Log clustering tool

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 esta 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.

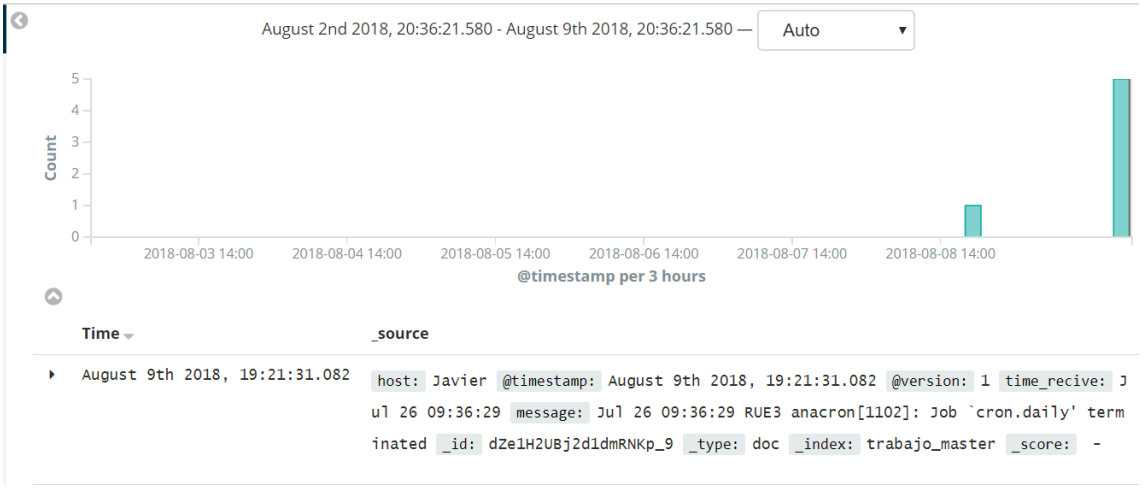

Figura 4. Dashboard que contiene dos visualizaciones sobre los logs.

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.

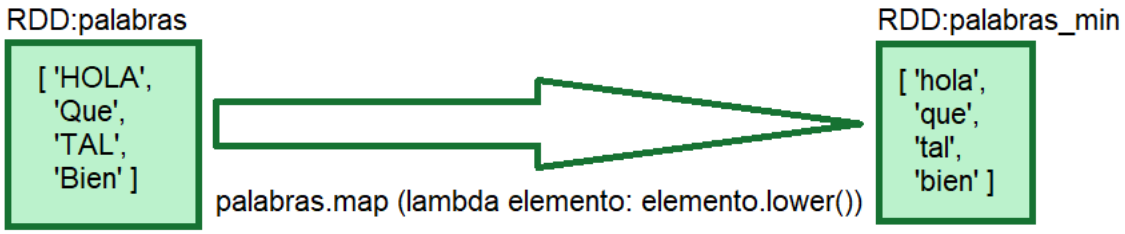

Figura 5. Ejemplo de funcionamiento de la transformación map(func) en un RDD primario.

- **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.

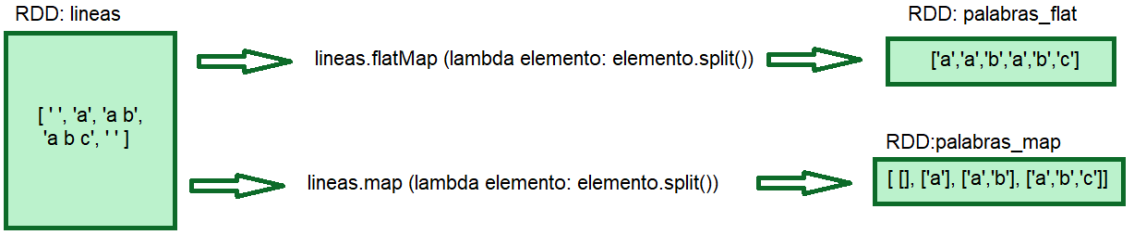

Figura 6. Ejemplo de funcionamiento de la transformación flatMap(func) y map(fun) en un RDD para comparar el funcionamiento de ambas.

- **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.

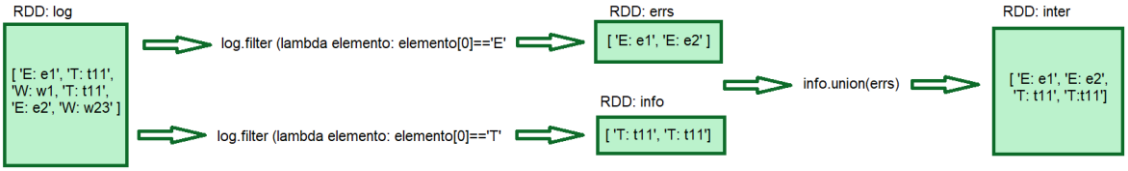

Figura 7. Ejemplo de funcionamiento de los métodos filter y union para transformar un RDD.

- **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.

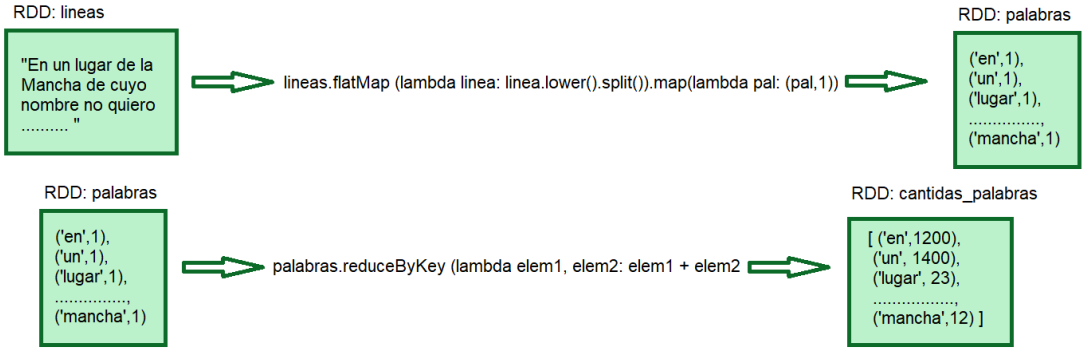

Figura 8. Ejemplo uso del método reduceByKey() para contar palabras.

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

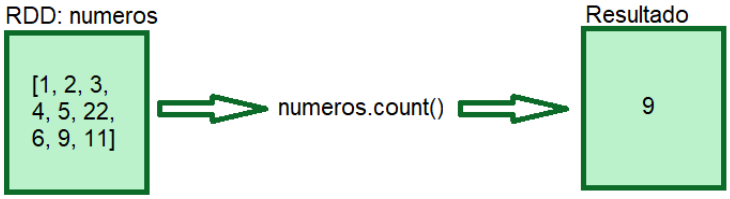

Figura 9. Ejemplo del método count() sobre un RDD.

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

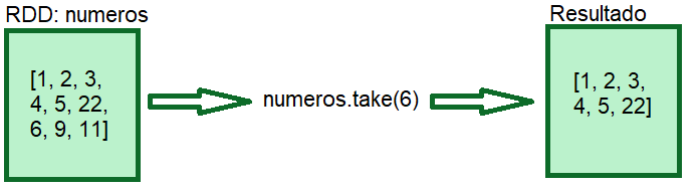

Figura 10. Ejemplo de aplicar la acción take(6) sobre un RDD.

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

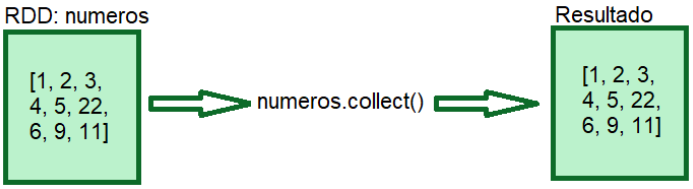

Figura 11. Ejemplo de aplicar la acción collect() sobre un RDD.

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)
```

24: clusterCandidatos = [ ]

aaa

Tras analizar otros algoritmos de este _ambito, procedemos a describir nuestro algoritmo de forma m_as profunda e incorporaremos ejemplos que clari_quen el contexto. Empezaremos viendo el pseudoc_odigo de nuestro algoritmo y, a continuaci_on, veremos qu_e tiene en com_un con los otros algoritmos descritos y cu_ales son las diferencias.

aa

`pyspark`

Una expresión regular (o RE) especifica un conjunto de cadenas que coincide con él; las funciones de este módulo le permiten verificar si una cadena en particular coincide con una expresión regular dada (o si una expresión regular dada coincide con una cadena en particular, que se reduce a la misma cosa).

`matplotlib`

generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy.

e Python que produce figuras de calidad de publicación en una variedad de formatos impresos y entornos interactivos en todas las plataformas

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