Programa Experto en Big Data

DMP Real Time con Spark Streaming y Drools

David Oñoro Navarro

Índice de contenido

1 Introducción

1.1 Motivación del proyecto

Este proyecto se enmarca dentro del mundo de la publicidad en internet, concretamente en el ámbito del Real Time Bidding. Es una técnica publicitaria está empezando a despegar en Europa con números muy positivos especialmente en Reino Unido, Alemania y Francia.

El RTB consiste en una subasta en tiempo real de los distintos espacios publicitarios, basada en los datos conocidos de navegación de los usuarios. De esta manera se puede aumentar o desechar una puja para determinada campaña en función del usuario que accede.

Para configurar estas campañas de publicidad se utilizan unos sistemas que reciben el nombre de *Demand Side Platform* o DSP. Estas plataformas son las encargadas de que un anuncio llegue en tiempo real a un público de mayor calidad, definiendo además el precio de cada impresión en función del usuario. Con el fin de obtener un mayor impacto, es necesario disponer además de un buen perfilado de los usuarios, que es para lo que se utilizan los DMP o *Data Management Platform*.

Un DMP procesa y almacena la información de tal manera que pueda ser utilizada por los DSP. En concreto, se ocupa de la recopilación y gestión centralizada de cookies de diferentes fuentes para su posterior análisis. Una vez agregados los datos se crean segmentos que permitan la optimización de las compras, que son ofrecidos a los DSP.

El objetivo de este proyecto es el de construir un DMP a partir de datos de navegación de usuarios, creando categorías que pudieran ser de interés para un DSP. Sin embargo, uno de los principales problemas de los DMP es que los datos no suelen estar suficientemente actualizados, por lo que la motivación principal de la aplicación será la de obtener un valor muy bajo de recencia, siendo capaz de categorizar al momento a los usuarios.

1.2 Alcance

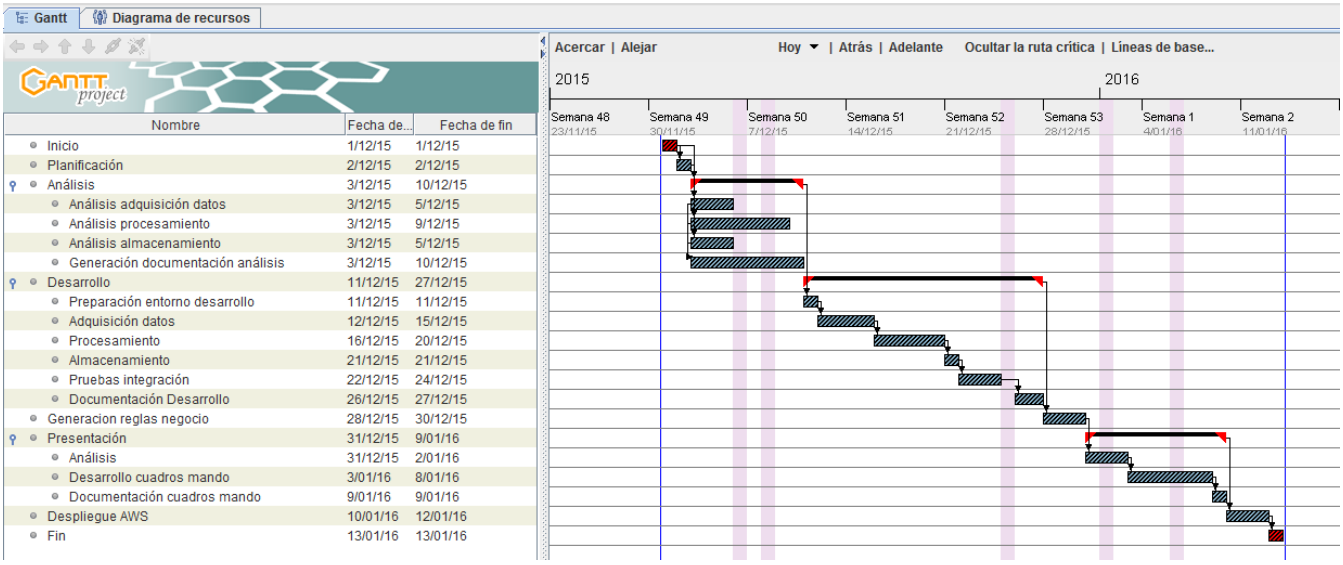
El alcance del presente proyecto es el de crear un sistema DMP capaz de perfilar a los usuarios según su navegación y venderlos lo antes posible. Para ello se partirá de un dataset con navegaciones con datos como el identificador de usuario, su ip, el país desde el que se accede, el dominio y la url concreta a las que se accede, la fecha, etc.

Se pretende crear una arquitectura escalable, capaz de procesar un flujo continuo de información, y que permita a un cliente consultar en qué está interesado determinado usuario para poder ofrecer al momento la publicidad que pueda causar mayor impacto. Esta arquitectura permitirá:

- Ingestar datos en un sistema de colas.
- Procesamiento en streaming los datos de navegación.
- Categorizar al vuelo a los usuarios mediante un repositorio de expresiones regulares.
- Por último, almacenar los resultados en un api que pueda ser consultado por un DMP.

2 Planificación

Para el desarrollo de este proyecto se han identificado las siguientes tareas, que se realizarán según la planificación adjunta:



Nombre	Fecha de...	Fecha de fin
Inicio	1/12/15	1/12/15
Planificación	2/12/15	2/12/15
Análisis	3/12/15	10/12/15
Análisis adquisición datos	3/12/15	5/12/15
Análisis procesamiento	3/12/15	9/12/15
Análisis almacenamiento	3/12/15	5/12/15
Generación documentación análisis	3/12/15	10/12/15
Desarrollo	11/12/15	27/12/15
Preparación entorno desarrollo	11/12/15	11/12/15
Adquisición datos	12/12/15	15/12/15
Procesamiento	16/12/15	20/12/15
Almacenamiento	21/12/15	21/12/15
Pruebas integración	22/12/15	24/12/15
Documentación Desarrollo	26/12/15	27/12/15
Generacion reglas negocio	28/12/15	30/12/15
Presentación	31/12/15	9/01/16
Análisis	31/12/15	2/01/16
Desarrollo cuadros mando	3/01/16	8/01/16
Documentación cuadros mando	9/01/16	9/01/16
Despliegue AWS	10/01/16	12/01/16
Fin	13/01/16	13/01/16

3 Arquitectura del DMP

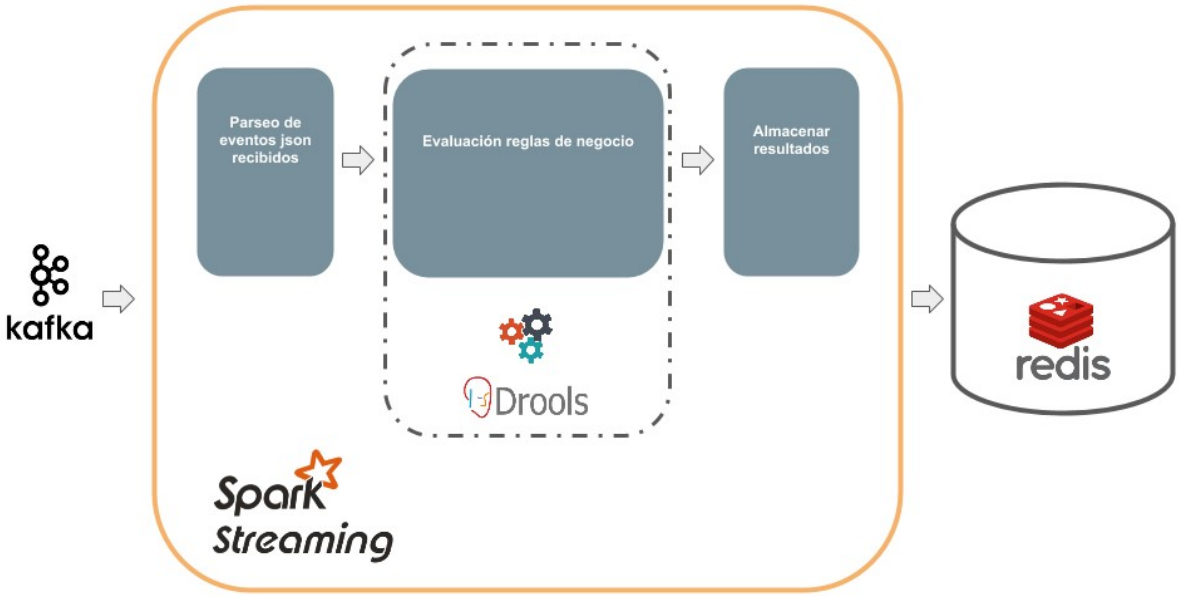
La tecnología CEP (Complex Event Processing) permite toma de decisiones de negocio en tiempo real a través de la captura, análisis y correlación de eventos, en base a unas determinadas reglas. Es una tecnología, por tanto, que permite detectar oportunidades o amenazas provenientes de diversas fuentes y dar, en tiempo real, una respuesta para actuar. Tiene, además, aplicaciones en diferentes industrias:

- Banca: bolsa, detección de fraude
- Aerolíneas: monitorización de operaciones
- Salud: reclamaciones, monitorización de pacientes
- Energía: interrupciones de servicio

Por su puesto, el procesado de eventos tiene también aplicación en el mundo del marketing, como es el caso este proyecto, en el que se desea categorizar a usuarios en tiempo real en base a las urls que han visitado.

La tecnología CEP, al igual que muchas otras, se ha visto afectada por la escalada que se ha producido en el volumen de datos durante los últimos años. Afortunadamente, existen varios proyectos en el ámbito del Big Data, que permiten presentar un arquitectura escalable como solución a este problema.

La estrategia a seguir es integrar un motor de reglas open source con un framework de procesamiento distribuido en streaming. En la siguiente imagen, se puede identificar las tecnologías elegidas para cada componente de la arquitectura:



A modo de resumen, los componentes utilizados en cada capa de la arquitectura son los siguientes:

- La ingestión de los datos se va a realizar a través de una cola *Kafka*.
- El procesamiento en streaming de los datos se va a realizar mediante *Spark Streaming*. En este caso el procesamiento no es tiempo real puro, sino que se realiza en micro batches, en los que se completa el análisis, clasificación y persistencia de los datos. La frecuencia de estos batches puede ser tan pequeña como se quiera, por ejemplo cada segundo, obteniendo una latencia total en el sistema de pocos segundos.
- *Drools* se va a utilizar como repositorio para las reglas de negocio; en este caso serán expresiones regulares y categorías asociadas a las mismas. Se trata de un motor de reglas open source que va permitir separar la lógica de la aplicación de la tecnología propiamente dicha, y que permite que tanto usuarios técnicos como usuarios de negocio sean capaces de diseñar o modificar fácilmente la lógica de negocio.
- Para almacenar las métricas y los datos de los usuarios se va a utilizar *Redis*.

A continuación se ofrece una pequeña explicación de cada una de las capas del sistema, la motivación de la tecnología elegida así como una descripción del flujo de los datos en cada parte de la arquitectura.

3.1 Adquisición de datos

El objetivo es esta capa es el de la recibir los documentos de navegación. Para ello se va a utilizar *Apache Kafka* como sistema de mensajería.

Kafka es un sistema distribuido de publicación-suscripción que ofrece particionado y replicado, lo que unido a su velocidad en lectura y escritura, lo convierten en una herramienta adecuada para tratar streams de datos generados a gran velocidad. Inicialmente fue desarrollado por *Linkedin*, aunque su utilización se encuentra actualmente muy extendida. Entre otras características adicionales, *Kafka* ofrece persistencia en los mensajes, y la comunicación entre publicadores y suscriptores se realiza mediante *TCP*, por lo que admite clientes en muchos lenguajes como *Java* o *Scala*.

Adicionalmente, *Kafka* se apoya en *Zookeeper* para almacenar estados y configuraciones en todo el cluster, como por ejemplo la distribución de las particiones de los tópicos o el offset de los consumidores.

Debido a todas estas características, la elección de *Apache Kafka* como sistema de mensajería parece apropiada, ya que nos va a proporcionar junto a *Zookeeper*, un sistema escalable y tolerante a fallos, capaz de procesar el stream de datos de navegación con el que se va a trabajar en el proyecto.

En cuanto a los datos con los que se va trabajar en el proyecto, se van a tratar documentos json como el siguiente:


```
{"UUIDC":"a517d781-af14-42ed-be95-
b1cc00fd9087","_F":"100001742096445","_GPID":"","_HTM":"","_TW":"","_gcmID":"APA91b
GpT0GP42KilXoPv4hBfnCJHd4YmQgbXslJ5vGZXGQwgHcScqKnLsZFuGBr2Eq0_SdPfKyZf3KHkT-
y9om13bFN3me3qOVxel-lnLLfUo17AhqPHQ8MTuhSrExB65HPzaX-p7KVpz_h-
el2PXXWnvUPoFbFoQ","_id":
{"$oid":"564e54a071b0479740223d28"},"_lp":"77","browser":"chrome","dominio":"es.wik
ipedia.org","dominio_md5":"f1f01dab7d045781add4efd381e7b4bc","dominio_origen":"goog
le.com.ec","extension_version":"80","fecha":"1447974009","fecha_insercion":"2015112
00000","fecha_local":"20151120","frame_id":"0","gaclient":"","gcmid_md5":"d5d940159
47b7492cd8569de825fdc4f","id_machine":"APA91bGpT0GP42KilXoPv4hBfnCJHd4YmQgbXslJ5vGZ
XGQwgHcScqKnLsZFuGBr2Eq0_SdPfKyZf3KHkT-y9om13bFN3me3qOVxel-
lnLLfUo17AhqPHQ8MTuhSrExB65HPzaX-p7KVpz_h-el2PXXWnvUPoFbFoQ","id_session":"ADW2|
S_FRIV_002","idextension":"dfhhpidgjbgbaddaneibkgafonllncan","idpageview":"9f380f11
2904194c05d1d25c1ba200a1","ip":"181.113.99.66","pais_iso2":"EC","parentframe_id":"-
1","partner":"ADW2","requestid":"23292","source":"ADW2|
S_FRIV_002","tabid":"23","timestamp_browser":"1447974005931000","top_id":"","url":"
https://es.wikipedia.org/wiki/Prometeo_encadenado","url_md5":"5aca2c9746d664ddee168
c9de9cc477d","url_referer":"https://www.google.com.ec/","urlorigen":"https://www.go
ogle.com.ec/","user_agent":"Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/46.0.2490.86 Safari/537.36","ventana":"main_frame"}
```


Entre los campos a destacar del documento, y con los que se va a trabajar en etapas siguientes de la arquitectura, se encuentra el id de usuario, la url a la que accede o la fecha en que se produce la navegación.

3.2 Procesamiento

El encargado de conectarse a un tópico *Kafka* y procesar en streaming los datos de navegación obtenidos será un proceso de *Spark Streaming*.

Esta tecnología no funciona en streaming real, sino que el procesado ocurre en micro-batches. En cada uno de estos micro-batches se van a incluir los siguientes pasos para cada uno de los documentos recibidos en ese stream de eventos:

1. Parseo del documento json y generación de objetos con información relevante.
2. Ejecución lógica de negocio con cada url recibida.
3. Almacenamiento en un api de las métricas necesarias.

Para ejecutar la lógica de negocio se va a utilizar un motor de reglas complejas como Drools. Concretamente se va a utilizar como un repositorio de expresiones regulares contra las que se va a ejecutar cada url para otorgarle todas las categorías que apliquen a un determinado patrón. A modo de ejemplo, una url del tipo http://tiendaonline.orange.es/moviles-google/nexus-6-32gb-blanco podrían asignarse varias categorías (orange, nexus, móviles).

El principal beneficio de utilizar un motor de reglas para la evaluación de las expresiones regulares es separar la lógica de negocio de la componente tecnológica propiamente dicha. De esta forma, es posible que tanto usuarios tecnológicos como usuarios de negocio modifiquen el repositorio de reglas. Con el fin de ilustrar este último punto, se va usar una hoja de cálculo en formato excel en la que se van a listar todas las posibles expresiones regulares así como la categoría a la que apliquen:

11	NAME	url	categoria
12	r1	^[\\w]*orange[.]com.*$	orange
13	r2	^[\\w]*orange[.]com[\\/].*shop.*$	orange (cliente potencial)
14	r3	^.*nexus.*$	nexus
15	r4	[\\da-z\\.-]+.com.*$	dominio.com

De esta manera, en el caso de que se deseen añadir, modificar o eliminar categorías bastaría con modificar esta hoja, sin necesidad de alterar el código de la aplicación.

El objetivo, por tanto de esta capa de procesamiento es obtener para cada documento recibido un objeto que contenga todas las categorías que apliquen a dicha navegación. Para almacenar correctamente esta información, y llevar a cabo las métricas adecuadas, se va a usar la lógica del motor de base de datos.

3.3 Almacenamiento

Como api para almacenar las métricas de la aplicación se va a utilizar *Redis*. Se trata de un motor de base de datos en memoria, basado en el almacenamiento de estructuras clave-valor, y que está diseñado especialmente para ofrecer velocidad. Actualmente soporta también persistencia en disco de la información, por lo que se puede utilizar como base de datos *NoSQL*.

La razón principal de elegir *Redis* como base de datos es la rapidez de escritura que ofrece, lo cual es necesario en el escenario de tratamiento de datos con el que se trabaja en el presente proyecto. Por otra parte, desde Abril de 2015, es posible distribuir la información almacenada mediante sharding automático entre diferentes nodos de un cluster, lo que va a permitir que la arquitectura escale en caso de ser necesario.

Entre los tipos de datos que soporta *Redis* están los *hashes*. Esta va a ser la estructura de información elegida para almacenar la información de navegación de un usuario. De esta forma, usando el identificador de usuario como clave, es posible guardar o actualizar todos los datos referentes a dicho usuario.

La información obtenida de las navegaciones, que se va a almacenar para cada usuario va a ser la siguiente:

- Id de usuario
- País de origen
- Fecha actualización del usuario
- Categoría de la url visitada
- Fecha última visita de a una categoría

Todos estos datos se van a guardar en un *hash* bajo la clave única del usuario. Además, con el fin de dar pesos a las distintas categorías visitadas, se utilizará la el comando *hincrby* de *Redis* para llevar un contador de cuantas veces determinado usuario se ha interesado en una categoría.

Por otra parte, se va a aprovechar la potencia y versatilidad de datos de *Redis* para poder computar al vuelo otro tipo de estadísticas, como los usuarios más activos o *heavy hitters*, o totalizaciones por geográficas o por categoría.

4 Descripción de la solución

A continuación, se ofrece una explicación más detallada de las 3 capas que se han diseñado para dar solución al problema, partiendo de la ingesta de datos en *Kafka*, procesado de los mismos, y presentación de los resultados en unos cuadros de mando diseñados a tal efecto.

Como fuente principal de los datos se han utilizado ficheros con navegaciones de usuarios almacenados según el día, en el formato json antes indicado, cada uno de los cuales tiene un tamaño aproximado de unos 7,5Gb.

4.1 Ingesta en *Kafka*

4.1.1 Publicación

Para la ingesta de datos en un tópico *Kafka* se ha implementado un cliente java, capaz de leer la información de los ficheros situados en un determinado directorio que se pasa como parámetro e insertarlos en un tópico, que también es especificado como parámetro.

Además, y con el fin de controlar la cantidad de mensajes que se ingestan en el tópico, este cliente se ha diseñado de tal forma que los mensajes son enviados en forma de batch, pudiéndose controlar, tanto el tamaño como la frecuencia del mismo.

El código de este cliente se puede encontrar en el siguiente repositorio:
https://github.com/davidonoro/KafkaPublisher

Este cliente se basa en dos únicas clases, con las siguientes características:

- *ProductorKafka*. Esta clase es el cliente en sí mismo. Gestiona la conexión con el broker y el tópico y presenta los métodos para enviar los mensajes y cerrar la conexión con el servidor.

- *PublicadorKafka*. Clase principal del programa. Se ocupa de leer los ficheros de entrada en el directorio especificado, y generar los batches de mensajes a enviar.

4.1.2 Configuración *Kafka*

Para lograr un mayor throughput a la hora de procesar los mensajes, *Kafka* utiliza la figura de las particiones. Las particiones de un tópico en *Kafka* se pueden entender como una unidad de paralelismo. Por un lado la escritura de los productores en diferentes particiones se puede ejecutar totalmente en paralelo; por otro lado, en la parte consumidora, *Kafka* siempre asigna los datos de una partición a un hilo. Por tanto, el grado de paralelismo está ligado al número de particiones.

El número de particiones de un tópico puede especificarse en el momento de su creación:

```
$KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions <numPartitions> --topic <topicName>
```

4.2 Procesamiento de la información

Todo el código de la aplicación, en cuanto al procesamiento, almacenaje y visualización de los datos puede encontrarse en el siguiente repositorio:

https://github.com/davidonoro/RTDMP

Se describen a continuación los componentes desarrollados según su utilización:

4.2.1 Consumo de mensajes desde *Kafka*

Para consumir los mensajes de un tópico *Kafka* mediante *Spark Streaming* se ha optado por una aproximación directa, sin receptores, mediante la utilización de un *directStream*. De esta forma, se pregunta de forma periódica a *Kafka* por los offset que deben ser consumidos en cada batch.
Esta es una forma sencilla de obtener paralelismo en la consumición de mensajes, ua que se crean tantos *RDDs* como particiones tenga definidas el tópico en el servidor *Kafka*. Esto está implementado de la siguiente forma:

```
val sparkConf = new SparkConf().setAppName("RTDMP")
val ssc = new StreamingContext(sparkConf, Seconds(5));
val kafkaParams = Map[String, String](       "metadata.broker.list" ->
       BROKER,"auto.offset.reset"->"largest")
val topics = Map[String,Integer](TOPIC->8)
val messages = KafkaUtils.createDirectStream[String, String, StringDecoder, StringDecoder]
(ssc,kafkaParams,Set(TOPIC))

messages.foreachRDD{
       ....
}
```

4.2.2 Parseo de los documentos recibidos

Se ha definido una objeto en scala, *NavigationDataExtractor*, capaz de parsear los documentos recibidos y generar un *POJO* (*Navigation*) con el que va poder trabajar el motor de reglas. Este objeto, solamente contiene los datos de interés de cara a la ejecución de la lógica de negocio y a la inserción en la base de datos, como el usuario, el dominio y la url visitadas, la fecha, etc.

A la hora de extraer los datos recibidos en el json, se ha utilizado una *case class* que define la estructura del json recibido. En el caso de que exista algún error en el documento, o que

no se ajuste a la estructura de datos esperada, se devuelve un *POJO* vacío, que es desestimado en las siguientes fases del procesamiento.

Por último, con el fin de simplificar el código en el driver de *Spark Streaming*, se ha definido esta implementación de tal manera que es capaz de trabajar con un conjunto de mensajes, que son los que se recibirán en cada ejecución del batch.

```
def parseMultipleData(jsonList: Iterator[(String,String)]): Iterator[Navigation]={
       val navigations = jsonList.map(kfkMsg=>{
              val parsedJson = parse(kfkMsg._2)
              try{
                     val doc = parsedJson.extract[event]
                     val data = new
              Navigation(doc.UUIDC,doc.pais_iso2,timeToStr(doc.fecha),doc.dominio,doc.url)
                     data
              } catch{
                     case mpe:MappingException => {
                     val data = new Navigation()
                     data
                     }
              }
       })
       navigations
}
```

4.2.3 Ejecución de reglas de negocio

Como ya se ha explicado, se ha utilizado *Drools* como motor de reglas para asignar categorías a cada navegación realizada. Además, se ha utilizado una hoja excel como repositorio de todas las reglas de negocio que se han ejecutado para cada documento recibido. Para implementar esta funcionalidad se han tenido en cuenta los siguientes aspectos:

Mediante un patrón *factory* se consigue crear una sola sesión contra el motor *Drools* por cada partición, de tal manera que se reutilizan estas sesiones en todas las nuevas ejecuciones del stream.

```
KieContainer kContainer =
       kieServices.newKieContainer(kieRepository.getDefaultReleaseId());
return kContainer.newStatelessKieSession();
```

Se utiliza una sesión sin estado ya que en este caso no es deseable que cambios en los hechos sean tenidos en cuenta por el motor de reglas, como por ejemplo un cambio en el objeto que se está tratando. Esto es una ventaja de diseño que nos permite modificar el objeto una vez evaluado.

Por tanto, en por cada ejecución del batch, y en cada partición, se ejecutan las reglas de negocio contra el objeto Navigation:

```
def evaluarMultipleReglas(data:Iterator[Navigation]):Iterator[Navigation]={
       val ksession = KieSessionFactory.getKieSession(rulesFile)
```

```
        val reglasEvaluadas = data.map(navigation=>{

        if(navigation.getDominio!=null){
        ksession.execute(navigation)
        navigation
        }else{
        new Navigation()
        }
        })
        reglasEvaluadas
}
```

La evaluación de los documentos de navegación se ha realizado en las reglas contra el dominio visitado y no contra la url. La razón de ejecutar de esta forma de reglas ha sido el desconocimiento del negocio y la rapidez de diseño del set. De cualquier forma, sería posible cambiar las reglas sin necesidad de modificar el código de la aplicación solamente mediante la utilización de otra hoja con el repositorio más complejo.
Para la obtención de los dominios se ha desarrollado un pequeño proceso de spark, en el que se hace un sencillo conteo de los dominios más visitados para su clasificación. Este proceso se puede encontrar en el paquete de test de la aplicación.

En la siguiente captura del repositorio Excel de reglas se puede comprobar el simple funcionamiento de las reglas definidas. Si el dominio del objeto Navigation cumple una expresión regular, se le añade la categoría correspondiente a dicho dominio:

RuleTable MisReglas		
NAME	CONDITION	ACTION
	nav:Navigation()	
	dominio matches "$1"	nav.addCategoria("$param");
NAME	dominio	categoria
r1	^.*facebook.com.*$	Redes_Sociales
r2	^.*login.live.com.*$	Email
r3	^.*xvideos.com.*$	Pornografia
r4	^.*mail.google.com.*$	Email

Es es en este punto en el que se ha obtenido beneficio de utilizar una sesión sin estado, puesto que la propia regla modifica el propio objeto de entrada para cada regla que se cumple. De esta forma podemos reutilizar los objetos y pasarlos a la siguiente fase del procesamiento, que es la inserción de los datos en la base de datos en memoria.

4.2.4 Inserción en *Redis*

Para la inserción de los datos en *Redis* se ha utilizado un cliente *Java*, *Jedis*, que da soporte a todas las operaciones necesarias en la aplicación. En caso de ser necesario escalar, y se plantease utilizar un cluster de *Redis*, este proyecto también daría soporte.

En primer lugar, para gestionar el acceso concurrente al servidor desde las diferentes particiones se ha utilizado un objeto *JedisPool*, de tal manera que es el propio cliente el que gestiona las conexiones de forma segura.

```
static JedisPool pool = new JedisPool(new JedisPoolConfig(),HOST);

public static Jedis getJedisCluster(){
    return pool.getResource();
}
```

En cuanto a los datos almacenados en por cada navegación, se han utilizado tipos de datos complejos para poder almacenar la información de forma eficiente: *hashes* y *sorted sets*.

En primer lugar, y como funcionalidad principal de la aplicación, se utiliza un hash como almacén de datos con la información de cada usuario. Usando como clave su id único, se almacena su información en diferentes campos, usando los comandos hset o hincryby según sea el caso. Es importante señalar que la complejidad temporal de ejecución en estas operaciones contra el servidor es O(1), lo que favorece la rapidez de las inserciones.

```
// Estadisticas de usuario
cluster.hset(data.getUser(),"USER",data.getUser());
cluster.hset(data.getUser(),"COUNTRY",data.getPais());
cluster.hset(data.getUser(),"LAST_UPDATE",data.getFecha());
long totalHits = cluster.hincrBy(data.getUser(),"TOTALHITS",1);
```

A parte, de estos datos básicos de los usuarios, se aprovecha y se llevan a cabo diferentes totalizaciones, como por ejemplo el total de navegaciones, agregado por categoría, país y por ambas. Para el caso de las totalizaciones por país se ha utilizado otro hash, en el que los campos corresponden al código de país recibido en las navegaciones. Estas totalizaciones se utilizarán más adelante como base de uno de los cuadros de mando.

```
// Estadisticas totales
cluster.incrBy("TOTALHITS",1);
if(data.getPais() != ""){
       cluster.hincrBy("TOTALHITS_COUNTRY",data.getPais(),1);
}
```

Por último, se lleva a cabo también una estadística de los usuarios más activos (heavy hitters). Esta totalización se lleva a cabo tanto a nivel global, es decir, los usuarios que más urls han visitado en general, como por categoría visitada. Para llevar a cabo esta contabilidad de usuarios, se ha utilizado uno de los tipos complejos que ofrece *Redis*, los *sorted sets*. Este tipo de datos permiten añadir miembros con un determinado score. En este caso, se añade el usuario con su totalización de hits. Con el fin de que la operación no

se demore excesivamente, pues su complejidad es O(log(N)) siendo N el tamaño del set, se reduce el tamaño a 100 usuarios tras cada nueva inserción o actualización.

```
private static final int TOPKHITTERS = -101;
cluster.zadd("HEAVYHITTERS",totalHits,data.getUser());
cluster.zremrangeByRank("HEAVYHITTERS",0,TOPKHITTERS);
```

4.2.5 Orquestación en Spark Streaming

La orquestación en un job de Spark Streaming de todos estos componentes se realiza de la siguiente forma consiste en la unión de todos los pasos anteriormente explicados. Todos los pasos (extracción información del documento json, ejecución de reglas de negocio, filtrado de documentos válidos e inserción en la base de datos) se realizan de forma periódica en cada ejecución del stream:

```
messages.foreachRDD(rdd => {

       println("Recibidos "+rdd.count()+" mensajes")

       val navList = rdd.mapPartitions(listMsg=>{

       val navs = NavigationDataExtractor.parseMultipleData(listMsg)
       val evalNavs = rulesExecutor.evaluarMultipleReglas(navs)
       val navFilteredList = evalNavs.filter(nav=> nav.getUser != null &&
                                    !nav.getCategorias.isEmpty  && nav.getPais != null)
       navFilteredList
       })

       val jedisManager = new JedisManager
       navList.foreach(nav=> {
       jedisManager.insertData(nav)
       })

       println("Batch insertando "+navList.count()+" registros en Redis")

})
```

4.3 Visualización de los datos

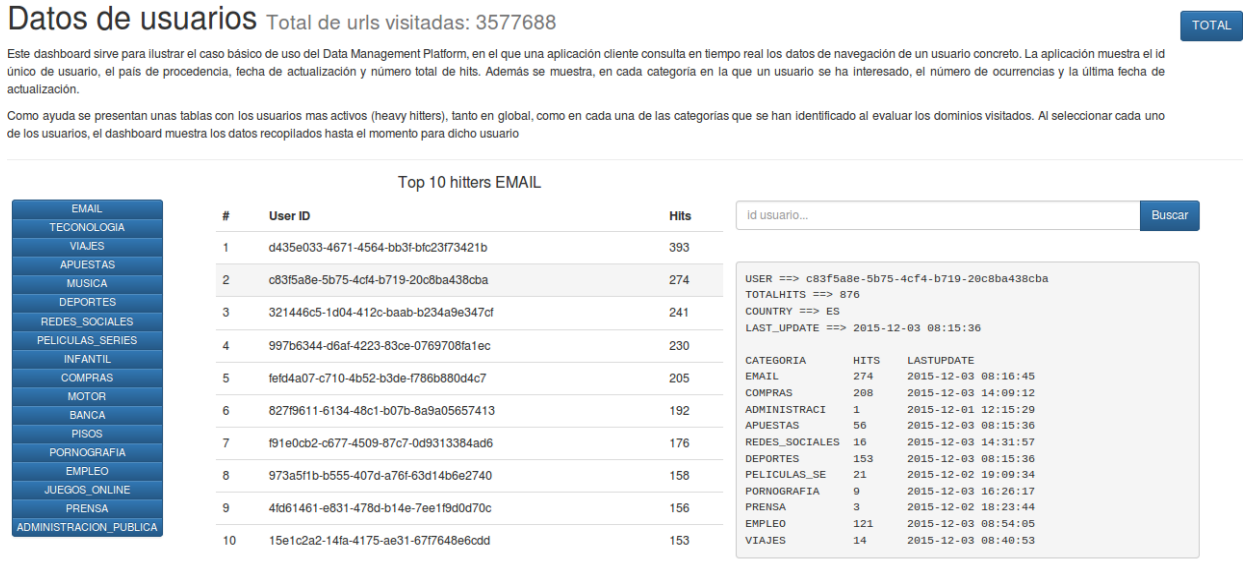
Para la visualización de los datos se han definido 2 cuadros de mando diferentes, que intentan mostrar dos aspectos diferentes de los calculados durante el procesamiento. Por un lado, se ha diseñado un cuadro de mando básico, que responde al caso de uso básico del DMP: ¿qué categorías ha visitado un usuario?. Por otro lado, se ha diseñado un cuadro de mando que da una visión geográfica de los datos, mediante comparación de datos procedentes de diferentes países.

Para el desarrollo de los cuadros de mando se ha utilizado HTML y Javascript, con apoyo de diferentes librerías: *d3*, *jquery*, *datamaps*, *boostrap*... Por otro lado, para hacer accesibles los datos de Redis a la capa de presentación, se ha utilizado el proyecto *Webdis*, que levanta un servidor http y proporciona un interfaz REST a la base de datos.

4.3.1 Datos de usuarios

Este cuadro de mando intenta resolver el principal caso de uso del DMP, que es saber qué categorías ha visitado un usuario para poder decidir qué anuncio mostrarle. Como el número de usuarios únicos de los que se tienen registro en los datos es muy alto, y para no tener que conocer los identificadores, se ha diseñado el cuadro de mando apoyándose en el dato conocido de los usuarios más activos.

Por tanto, el dashboard muestra inicialmente la lista de los 10 usuarios más activos a nivel general. Haciendo click en el id de cada uno, se muestran los datos de navegación conocidos en cada caso: id de usuario, país de origen, fecha de última actualización. Además, se muestran otros datos que se consideran de interés para el DMP: para cada categoría que se ha visitado se muestra también el número de veces y la fecha de la última ocurrencia.

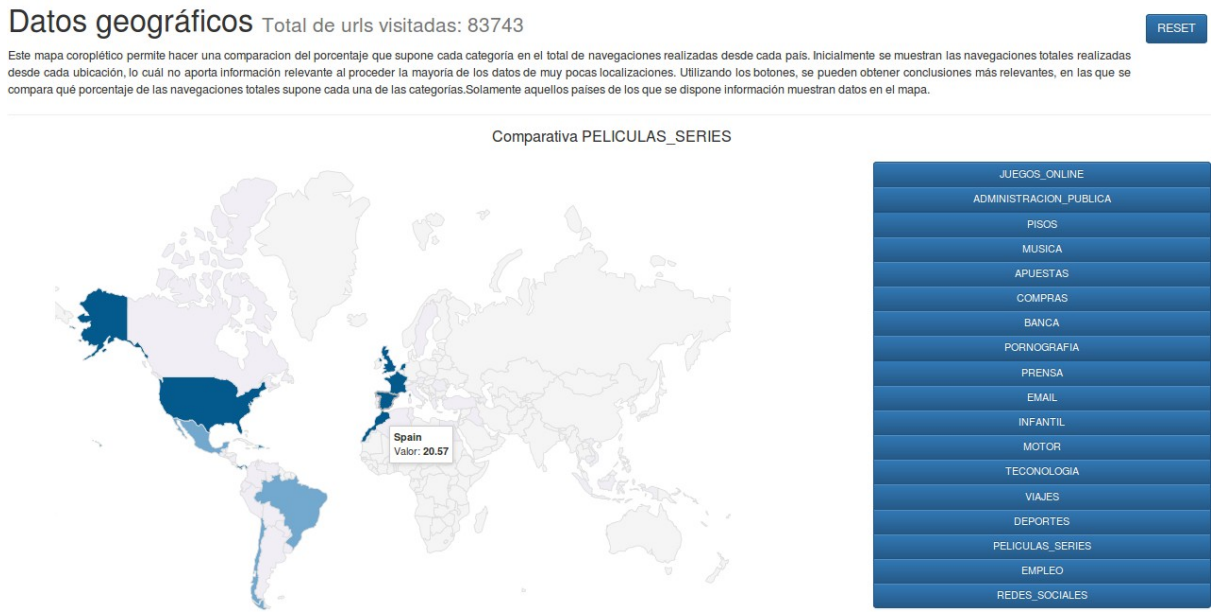


Para enriquecer el cuadro de mando, y poder consultar por un mayor número de usuarios, se permite además filtrar los usuarios más activos en cada una de las categorías identificadas en las reglas de negocio. Además, se ha incluido un campo de área de búsqueda para consultar los datos de cualquier usuario conocido.

4.3.2 Visión geográfica

Este cuadro de mando intenta ofrecer una visión geográfica sobre los datos mediante un mapa coroplético del mundo. Inicialmente se muestra el número total de navegaciones recibidas desde cada país, lo que no es dato especialmente significativo ya que los datos se encuentran muy sesgados al ser la mayoría de procedentes de los mismos países.

Para poder ofrecer información que realmente pueda ser comparada mediante el mapa, se utilizan los datos de los totales de navegación según la categoría y el país, lo que permite calcular, para cada país, qué porcentaje sobre el total de navegaciones representa determinada categoría.
Este porcentaje sí es un dato comparable mediante el mapa, y su visualización puede dar una idea más significativa sobre la navegación de los usuarios según su procedencia. Mediante los botones del dashboard se puede cambiar de categoría o volver a la representación inicial del mapa.



5 Configuración y ejecución

Las versiones del software utilizado para el desarrollo del proyecto han sido las siguientes:

- *Apache Kafka 0.8.2.2*
- *Spark 1.5.2*
- *Drools 6.3.0*
- *Redis 3.0.5*

Hay que tener en cuenta que se ha utilizado la versión 2.11 de Scala, lo cuál afecta a las distribuciones de *Kafka* y *Spark* que se han utilizado, puesto que ambos deben dar soporte a dicha versión del lenguaje. Especialmente delicado es en este caso la distribución de *Spark*, pues es necesario bajar los fuentes y compilarlos específicamente para generar los binarios necesarios para la ejecución. Una vez se baja el código fuente, es posible compilarlo mediante los siguientes comandos:

```
>$ export MAVEN_OPTS="-Xmx2g -XX:MaxPermSize=512M -XX:ReservedCodeCacheSize=512m"
>$ cd $SPARK_HOME
SPARK_HOME$> ./dev/change-scala-version.sh 2.11
SPARK_HOME$> ./make-distribution.sh --tgz -Pyarn -Phadoop-2.6 -Dhadoop.version=2.7.1 -Dscala-
2.11 -Phive -Phive-thriftserver -DskipTests
```

Para levantar los diferentes componentes del entorno, se deben seguir en orden los siguientes comandos:

Para arrancar el servicio de *Zookeeper*, que es dependencia de *Kafka* para poder manejar los tópicos, offset de los mensajes, etc:

```
$KAFKA_HOME/bin/zookeeper-server-start.sh $KAFKA_HOME/config/zookeeper.properties
```

Arranque del servidor *Kafka*:

```
$KAFKA_HOME/bin/kafka-server-start.sh $KAFKA_HOME/config/server.properties
```

En el caso de ser necesaria la creación de un tópico para recibir los mensajes, se puede realizar mediante el siguiente comando:

```
$KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1
--partitions 8 --topic TOPICO.RTDMP
```

Arranque del servidor *Redis*:

```
$REDIS_HOME/src/redis-server
```

Arranque del servidor webdis, para poder hacer peticiones rest al motor de base de datos:

```
$WEBDIS_HOME/webdis
```

Una vez están todos los componentes levantados, se puede iniciar el job de *Spark* streaming. Para ello será necesario empaquetar el proyecto mediante Maven, y lanzarlo al cluster. En este caso se ha lanzado a un cluster local, en el que se simulan 4 workers. Por otro lado, la aplicación necesita 3 parámetros de configuración que deben ser especificados como argumentos: el servidor de *Kafka* contra el que nos conectamos, el nombre del tópico del que se desean consumir los mensajes y por último el fichero con las reglas de negocio contra las que se van estudiar los documentos recibidos.

```
$SPARK_HOME/bin/spark-submit --master local[4]
                             --files $PATHTOFILE/ReglasTest.xlsx
                             --class com.pebd.rtdmp.RTDMP
                             ./target/RTDMP-0.0.1-SNAPSHOT-jar-with-dependencies.jar
                             localhost:9092
                             TOPICO.RTDMP
                             $PATHTOFILE/ReglasTest.xlsx
```

6 Conclusiones

Se presentaba en este proyecto un problema de clasificación de documentos en tiempo real en base a unas determinadas reglas de negocio, en este caso un set de expresiones regulares. El propósito era el de poder dar una respuesta en tiempo real de los intereses de usuarios para poder ofrecerles anuncios con mayor impacto.

Aislando el conocimiento de negocio, se ha apostado por una arquitectura de CEP para dar solución al problema. Esta solución se ha basado en en un motor de procesamiento en streaming (*Spark*) y un motor de reglas (*Drools*), todo ello apoyado en un potente servidor de mensajes como *Kafka*, y una base de datos en memoria (*Redis*), que permitiese la escritura de una gran cantidad de datos con gran velocidad.

Por otra parte, la utilización de un motor de reglas ha permitido separar completamente la parte tecnológica de la aplicación de la lógica de negocio, pudiendo ejecutar un set de reglas completamente diferente simplemente cambio una hoja de Excel.

6.1 Posibles mejoras

Existen varias mejoras que podrían introducirse en el proyecto de cara a mejorar su funcionalidad, especialmente en cuanto a la lógica del motor de reglas que se ha implementado.

Una posibilidad simple, que supone una evidente mejora del motor de reglas, es la de realizar un despliegue de un servidor de reglas remoto, de tal manera que las modificaciones y mantenimiento del mismo, no supongan un reinicio del procesado de las navegaciones. *Drools* provee un componente con esta funcionalidad, el *KIE Execution Server*, que puede ser desplegado en cualquier contenedor web y que ....
En principio, el impacto en el proyecto sería mínimo, ya que solamente habría que tocar el componente *KieSessionFactory*, de tal manera que ataque al servidor remoto en lugar de a la hoja excel utilizada en esta versión. De esta forma, la lógica de la categorización de las navegaciones quedaría completamente separada de tratamiento en streaming de los datos en la aplicación.

Otra posibilidad de mejora del proyecto sería la de intentar categorizar de forma automática aquellos dominios que no están contemplados en el repositorio de reglas actual. Por ejemplo se podría desarrollar la siguiente solución:

1. En el procesamiento en streaming, se insertarían estas url en un nuevo tópico *Kafka* para su posterior procesamiento en un proceso batch diario.
2. Este proceso estaría encargado de categorizar dichas url (una posibilidad sería la de leer el contenido de la web con un scrapper y realizar un wordcount del resultado obtenido), para posteriormente actualizar el repositorio de reglas remoto.

De esta forma, se podría ir mejorando de forma automática el repositorio de reglas sin necesidad de parar el procesamiento en streaming de los datos de navegación.

7 Bibliografía

http://www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/

https://databricks.com/blog/2015/03/30/improvements-to-kafka-integration-of-spark-streaming.html

http://blog.cloudera.com/blog/2015/11/how-to-build-a-complex-event-processing-app-on-apache-spark-and-drools/

https://github.com/xetorthio/jedis

http://webd.is/