las líneas del segundo patrón.

Por lo que analizando todo lo anterior se cree que el mejor soporte para el algoritmo log clúster es el del 25%, al igual que para el algoritmo slct. Una vez se ha tomado esta decisión se obtienen los modelos para esos patrones (recordemos que los modelos no son mas que las expresiones regulares que cumplen dichos patrones). Para ambos se obtienen los siguientes modelos:

Modelos para los patrones obtenidos con el algoritmo LogCluster y con soporte 25%

Patrón:
[FECHA] [HORA] RUE3 *{1,21}
Modelo:
^\s*\[FECHA\]\s*\[HORA\]\s*RUE3(\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*)?

Ahora cada línea nueva que llegue se le pasa por este modelo y sino cumple ninguno de los dos patrones se clasifica como un outlier.

Modelos para los patrones obtenidos con el algoritmo Slct y con soporte 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 aun 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 mas tarde de lo debido ya que se ha 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 cuando no cumpla este se puede 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 55116 líneas.

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

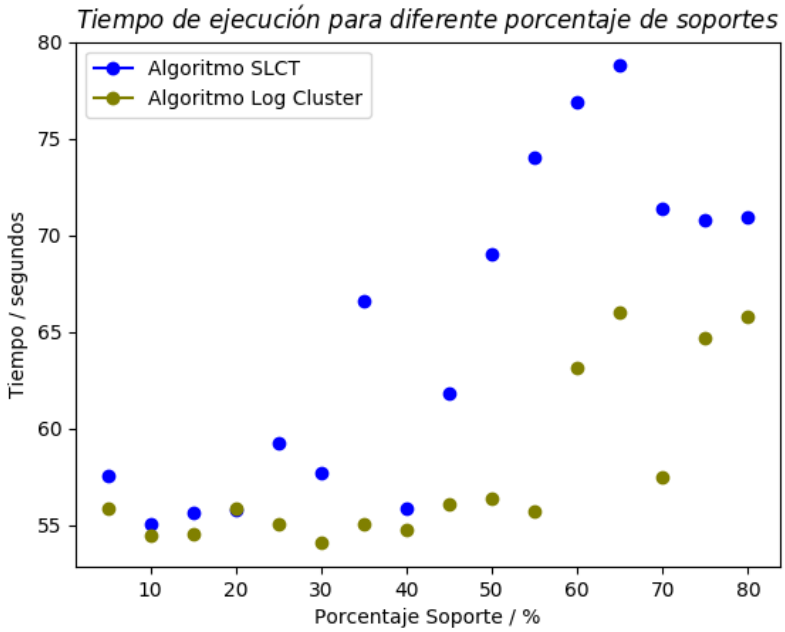

Figura 17. Tiempo de ejecución para diferentes porcentajes de soportes.

En la figura 17 puede apreciarse el tiempo de ejecución de los dos algoritmos implementados en esta ocasión se puede ver como se ha tardado mas en analizar todos los datos disponibles que para el caso de uso anterior, y además también vemos como el algoritmo SLCT en general tarda mas en ejecutarse completamente que el log cluster 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 log cluster en líneas generales tardaría entre 55 y 60 segundo en devolver un resultado, mientras el algoritmo SLCT no tiene un término medio.

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

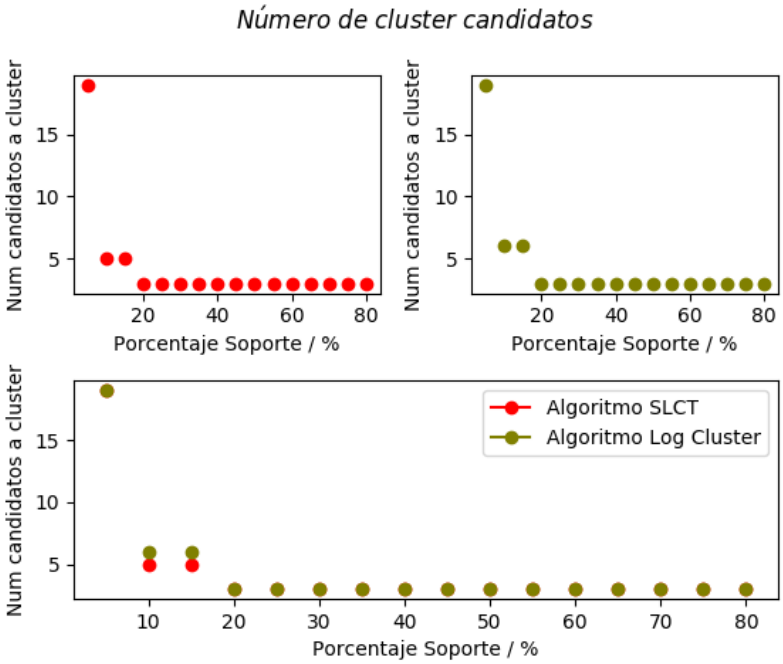

Figura 18. Número de candidatos a clúster para diferentes porcentajes de soportes

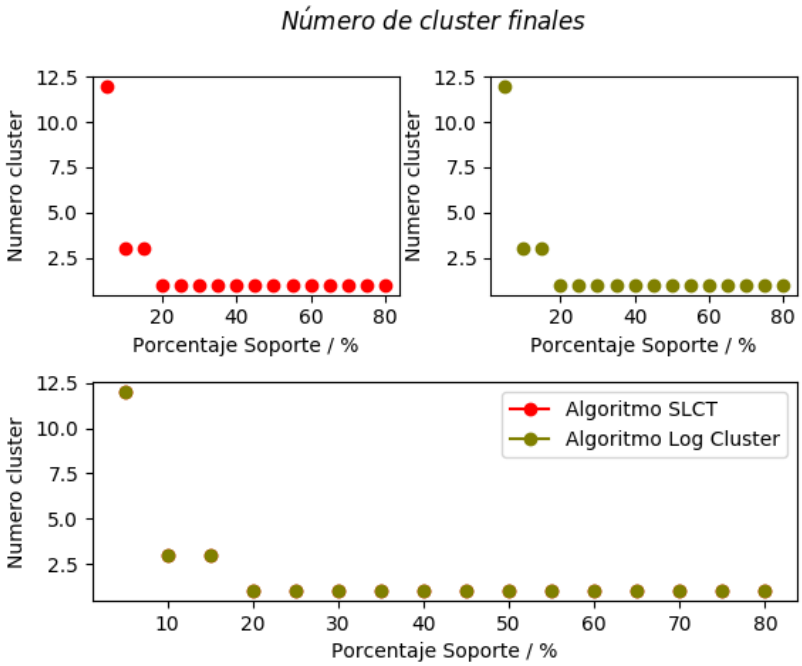

Figura 19. Número de clúster finales para diferentes porcentajes de soportes

En la figura 19 se puede ver el numero final de cluster y como puede apreciarse a partir de un soporte del 20% ambos algoritmos ofrecen un único cluster final como solución final con lo que se cumplen las sospechas descritas anteriormente de que a partir de dicho soporte no se pierde información, por ultimo cabe señalar como el algoritmo log cluster a pesar de tener mas candidatos a cluster siempre acaba teniendo el mismo numero de cluster finales que el algoritmo SLCT.

Para tomar una decisión final, hay que ver que 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.

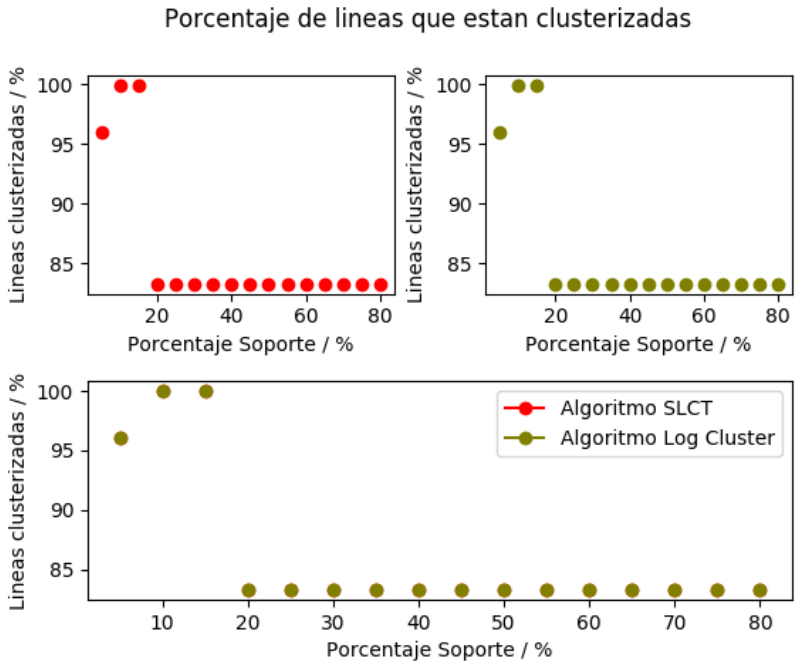

Figura 20. Porcentaje de líneas clusterizadas para diferentes porcentajes de soportes.

A la vista de los resultados mostrados en la figura 20 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 mas líneas. Lo que si que se ve es que ambos algoritmos consiguen distribuir en los diferentes clusteres 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 log cluster aparecerán el número de palabras finales y en el slct no.

Tras realizar todo este análisis nos queda preguntarnos si de verdad con apache spark para el algoritmo log cluster se gana tiempo respecto a una implementación clásica, ya se ha visto en la figura 16 que para pocas líneas es más rápido no hacer una implementación usando apache spark, pero para este caso en el que tenemos mas de 55.000 líneas quizás si que sea mas veloz utilizar apache spark, esto lo comprobamos en la figura 21 en la que se muestra el tiempo de ejecución para ambas implementaciones del algoritmo log cluster 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 esta ejecutando para un único nodo, con esto se hace hincapié en que si hubiese mas 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 numero de líneas ya que en realidad se suelen tener muchas mas líneas de las que se han usado para este trabajo.

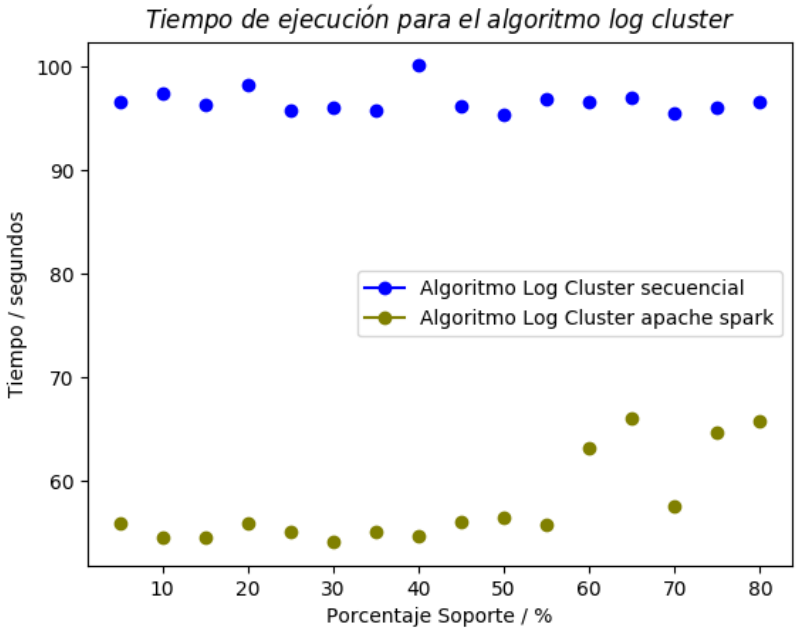

Figura 21. Tiempo de ejecución para dos implementaciones diferentes del algoritmo log clúster para diferentes porcentajes de soportes.

Finalizamos el análisis mostrando los patrones retornados y el número de líneas que siguen dicho patrón y a partir de toda la información anterior concluiremos con que soporte y por consiguiente patrón es mejor quedarnos para obtener un modelo que verifique nuevas líneas de información.

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)

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 y no serán tan generales como lo son para soportes superiores.

Se obtienen los siguientes modelos:

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*\(late=Fa
lse\):\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*d
ata\.
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*\(late=Fa
lse\):\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 log cluster es mas 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 quedara etiquetada como outlier.

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 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 mas importante que soporte utilizar ya que esta decisión es crítica ya que se pueden obtener patrones/modelos muy generales o en caso contrario muy representativos es importante buscar un termino 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 cluster mas general y otros más específicos que para soportes mayores pueden estar dentro de este cluster general, la idea es asignar pesos a los patrones para que una línea nueva sea solo asignada a un único cluster.

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.

7. Bibliografía

1. *Performance evaluation of cloud-based log file analysis with Apache Hadoop and Apache Spark.* **Ilias Mavridis y Helen Karatza.** 2017, The Journal of Systems and Software, págs. 133-151.

2. *Scalability and Performance of Web Applications in a Compute Cloud.* **T. C. Chieu, A. Mohindra y A. A. Karve.** Beijing : s.n., 2011, IEEE 8th International Conference on e-Business Engineering, págs. 317-323.

3. **Python. [En línea] https://www.python.org/.**

4. ElasticSearch. [En línea] https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html.

5. Logstash. [En línea] https://www.elastic.co/guide/en/logstash/current/index.html.

6. Kibana. [En línea] https://www.elastic.co/guide/en/kibana/current/index.html.

7. Apache Spark. [En línea] https://spark.apache.org/.

8. *Mafia efficient and scalable subspace clustering for very large data sets.* Sanjay Goil, Harsha Nagesh and Alok Choundhary.

9. *A Data Clustering and Pattern Mining.* Varaandi, Risto. TUT Centre for Digital Forensics and Cyber Security : s.n., 2003.

10. *LogCluster - A Data Clustering and Pattern Minig Algorithm for Wvent Logs.* Pihelgas, Risto Vaarandi and Mauno. TUT Centre for Digital Forensics and Cyber Security : s.n., 2015.

11. Github. [En línea] https://github.com/villarj/TFM-LogCluster-for-Event-Logs.