Big Data.
Práctica de Spark Streaming

Iván Penedo[1]

Universidad Internacional Menéndez Pelayo, Madrid, España
https://uimp.es/

1 Introducción

Se ha diseñado e implementado un sistema capaz de detectar facturas anómalas utilizando técnicas de *clustering*, concretamente los algoritmos kMeans y Bisecting$k$Means sobre un flujo de datos en tiempo real. El objetivo de dicho sistema es identificar posibles fraudes, detectar oportunidades comerciales con clientes especiales, además de comparar la efectividad de ambos métodos mediante A/B testing y gestionar errores de datos de identificar facturas canceladas en tiempo real.

2 Entrenamiento

En un primer lugar, se llevó a cabo en el entrenamiento de los modelos de clustering que se iban a utilizar.

2.1 Fichero de entrenamiento

En el esqueleto del proyecto dado, se incluía un fichero `train.scala` que estaba diseñado para entrenar tan solo el modelo de kMeans. Para poder llevar a cabo el enrenamiento del otro modelo, se ha duplicado dicho fichero y se ha creado uno nuevo para el algoritmo de Bisecting$k$Means, de manera que podrían ser entrenados por separado, utilizando uno de los dos ficheros.

En este nuevo fichero, se han modificado específicamente las funciones `trainModel` y `distToCentroid` para utilizar el modelo `BisectingKMeansModel` correspondiente.

2.2 Modificaciones en la clase `Clustering`

En un primer lugar, se ha implementado la función `featurizeData`, en la cual se ha *parseado* primero la fecha a *timestamp* en el *dataframe* `dfWithTimestamp` y posteriormente se ha recuperado las horas con los minutos en decimales en el *dataframe* `dfWithHour`. Este valor se ha obtenido recuperando la hora y los minutos de la compra, obteniendo la parte decimal de la hora respecto a los minutos y sumando ambos valores tal que $hora\_decimal = hora + minutos/60.0$.

Tras esto, se ha agrupado por ID de facturas para calcular las métricas de estas ‘dfWithHour.groupBy("InvoiceNo")‘ y se han recuperado los datos con las métricas solicitadas, entre las que se encuentran:

- `AvgUnitPrice`
- `MinUnitPrice`
- `MaxUnitPrice`
- `Time`
- `NumberItems`

Además de estas, se ha añadido también el valor `HasCustomer` que indica que si el conjunto de productos que se han comprado en una misma factura tiene un identificados de cliente válido. Finalmente, se convierten las métricas a Double para utilizarlas para entrenar los modelos con la función `cast(DoubleType)`.

Tras esto, se ha implementado la función `filterData` que se encarga del filtrado el *dataset* eliminando aquellas filas que representen facturas canceladas (identificadas con “C” al inicio del `InvoiceId`), que contengan valores nulos debido a un error en la fase de `featurizeData`, o que contengan valores no validos por estar fuera del rango que les corresponde.

Para conocer el número de clusters adecuado para esta tarea, se ha utilizado el método *elbow*, por lo que se ha implementado la función `elbowSelection`. En este, se ha iterado sobre los posibles k dada la lista de costes y se ha recuperado el primero de ellos que supere el ratio dado como parámetro, siguiendo así la especificación del enunciado de la práctica, de forma que

$$\frac{error(k)}{error(k-1)} > 0.7$$

siendo 0.7 el ratio establecido.

2.3 Script de ejecución del entrenamiento

Se ha creado el fichero `start_training.sh` con en el que se podrían realizar ambos entrenamientos, para el kMeans y el Bisecting$k$Means. En el se ha utilizado dos instrucciones `spark-submit` con las correspondientes clases `KMeansClusterInvoices` y `BisectingKMeansClusterInvoices`, el fichero jar `anomalyDetection-assembly-1.0.jar` previamente compilado y la ruta del fichero de entrenamiento `training.csv`. Además, se introducen las rutas del directorio `clustering` y el fichero `threshold` correspondientes.

De esta manera, se realizaría primero el entrenamiento de kMeans con el fichero `train.scala` y, posteriormente, el entrenamiento de Bisecting$k$Means utilizando el fichero `trainBisecting.scala`. Tras la ejecución de este script, se obtienen los directorios **clustering/** y **clustering_bisecting**, junto a los ficheros **threshold** y **threashold_bisecting/**, para poder continuar con el desarrollo.

3 *Pipeline*

En esta sección se comenta el diseño del *pipeline* de Apache Spark Streaming implementado, el cual se encuentra representado en el diagrama de flujo de la figura 1.

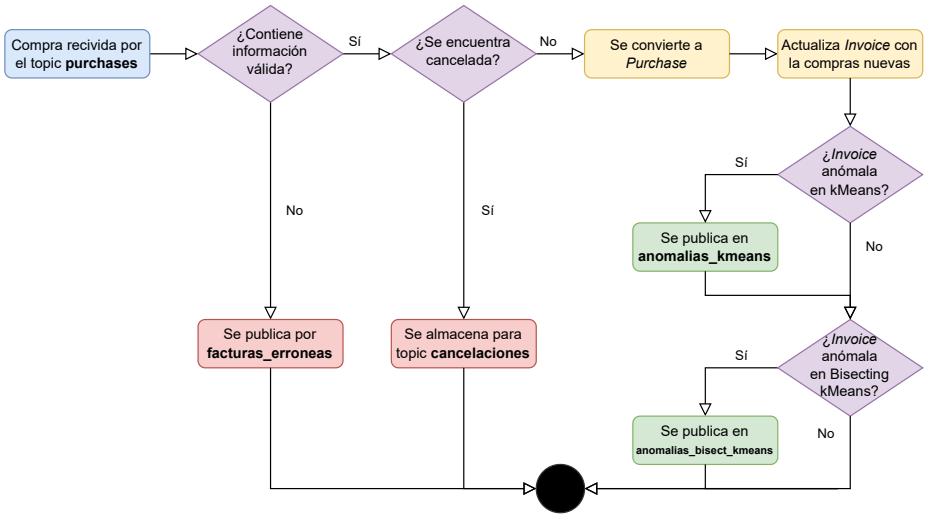

Fig. 1. Diagrama de flujo del *pipeline* desarrollado.

Este diagrama resume el proceso que sigue el flujo de datos que se recibe por el tema **purchases**, detectando las facturas erróneas y las cancelaciones para no trabajar con ellas, y posteriormente recogiendo las facturas anómalas en ambos modelos kMean y Bisecting$k$Means previamente entrenados. Como se puede observar, las compras que no son válidas o que se encuentran canceladas, no se utilizan para actualizar las facturas y, por lo tanto, no se predicen con los modelos. Además, todas las facturas, tras actualizarse, son predichas tanto con el modelo kMeans como con el modelo de Bisecting$k$Means.

En primer lugar, para poder cargar ambos modelos, se ha duplicado la función `loadKMeansAndThreshold` a la correspondiente `loadBisectingKMeansAndThreshold` para que, utilizando ambas funciones, se pueda cargar los ficheros obtenidos del entrenamiento realizado. Por otra parte, se inicializó el *broadcast* del `SparkContext`.

3.1 Facturas erróneas

Para tema de `facturas_erroneas` se han filtrado aquellas que contienen algún valor vacío o con más o menos valores de los ocho esperados, puesto que han sido los únicos casos que se han considerado que puedan describirse facturas erróneas. Posteriormente, estas son almacenadas en un *stream* `invalidInvoices` y se emiten por dicho tema.

3.2 Facturas canceladas

Para el tópico de `cancelaciones` se han filtrado aquellas cuyo `invoiceNo` comience con el carácter "C". Como en vez de los identificadores de las transacciones que hayan sido canceladas, se solicita tan sólo el numero de estas en los últimos 8 minutos, se han contado las transacciones y se ha utilizado la función `reduceByKeyAndWindow`. En esta, se ha definido la función de agregación (en este caso, la suma de los valores) junto a la de reducción (la resta de los valores), al igual que un tiempo de ventana de ocho minutos para recoger los datos dentro de esta franja y también la frecuencia de publicaciones a un minuto.

De esta manera, emitiendo cada RDD por el tópico `cancelaciones`, se obtiene un mensaje cada minuto de ejecución en el que se encuentre la cantidad de transacciones recientes canceladas en dicha ventana.

3.3 Transformación de facturas

Para poder convertir las compras a facturas, primero se ha buscado convertir los pares $(clave, valores[\,])$ a *Purchases*, un objeto ya definido. Para ello se ha creado la función `parsePurchases`, que recupera la información que se requiere para definir dicho objeto y finalmente devuelve un par con la *Purchase* creada junto a su identificador.

Tras esto, se define la función `updateInvoiceState` que se encarga de ir agrupando aquellas compras con mismo `invoiceNo`, recuperando las transacciones previas con su `state` y actaulizando sus valores respecto a las nuevas transacciones `newPurchases`.

En el caso de que no haya un estado previo, se crea uno de base para posteriormente actualizarlo. Tras esto, se obtienen los datos antiguos de la factura con el fin de recalcularlos utilizando los valores de cada una de las nuevas transacciones, manteniendo la coherencia de los cálculos de promedio, mínimo y máximo de los precios, así como el número de elementos. El promedio se actualiza tal que

$$avg = \frac{avg * n + p * q}{n + q}$$

donde avg es el promedio, n el número de productos en la factura, p el precio del producto y q el número de productos que se han comprado. Finalmente, se genera una nueva 'Invoice' con los datos actualizados.

Utilizando estas dos funciones, podemos aplicar `map` al flujo de datos obtenido tras eliminar las compras no válidas o canceladas del original `purchasesFeed`, además de la función `updateStateByKey` utilizando la función `updateInvoiceState` previamente mencionada. De esta manera, se obtendría el flujo de datos con las facturas recientemente actualizadas.

3.4 Detección de anomalías

Para saber si una factura es una anomalía en alguno de los modelos kMeans o Bisecting$k$Means ya entrenados, se debe obtener que la distancia de esta al centroide más cercano es mayor que el umbral *threshold* dado por el entrenamiento

del modelo. Para ello, se han creado dos nuevas funciones con el mismo nombre donde

```
isAnomaly(
    model: KMeansModel,
    invoice: Invoice,
    threshold: Double
): Boolean
```

para el primer modelo, y

```
isAnomaly(
    model: BisectingKMeansModel,
    invoice: Invoice,
    threshold: Double
): Boolean
```

para el segundo.

En estas, se han reutilizado las funciones `distToCentroid` implementadas en los ficheros `train.scala` y `trainBisecting.scala` para obtener la distancia de dichas `Invoices` a los centroides.

Tras esto, se pueden filtrar aquellas anomalías en cada uno de los modelos y finalmente publicar en los tópicos `anomalias_kmeans` y `anomalias_bisect_kmeans` aquellas facturas que fueron detectadas como anómalas.

4 Guía de ejecución

Para poder ejecutar la aplicación, deben estar los servidores de `zookeeper` y `kafka` en ejecución. Estos pueden ser levantados ejecutando las siguientes líneas, cada una en una nueva terminal, desde el directorio `/opt/Kafka/kafka_2.11-2.3.0/`:

```
sudo bin/zookeeper-server-start.sh config/zookeeper.properties
sudo bin/kafka-server-start.sh config/server.properties
```

Además de esto, es necesario tener los temas o *topics* necesarios ya creados:

- `purchases`
- `facturas_erroneas`
- `cancelaciones`
- `anomalias_kmeans`
- `anomalias_bisect_kmeans`

Esto puede comprobarse con el siguiente comando:

```
bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

En caso de que no se aparezca alguno de los *topics*, pueden crearse reemplazando `TOPIC_NAME` por el *topic* deseado en el siguiente comando:

```
bin/kafka-topics.sh --create bin/kafka-topics.sh --topic TOPIC_NAME \
    --partitions 1 -replication-factor 1 --zookeeper localhost:2181
```

Posteriormente, se puede compilar el proyecto ejecutando el siguiente comando desde la raíz del proyecto:

```
sbt clean assembly
```

Aunque ya se den entrenados los modelos en el proyecto entregado, como se ha comentado anteriormente pueden volver a entrenarse ejecutando el script `start_training.sh` diseñado con este propósito.

Antes de ejecutar el proyecto, es necesario también tener ejecutado en una terminal el script `productiondata.sh`, el cual se encarga de enviar las transacciones al *topic* de `purchases` que la aplicación leerá para comenzar con el *pipeline*.

Finalmente, se puede ejecutar el script `start_pipeline.sh` para levantar el ejecutable de la aplicación desarrollada y que se encargará de publicar los datos solicitados en los cuatro *topics* mencionados.

Para poder comprobar su funcionamiento, se puede conectar una terminal a uno de los *topics* reemplazando `TOPIC_NAME` por el nombre del *topic* deseado en el siguiente comando:

```
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
    --topic TOPIC_NAME -from-beginning
```