

ESIC Business & Marketing School

MÁSTER EN BIG DATA MANAGEMENT

**ANÁLISIS DE TRANSACCIONES
MONETARIAS PARA LA DETECCIÓN DE
FRAUDE**

Trabajo Final de Máster

ID de Alumno: 149344

Enero 2021

Índice

1	Introducción	3
2	Arquitectura	3
3	Ciclo de vida	5
3.1	Ingesta/Preprocesado	5
3.2	Procesamiento	5
3.3	Almacenamiento	6
3.4	Explotación	6
4	Modelo de datos	7
4.1	Transactions	7
4.2	Identity	8
5	Desarrollo del proyecto	8
6	EDA: Análisis exploratorio sobre el conjunto de datos	9
6.1	Análisis de variables	9
6.1.1	Descripción de los dataframes	10
6.1.2	Transaction ID	10
6.1.3	Transaction AMT	10
6.1.4	Product CD	11
6.1.5	Variables card	11
6.1.6	P_emaildomain y R_emaildomain	12
6.1.7	Variables Device	12
6.1.8	isFraud	13
6.1.9	Correlación entre variables	13
6.2	Aproximación al problema	14
7	Ingesta	16
7.1	Apache Flume	16
7.2	Apache Kafka	18
8	Procesamiento de datos estáticos	19
9	Generar datos para el flujo dinámico	22
10	Procesamiento en tiempo real	24
11	Almacenamiento	26
12	Explotación	29
13	Monetización y coste del proyecto	29
14	Conclusiones	33

15	Lista de figuras	33
16	Bibliografía	34

Trabajo de Fin de Máster: Análisis de transacciones monetarias para la detección de fraude

ID de Alumno: 149344¹

1. Introducción

Las Fintech son un nuevo referente en el sector bancario, con la capacidad de proporcionar nuevos servicios financieros gracias a su foco en las nuevas tecnologías, lo que permite la captación de un nuevo público, abarata costes y proporciona nuevas oportunidades de negocio [1]. El caso de uso que se desarrollará en este Trabajo de Fin de Máster es el desarrollo de un entorno de pruebas capaz de detectar el fraude bancario, tanto en compras con tarjetas de crédito como en transferencias bancarias. En este hipotético escenario, pertenecemos a una empresa relacionada con el sector bancario el cual ha creado una nueva firma para su marca, una Fintech, con una aplicación móvil y una tarjeta electrónica que permite su uso por el público medio a lo largo de todo el territorio nacional y extranjero en diferentes comercios y puntos de venta. Debido a la diferencia a nivel de desarrollo tecnológico (sobre todo en la parte transaccional) que hay entre la rama principal de la empresa y esta nueva start-up, es de vital importancia crear un sistema propio que permita detectar y prevenir el fraude bancario. Es por ello, que se realizará un desarrollo completo aprovechando la infraestructura de Big Data disponible, con un dataset de prueba y que permitirá reportar una base para este desarrollo a nuestros superiores. A su vez, se realizará un análisis de cómo se podría implementar una arquitectura similar en un entorno de producción, haciendo hincapié en las configuraciones de las tecnologías y distribuciones que se van a utilizar. Por último, se hará una estimación del coste que supondría la implementación del proyecto, el coste de las licencias, software y personal necesario para poner en marcha este desarrollo, a falta de las pertinentes reuniones con proveedores de software para negociar estos acuerdos comerciales.

2. Arquitectura

La infraestructura y el flujo de datos de este desarrollo se muestra en la siguiente imagen 1.

¹Máster en Big Data Management 2019-2020

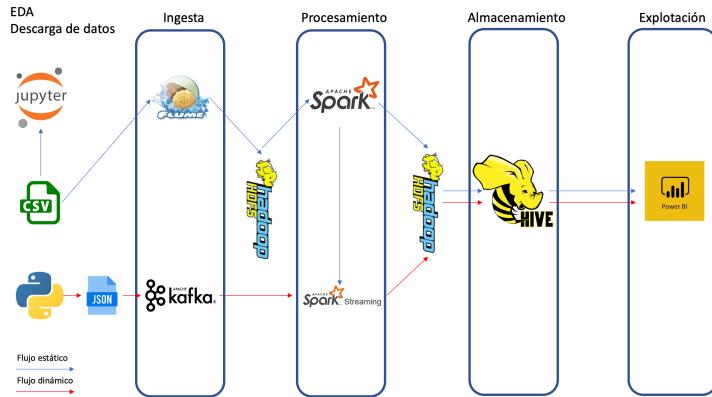


Figure 1: Arquitectura

La primera fase comprende varias partes. Se realiza un análisis exploratorio y un modelaje inicial sobre los datos que poseemos, con el objetivo de obtener más información que acerca de los datos de la que disponemos inicialmente (ver sección 4).

De forma añadida, durante esta fase (aunque no de forma paralela al proceso anterior), se ejecuta el script desarrollado en Python, el cual lee el archivo de train estático procesado por Apache Spark en capas posteriores y produce muestras aleatorias basadas en esta información, con el fin de replicar una arquitectura en tiempo real. Por último, este script realiza una conexión con nuestro servicio de Apache Kafka, abriendo un productor que ingesta los datos previamente generados en formato json en el topic designado.

La siguiente fase consiste en la ingestión de los datos estáticos y dinámicos. De forma inicial se produce la ingestión de los datos en bruto al sistema de almacenamiento disponible, que es HDFS, mediante Apache Flume. El motivo de la elección de Apache Flume sobre otras tecnologías disponibles, como la que utilizamos a la hora de ingestar los datos dinámicos, es el tipo de ficheros a insertar. Al ser una serie de ficheros estáticos en formato CSV, Apache Flume nos provee de forma nativa de la funcionalidad suficiente para transferirlos a nuestro sistema de almacenamiento principal, HDFS, sin tener que añadir conectividad extra, como en el caso de Apache Kafka, el cual se especializa principalmente en la ingestión de ficheros de tipo log.

En el mismo proceso de ingestión, se produce la lectura de los ficheros json de muestra para que los consuma Apache Spark Streaming en un futuro.

La siguiente fase consiste en el procesamiento de la información disponible. Para ello disponemos de Apache Spark, con el cual por una parte realizaremos un modelo de predicción de fraude con los datos disponibles, guardaremos el archivo de datos de entrenamiento procesado para que lo utilice el script de Python correspondiente a la primera capa y procederemos a guardar el modelo después de validararlo, además de almacenar los resultados de predicción del conjunto de testing en HDFS.

Apache Spark Streaming, en cambio, realiza una lectura de los ficheros json que se encuentran en el topic de Apache Kafka, abriendo una conexión de consumidor, y aplicando un esquema determinado para su lectura. Una vez se lee cada batch de datos, se abre el modelo guardado previamente en la otra fase de procesamiento y se realiza una predicción de los datos en Streaming. Finalmente, se almacenan estos resultados en HDFS.

La tercera fase es el almacenaje, para su posterior explotación. Para ello utilizaremos Apache Hive, y crearemos las tablas necesarias para almacenar tanto los datos que poseíamos en crudo como los ya procesados para futuras consultas.

Estas consultas se realizarán desde Microsoft PowerBI, herramienta gráfica con la que desarrollaremos un informe para la plantilla ejecutiva sobre los resultados de nuestro proyecto en el entorno de pruebas.

3. Ciclo de vida

60 El ciclo de vida que se ha elegido para este desarrollo es el modelo iterativo. El motivo de ello es que este desarrollo lo realizará una única persona, por lo que el uso de nuevas metodologías de gestión de proyectos como Agile no se contemplan, al no existir un reparto de tareas. Las ventajas de elegir este ciclo de vida de desarrollo frente al modelo en cascada convencional es la capacidad de reducir el riesgo a la hora de entregar el producto, para así poder ajustarse más a las necesidades del usuario. Debido a que no existe ninguna infraestructura similar en la start-up, todas las etapas de la arquitectura mencionada anteriormente son fundamentales para el desarrollo de este entorno de pruebas y su futura implementación. Es por ello que cada una de las iteraciones de este ciclo de vida se centrará en una etapa de la arquitectura.

3.1. Ingesta/Preprocesado

- Set up de la distribución de Apache Kafka a utilizar, con sus configuraciones pertinentes (Tiempo estimado: 20h)
- Búsqueda de cambios y posibles mejoras a implementar para un entorno de producción en la distribución de Apache Kafka (Tiempo estimado: 4h)
- Análisis exploratorio sobre los datasets de muestra en Python (Tiempo estimado: 2h)
- Preparación de datos de muestra para el flujo de datos en Streaming (Tiempo estimado: 1h)
- Preparación del modelo a realizar en fases posteriores, así como puesta a punto de hiperparámetros del mismo (Tiempo estimado: 5h)
- Realización de pruebas de ingesta (Tiempo estimado: 1h)
- Realización de pruebas con el Script de datos en Streaming (Tiempo estimado: 1h)
- Despliegue y pruebas de integración (Tiempo estimado: 1h)

3.2. Procesamiento

- Flujo estático
 - Modelaje en base a los resultados anteriores en Apache Spark (Tiempo estimado: 5h)
 - Pruebas (Tiempo estimado: 1h)
 - Almacenaje del modelo
- Flujo dinámico

- Predicción del fraude en los datos generados en base al modelo previo (Tiempo estimado: 1h)
- Reentrenar el modelo (Tiempo estimado: 1h)
- Pruebas (Tiempo estimado: 1h)
- Almacenaje del modelo

- **Despliegue y pruebas de integración (Tiempo estimado: 1h)**

3.3. Almacenamiento

- Creación del modelo de datos (Tiempo estimado: 1h)
- Carga de los datos en las tablas (Tiempo estimado: 1h)
- Creación de vistas necesarias en base a uso (Tiempo estimado: 1h)
- Despliegue y pruebas de integración (Tiempo estimado: 1h)

3.4. Explotación

- Conexión entre Apache Hive y Microsoft Power BI (Tiempo estimado: 5h)
- Creación del informe ejecutivo (Tiempo estimado: 10h)
- Despliegue y pruebas de integración (Tiempo estimado: 1h)

Por último, se añade una fase final en la cual se realizarán pruebas de integración de cada uno de los módulos y se automatizará el proceso completo, con un tiempo estimado de desarrollo de 20 horas más. En conjunto, este desarrollo consistiría en un total de 85 horas. La planificación de este proyecto es la siguiente.

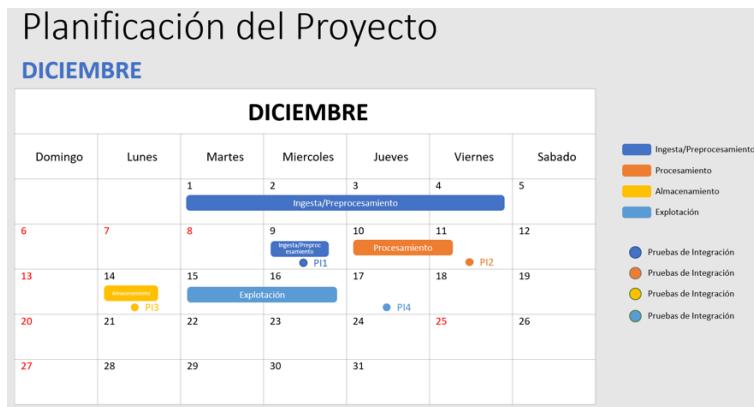


Figure 2: Planificación del proyecto

Si se dispusiese de más personal para el proyecto, otra metodología aceleraría el proceso. Siendo dos empleados los encargados del desarrollo, reduciría de forma considerable el tiempo de este (hasta aproximadamente la mitad), en base a la modularidad existente en el proyecto y su nula interdependencia (salvo por la automatización de procesos y la unión de este, así como de las pruebas de integración)

4. Modelo de datos

En este apartado haremos referencia al modelo de datos que se va a utilizar en nuestra base de datos (en este caso, Apache Hive). Cabe destacar que la mayoría de la información disponible no está catalogada. Esto implica que existe un gran volumen de información desconocida en los datasets que disponemos y que sólo podemos inferir. Uno de los objetivos a largo plazo de este desarrollo es categorizar de forma correcta las fuentes de datos de origen y todos los datos pertenecientes al flujo creado, en cuanto se disponga información de los mismos. Por el momento, el modelo de datos actual es el siguiente. Debido a la gran cantidad de variables, se ha decidido no mostrar de forma gráfica todos los atributos que tiene cada entidad.

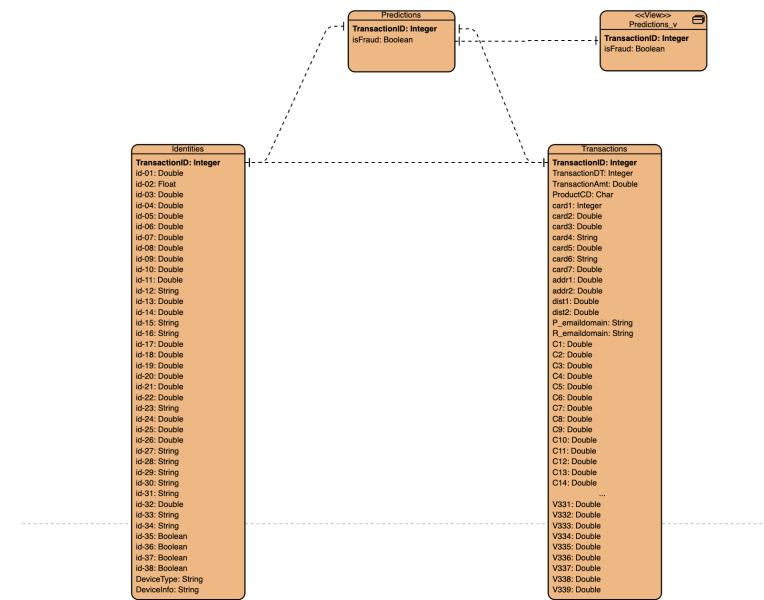


Figure 3: Modelo de datos

- 120 La tabla de transacciones (Transactions), contiene datos de transacciones derivadas de la compra de bienes, servicios y transacciones monetarias. Las variables que lo conforman son.

4.1. *Transactions*

- `TransactionID`: ID de la transacción
- `TransactionDT`: Variación de tiempo (no es un timestamp). Por temas de privacidad de la información sólo se proporcionan variaciones de tiempo respecto al momento en el que se empezaron a recoger los datos.
- `TransactionAMT`: Importe en dólares de la transacción
- `ProductCD`: Código del producto/servicio (el motivo) de la transacción
- `card1-card6`: Información de la tarjeta utilizada, como tipo de tarjeta, entidad bancaria, etc.

- addr: Dirección del emisor de la transacción
- addr1: Región fiscal
- addr2: País fiscal
- dist: Distancias entre sus direcciones
- P_emaildomain, R_emaildomain: Dominio del correo electrónico del emisor y receptor
- C1-C14: Valor anónimo que indica si existe información del emisor/receptor en la BBDD.
- D1-D15: Deltas entre transaccionesy otros pagos (no se especifica más información)
- M1-M9: Indican si hay unión entre nombres/tarjetas, etc (valor anonimizado)
- Vxx: Atributos generados por Vesta (no se especifican)

La tabla Identity hace referencia al usuario que ha realizado la transacción. Datos relevantes de esta tabla (además del ID de transacción con las que se vincula al resto) son elementos como el dispositivo en el que se realiza la transacción, la aplicación, entre otras:

4.2. *Identity*

- TransactionID: ID de la transacción
- TransactionDT: Variación de tiempo (no es un timestamp). Por temas de privacidad de la información sólo se proporcionan variaciones de tiempo respecto al momento en el que se empezaron a recoger los datos.
- id-01 - id-11: Información anonimizada sobre la información de la sesión del usuario (proxy, IP, red, etc)
- id-12 - ?: Información del dispositivo utilizado

La siguiente tabla, Predictions, solo dispone de dos atributos, el ID de la transacción y el valor que indica si es considerado fraude o no. Debido a su futuro frecuente acceso, se realizará una vista materializada de la misma para su consulta.

5. Desarrollo del proyecto

Durante las siguientes secciones se realizará una descripción del proceso completo siguiendo el orden de ejecución, con el fin de guiar al lector a la hora de replicar el proceso seguido.

156 **6. EDA: Análisis exploratorio sobre el conjunto de datos**

En esta sección se realizará un análisis exploratorio del conjunto de datos inicial que poseemos. Para ello, utilizaremos los cuadernos de Júpiter, que nos permiten de forma sencilla y muy visual realizar este primer análisis. El archivo correspondiente se encuentra en la carpeta que contiene todos los documentos del proyecto.

```
In [104]: #Unimos los conjuntos de train y test respectivamente
train = pd.merge(train_identity,train_transaction, on='TransactionID', how='left')
test = pd.merge(test_identity,test_transaction, on='TransactionID', how='left')

In [105]: train.head()
Out[105]:
   TransactionID  id_01  id_02  id_03  id_04  id_05  id_06  id_07  id_08  id_09 ...  V330  V331  V332  V333  V334  V335  V336  V337  V338  V339
0  2987004    0.0  70787.0  NaN  NaN  NaN  NaN  NaN  NaN  ...  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
1  2987008   -5.0  98945.0  NaN  NaN  0.0  -5.0  NaN  NaN  ...  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
2  2987010   -5.0  191631.0  0.0  0.0  0.0  0.0  NaN  NaN  ...  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
3  2987011   -5.0  221832.0  NaN  NaN  0.0  -6.0  NaN  NaN  ...  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
4  2987016    0.0  7460.0  0.0  0.0  1.0  0.0  NaN  NaN  0.0  ...  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

5 rows x 434 columns

In [106]: test.head()
Out[106]:
   TransactionID  id_01  id_02  id_03  id_04  id_05  id_06  id_07  id_08  id_09 ...  V330  V331  V332  V333  V334  V335  V336  V337  V338  V339
0  3663586   -45.0  280290.0  NaN  NaN  0.0  0.0  NaN  NaN  ...  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
1  3663588    0.0  3579.0  0.0  0.0  0.0  0.0  NaN  NaN  ...  0.0  0.0  310.0  90.0  0.0  310.0  90.0  0.0  0.0  0.0
2  3663597   -5.0  185210.0  NaN  NaN  1.0  0.0  NaN  NaN  ...  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
3  3663601   -45.0  252944.0  0.0  0.0  0.0  0.0  NaN  NaN  0.0  ...  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
4  3663602   -95.0  328680.0  NaN  NaN  7.0  -33.0  NaN  NaN  ...  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN  NaN
```

Figure 4: Train Dataset

El primer paso es leer los archivos que poseemos. Como se puede comprobar, inicialmente poseemos dos datasets, divididos en cuatro ficheros distintos. Estos datasets son los conjuntos de entrenamiento y testing, y cada uno de ellos posee dos archivos, ***Identity*** y ***Transaction***. Tal y como se ha comentado en la sección 4, La tabla ***Transaction*** contiene datos de transacciones derivadas de la compra de bienes, servicios y transacciones monetarias mientras que ***Identity*** hace referencia al usuario que ha realizado la transacción. Como poseen los campos de ID de transacción en común, podemos realizar un join a las dos tablas y obtener los datasets train y test, respectivamente.

168

Una vez realizado este paso, hacemos un primer vistazo a las tablas. Estas poseen más de 430 columnas, de las cuales no se posee información de todas ellas según el modelo de datos. Además, las columnas del dataset de testing poseen los nombres del conjunto de columnas distinto al de training, por lo que formateamos estos valores para que coincidan. Esto tendrá que tenerse en cuenta más adelante en la capa ??.

Para realizar un análisis exploratorio más preciso, vamos a utilizar SweetViz [2], que es una librería de código abierto de Python que genera este análisis exploratorio de forma visual y permite exportarlo a diferentes formatos para su posterior consumo.

Además, te permite realizar una comparación entre dos dataframes distintos, lo cual es conveniente para este caso en concreto.

6.1. Análisis de variables

180

Utilizando SweetViz, podemos obtener el reporte en html del análisis sobre los dataframes. Lo primero a tener en cuenta es que en el dataset de entrenamiento poseemos más de un 60 por ciento de columnas con valores nulos. Esto implica que a la hora de realizar el modelado, tendremos que

eliminar estas columnas del dataset de testing, para que se haga una predicción correcta de los valores de fraude.

A continuación se hace un análisis más detallado de los dataframes y de las variables que a primera vista, van a dar mayor valor y tenemos información de qué representan.

6.1.1. Descripción de los dataframes

Tal y como se puede ver en la imagen 5, los datasets contienen el mismo número de columnas, con la excepción de que el dataset de entrenamiento contiene la etiqueta que determina si una transacción es considerada como fraude o no.



Figure 5: Descripción de los dataframes

Además, también vemos que no posee valores repetidos, por lo cual no hay que realizar ese tratamiento posterior. Por otra parte, se categorizan las columnas de ambos dataframes en tres categorías distintas, variables numéricas, que corresponden a aproximadamente el 70 por ciento del total, variables categóricas, que corresponden a aproximadamente un 20 por ciento del total y las restantes, que son de tipo String.
192

6.1.2. Transaction ID

Esta variable representa el identificador único de cada transacción 6. No posee valores nulos ni elementos repetidos en ninguno de los dos datasets, tanto el de entrenamiento como el de testing.

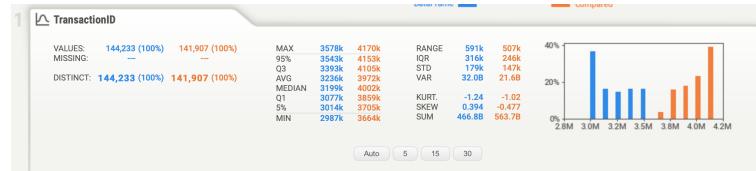


Figure 6: Transaction ID

6.1.3. Transaction AMT

Esta variable hace referencia al importe en dólares de la transacción realizada. Como podemos ver en la imagen adjunta 7, la mayoría de estos importes son menores a los 250 dólares. Al ser este un conjunto de datos real, la distribución de los mismos cobra sentido. Podemos ver que tampoco posee valores nulos, y que sólo un 5 por ciento de los mismos son valores únicos (debido a la distribución previamente mencionada).
204

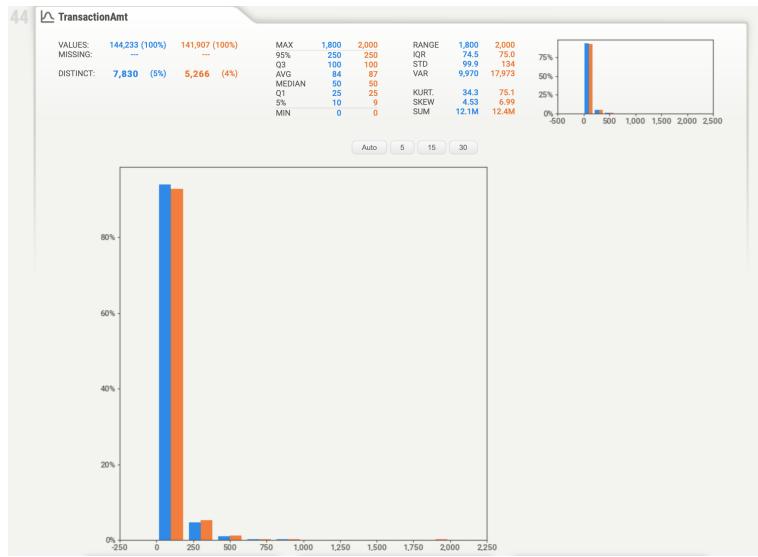


Figure 7: Transaction AMT

6.1.4. Product CD

El campo **Product_CD** hace referencia al código de producto asociado a la transacción. Podemos ver que no posee valores nulos y que hay cuatro categorías en esta variable. Lamentablemente, no poseemos el mapeo de qué representa cada categoría, por lo que no se puede obtener información del tipo de transacción.

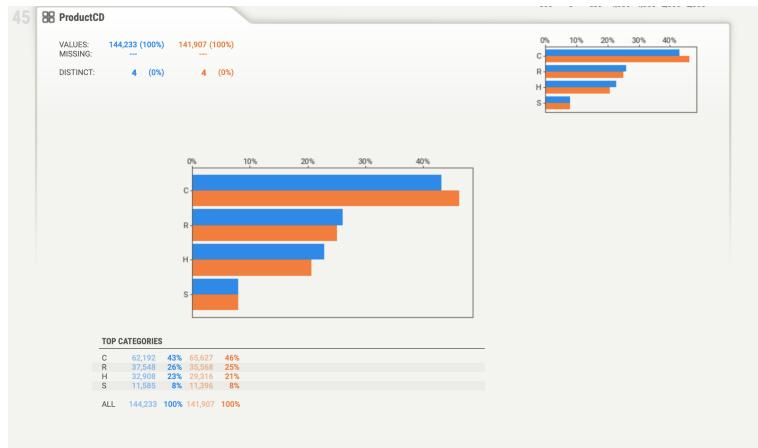


Figure 8: Product CD

6.1.5. Variables card

Las variables con formato *card_numero* representan información de la tarjeta utilizada, como tipo de tarjeta, entidad bancaria, etc. De estas, las variables de mayor interés son **card4** y **card6**.



Figure 9: Variables card

card4 representa la entidad bancaria que ha emitido la tarjeta, y podemos ver que las categorías son visa, mastercard, american express y discover. Un 60 por ciento de las tarjetas son VISA y un 30 por ciento son MasterCard, una tendencia que se mantiene en ambos conjuntos de datos.
216 Además vemos que en el conjunto de datos de testing existe un 1 por ciento de valores nulos y un valor aún menor en el conjunto de train, pero no son lo suficientemente representativos como para ser tratados.

card6 a su vez, representa el tipo de tarjeta. Nos encontramos tres tipos, débito, crédito y tarjeta prepago. Vemos que existe un número similar de valores nulos en esta variable, por lo que no es necesario su tratamiento.

6.1.6. *P_emaildomain* y *R_emaildomain*

Corresponden al dominio del correo electrónico del emisor y receptor. Podemos ver que hay 60 valores distintos, de los cuales los más representativos son *gmail* con un 40 por ciento de usuarios con estos correos y *hotmail* con un 20 por ciento. También es interesante comprobar que existe casi un 15 por ciento de usuarios que posee una cuenta de correo con dominio anónimo.

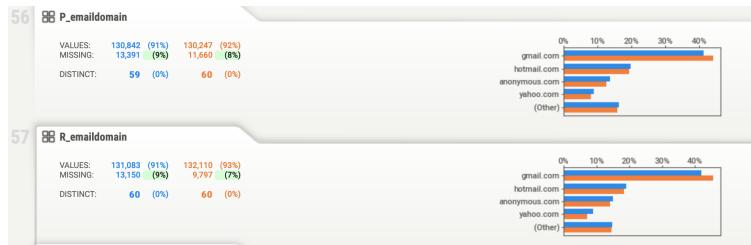


Figure 10: P_emaildomain y R_emaildomain

6.1.7. *Variables Device*

228 Estas variables aportan información acerca del dispositivo que ha utilizado el emisor de la transacción para realizarla. Hay casi paridad entre el número de transacciones realizadas en ordenador y en el móvil 11. Además, se puede ver la información del dispositivo en concreto usado para realizar la transacción (de forma detallada) en la variable *DeviceInfo*. Esta variable es categórica,

pero posee más de 2200 valores distintos, por lo que no es óptimo realizar su transformación y será posteriormente eliminada.

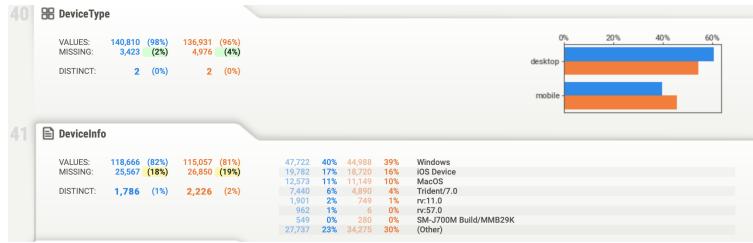


Figure 11: Variables Device

6.1.8. *isFraud*

Etiqueta del dataframe de entrenamiento, que indica el objetivo. Es una etiqueta binaria, por lo que es un problema de clasificación binaria al que nos vamos a enfrentar en próximos apartados.



Figure 12: isFraud

6.1.9. Correlación entre variables

Tal y como se ve en el reporte de *cor.html*, el grafo de correlaciones no ofrece ningún insight. Esto es debido al volumen de variables que analizar. Una opción para evitarlo es la creación de un dataframe auxiliar con las columnas de mayor interés, lo que nos genera el informe *correlaciones.html*.

En este gráfico 13 podemos ver la correlación entre las variables previamente analizadas. Los cuadrados representan asociaciones entre variables categóricas con un valor entre 0 y 1, mientras que los círculos representan correlaciones de Pearson con valores entre -1 y 1.

Podemos ver que la asociación con mayor correlación es el tipo de producto unido al importe de la transacción (un razonamiento obvio). También vemos relaciones entre los correos del emisor y el receptor, pero se deben a la distribución de los valores más que a una relación entre estas variables. Por último, cabe destacar que la variable de fraude no depende directamente de ninguna de las variables en este dataframe, por lo que tendremos que ver más adelante cuales son las variables más influyentes para este cálculo.

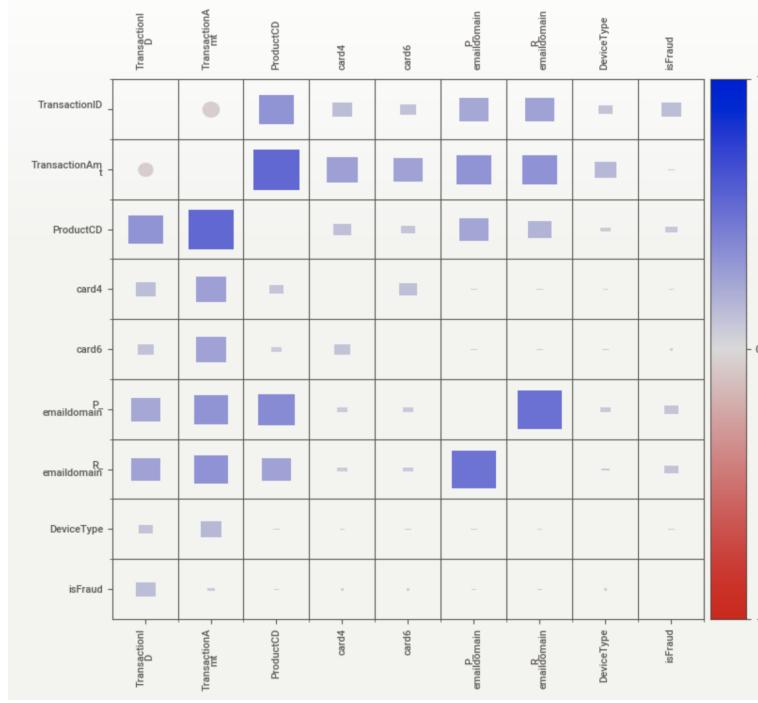


Figure 13: Gráfico de correlaciones

6.2. Aproximación al problema

En esta sección haremos una aproximación al problema de clasificación binaria presentado e intentaremos realizar un primer modelo en Python con el objetivo de tener una base a la hora de replicar este proceso en Scala con Apache Spark.

El primer paso es eliminar las columnas con un porcentaje alto de valores nulos. De forma arbitraria se ha decidido que el threshold sea un 60 por ciento. Una vez realizado este paso (tanto en el dataframde de entrenamiento como en el de testing), los conjuntos de datos se quedan con 247 columnas y 246 respectivamente (el de testing no posee la etiqueta de si es fraude o no).

In [112]:	train
Out[112]:	
	TransactionID id_01 id_02 id_05 id_06 id_11 id_12 id_13 id_15 id_16 ... V312 V313 V314 V315 V316
0	2987004 0.0 70787.0 NaN NaN 100.0 NotFound NaN New NotFound ... 0.000000 0.000000 0.000000 0.000000 0.000000
1	2987008 -5.0 98945.0 0.0 -5.0 100.0 NotFound 49.0 New NotFound ... 0.000000 0.000000 0.000000 0.000000 0.000000
2	2987010 -5.0 191631.0 0.0 0.0 100.0 NotFound 52.0 Found Found ... 90.327904 90.327904 90.327904 90.327904 0.000000
3	2987011 -5.0 221832.0 0.0 -5.0 100.0 NotFound 52.0 New NotFound ... 0.000000 0.000000 0.000000 0.000000 0.000000
4	2987016 0.0 7460.0 1.0 0.0 100.0 NotFound NaN Found Found ... 0.000000 0.000000 0.000000 0.000000 0.000000
...	...
144228	3577521 -15.0 145955.0 0.0 0.0 100.0 NotFound 27.0 Found Found ... 60.066002 60.066002 60.066002 60.066002 488.76591
144229	3577525 -5.0 172059.0 1.0 -5.0 100.0 NotFound 27.0 New NotFound ... 0.000000 NaN NaN NaN 0.000000
144230	3577529 -20.0 632381.0 -1.0 -36.0 100.0 NotFound 27.0 New NotFound ... 0.000000 0.000000 0.000000 0.000000 0.000000
144231	3577531 -5.0 55528.0 0.0 -7.0 100.0 NotFound 27.0 Found Found ... 0.000000 NaN NaN NaN 0.000000
144232	3577534 -45.0 339406.0 -10.0 -100.0 100.0 NotFound 27.0 New NotFound ... 0.000000 0.000000 0.000000 0.000000 0.000000

144233 rows × 247 columns

Figure 14: Head(Train)

El siguiente paso a realizar es el tratamiento de las variables categóricas. Hay diferentes aproximaciones a este problema, pero citando al libro [3], el uso del encoding de etiquetas es la mejor solución en este caso. El motivo es que a la hora de realizar **One Hot Encoding** cuando poseemos tantas columnas ya de por sí es que el dataframe excede el límite de memoria que tiene por defecto Python (en el peor de los casos) o que ralentice todos los procesos posteriores.

²⁶⁴ Una vez hemos tratado las variables categóricas, el siguiente paso a realizar es definir el modelo que utilizar. Al ser un problema de clasificación binaria, podemos utilizar modelos como la regresión logística y árboles, entre otros. Sin embargo, al haber realizado **LabelEncoding** y no **OneHotEncoding**, no podemos utilizar modelos lineales, por lo que utilizaremos un modelo basado en árboles.

Antes de implementar este modelo, hacemos una búsqueda de hiperparámetros óptimos utilizando **GridSearchCV**, una función del paquete de **Scikit-learn** que permite hacer una búsqueda en forma de rejilla para encontrar estos hiperparámetros.

²⁷⁶ Se aporta a la función un diccionario con listas de posibles valores de los hiperparámetros (número de hojas que puede tener cada rama, ratio de aprendizaje, tamaño máximo de elementos por rama, etc) y la función realiza combinaciones con estos valores y te devuelve aquella que haya dado mejor resultado según la métrica proporcionada (AUC). Además, hemos aportado que realice estas combinaciones con dos modelos basados en árboles distintos, el **GBDT** (Gradient Boosting Decision Trees) y **Random Forest**.

En nuestro caso, el mejor modelo es el **GBDT** con los parámetros que podemos ver en la siguiente imagen 15.

```
In [122]: gridParams = {
    'learning_rate': [0.01, 0.02, 0.05, 0.1],
    'n_estimators': [8,16,32],
    'num_leaves': [2,8,16,32], # large num_leaves helps improve accuracy but might lead to over-fitting
    'boosting_type' : ['gbdt', 'rf'], # for better accuracy -> try dart
    'objective' : ['binary'],
    'max_bin':[255, 510], # large max_bin helps improve accuracy but might slow down training progress
    'random_state' : [500],
    'colsample_bytree' : [0.65],
    'subsample' : [0.7,0.75],
}

grid = GridSearchCV(model, gridParams, verbose=1, cv=4, n_jobs=-1)
# Run the grid
grid.fit(X, y)

# Print the best parameters found
print(grid.best_params_)
print(grid.best_score_)

Fitting 4 folds for each of 512 candidates, totalling 2048 fits
[Parallel(n_jobs=-1): Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1): Done: 42 tasks | elapsed: 41.8s
[Parallel(n_jobs=-1): Done: 168 tasks | elapsed: 5.2min
[Parallel(n_jobs=-1): Done: 442 tasks | elapsed: 7.5min
[Parallel(n_jobs=-1): Done: 792 tasks | elapsed: 13.1min
[Parallel(n_jobs=-1): Done: 1242 tasks | elapsed: 19.0min
[Parallel(n_jobs=-1): Done: 1792 tasks | elapsed: 24.2min
[Parallel(n_jobs=-1): Done: 2048 out of 2048 | elapsed: 26.7min finished
('boosting_type': 'gbdt', 'colsample_bytree': 0.65, 'learning_rate': 0.05, 'max_bin': 255, 'n_estimators': 24, 'num_leaves': 8, 'objective': 'binary', 'random_state': 500, 'subsample': 0.7)
0.9387797322634741
```

Figure 15: Hyperparameter Tuning

Una vez realizado este proceso, creamos el modelo. En este caso, utilizaremos el paquete de LightGBM para definir el mismo. Usamos los parámetros que nos proporcionó la función anterior y con ello se entrena el modelo. Por último, hacemos una predicción de los valores del conjunto de testing, cuyo resultado se muestra en la siguiente imagen.

In [124]: result		
Out [124]:		
	TransactionID	isFraud
0	3663586	0.105416
1	3663588	0.142750
2	3663597	0.077761
3	3663601	0.378679
4	3663602	0.452101
...
141902	4170230	0.060211
141903	4170233	0.038309
141904	4170234	0.060211
141905	4170236	0.071816
141906	4170239	0.077761

141907 rows × 2 columns

Figure 16: Dataframe resultado

7. Ingesta

En esta sección haremos una descripción de las tecnologías utilizadas para la ingestá, Apache Flume y Apache Kafka.

7.1. Apache Flume

288

Apache Flume es un servicio distribuido de ingestá de información y que está totalmente integrado con el entorno de Hadoop, por lo que proporciona también toda la capa de seguridad que esto conlleva. Este servicio nos permite la ingestá de los CSV correspondientes a los conjuntos de datos, tanto de entrenamiento como de testing en nuestro sistema, puesto que en su source se puede configurar para que permita leer todos los nuevos archivos depositados en un directorio (la configuración del source corresponde a lo que se denomina Spooling Directory Source) [4]. Aunque Flume no asegura de forma directa la integridad del dato, puesto que si los sinks están colapsados puede perderse información (al ser su modelo de tipo push) y no proporciona una escalabilidad sencilla, estas desventajas se obvian al ser información en batch, puesto que podemos simplemente volver a realizar el proceso. Además, en el supuesto caso de que nos importase esto podemos levantar un topic de Kafka y utilizarlo como channel del Flume, lo que asegura el dato lo máximo posible.

Una vez hemos definido como se va a obtener el dato con Flume, la siguiente tarea es decidir donde dejarlo.

El sistema de almacenamiento escogido para ello es HDFS (Hadoop File Distributed System), que es el sistema nativo del entorno de Hadoop. De este entorno leerán el resto de servicios para sus propios procesos.

Para el uso de este servicio, se ha utilizado la distribución que poseemos en el Nodo Edge de nuestro clúster, por lo que la configuración de Zookeeper (el gestor de recursos del clúster) nos es indiferente.

Para Apache Flume hemos creado cuatro agentes, cada uno de ellos leyendo de una carpeta distinta (puesto que originalmente poseemos cuatro CSV que no debemos juntar) y que vuelcan el contenido en el directorio TFM/Raw/Static, donde almacenaremos los datos en crudo.

Un ejemplo de configuración se muestra en la siguiente imagen.

```
# Name the components on this agents
Agent1.sources = spool-source
Agent1.channels = memory-channel
Agent1.sinks = hdfs-sink

# Describe/configure the source
Agent1.sources.spool-source.type = spooldir
Agent1.sources.spool-source.spoolDir = /home/jmontero/TFM/Data/Static/Test/Identity
Agent1.sources.spool-source.deserializer.maxLineLength = 150000
Agent1.sources.spool-source.fileHeader = true

# Describe the sink
Agent1.sinks.hdfs-sink.type = hdfs
Agent1.sinks.hdfs-sink.hdfs.path = /user/jmontero/TFM/Raw/Static/Test/Identity
Agent1.sinks.hdfs-sink.hdfs.fileType = DataStream
Agent1.sinks.hdfs-sink.hdfs.rollSize = 0
Agent1.sinks.hdfs-sink.hdfs.rollCount = 0

# Use a channel which buffers events in memory
Agent1.channels.memory-channel.type = memory
Agent1.channels.memory-channel.capacity = 1000000
Agent1.channels.memory-channel.transactionCapacity = 1000000

# Bind the source and sink to the channel
Agent1.sources.spool-source.channels = memory-channel
Agent1.sinks.hdfs-sink.channel = memory-channel
```

Figure 17: Ejemplo de configuración de Agente en Flume

Nos encontramos tres apartados. El primero, la fuente.

- **type:** El tipo de fuente será spooldir. Esta fuente coge todos los archivos que se encuentren dentro del directorio especificado.
- **spoolDir:** Ruta al directorio del cual leer los ficheros para su ingesta.
- **deserializer.maxLength:** Tamaño máximo de línea que se almacena en el buffer. Debido a la cantidad de variables que contiene nuestro conjunto de datos, se amplia de forma considerable de su valor por defecto (5000) a un valor arbitrario (150000).
- **fileHeader:** Esta variable especifica si se añade la cabecera del archivo.

El siguiente apartado a tratar es el canal, el buffer que enviará los eventos leídos por el spoolDir.

- **type:** El tipo de canal será memoria. Teniendo en cuenta que los ficheros son estáticos y su disponibilidad es absoluta, no se cree necesario cambiar el tipo de canal para asegurar la integridad de la información. Esto hace que el proceso de ingesta sea más corto.

- **capacity:** Tamaño en bytes del buffer. Al estar utilizando los recursos del clúster, se pone como valor uno arbitrario.

324

- **transactionCapacity:** Tamaño máximo de transacción que es capaz de soportar el buffer. En este caso, ponemos el mismo valor que el campo capacity.

Por último, la definición del sink de la ingesta.

- **type:** El tipo de sink será hdfs, que es donde queremos realizar la ingesta de la información.
- **hdfs.path:** Ruta al directorio de hdfs donde ingestar los datos.
- **hdfs.fileType:** El formato del archivo será DataStream, para no comprimir el fichero resultado.
- **hdfs.rollSize,hdfs.rollCount:** Esta variables especifican que no se haga un roll a los datos basado en tamaño de archivo ni en número de eventos, respectivamente, con la idea de escribir un único fichero.

Para terminar esta sección, nos encontramos con el script que levanta a los distintos agentes

336

para realizar la ingesta.

```
/home/jmontero/flume/bin/flume-ng agent -f /home/jmontero/TFM/Flume/test_identity.conf
-Xmx1g --name Agent1 -Dflume.root.logger=INFO,console &
/home/jmontero/flume/bin/flume-ng agent -f /home/jmontero/TFM/Flume/test_transaction.conf
-Xmx1g --name Agent2 -Dflume.root.logger=INFO,console &
/home/jmontero/flume/bin/flume-ng agent -f /home/jmontero/TFM/Flume/train_identity.conf
-Xmx1g --name Agent3 -Dflume.root.logger=INFO,console &
/home/jmontero/flume/bin/flume-ng agent -f /home/jmontero/TFM/Flume/train_transaction.conf
-Xmx1g --name Agent4 -Dflume.root.logger=INFO,console
```

Figure 18: Script flume.sh

7.2. Apache Kafka

Apache Kafka es un sistema de mensajería de tipo producción-suscripción desarrollado por la fundación de software Apache, capaz de intercambiar datos entre procesos, aplicaciones y servidores.

Debido a su baja latencia y alta escalabilidad, se elige para poder realizar la ingesta de los datos en Streaming. Este servicio funciona bajo la regulación de zookeeper, y por ello es imprescindible implementarlo en un clúster a la hora de pasarlo a producción.

Al no tener acceso completo a la configuración del servicio por cuestiones de seguridad, la implementación será en modo standalone, es decir, local, y no ofrecerá el mismo resultado que una en producción. La distribución que se ha utilizado de este servicio es la que proporciona Confluent, la cual permite hacer un control de los servicios desde línea de comandos, lo que facilita en gran medida su uso.

Para ello, se ha descargado la distribución desde la web de Confluent y se ha guardado en el nodo edge. Al ya haber un servicio activo de Kafka en el clúster, la configuración base no nos sirve, ya que los puertos en los que escucha cada servicio (Zookeeper, Kafka y Kafka Connect) están ocupados.

La solución a este problema es, en el caso de la configuración de Zookeeper, modificar el puerto por defecto (se ha utilizado el puerto 2185, el cual está libre) y con Kafka y Kafka Connect se

modifican los listeners, que son los puertos donde escucha el servicio las peticiones, a los puertos 9095 y 8085, respectivamente.

Al estar realizando esta simulación de procesamiento en tiempo real en local, no es necesario modificar más variables, y se pueden dejar el resto de valores por defecto. En caso de estar en un entorno de producción, cada máquina que vaya a dar servicio a Kafka tiene que poseer su propio identificador único como broker y añadir al resto de direcciones a los listeners. Otros valores por defecto que se pueden modificar en un entorno de producción son los accesos al servicio y la seguridad del mismo, pudiendo asignar distintos tipos de seguridad a cada listener (si es una intranet se podría configurar por SSL, mientras que en una red externa se puede almacenar las conexiones seguras, etc). Todo este proceso se puede realizar también desde la Web UI que gestiona el clúster (en nuestro caso, Ambari) para facilitar el proceso, haciéndolo más visual.

Una vez hemos modificado esos parámetros, añadimos la variable de entorno **CONFLUENT_HOME** y la añadimos a la variable de entorno **PATH**. Esto nos permite utilizar la CLI de confluent para gestionar los servicios, lo cual nos ofrece la ventaja de poder instalar plugins, levantar y apagar servicios, comprobar logs, etc.

El proximo paso es crear el topic **test**, con el cual vamos a trabajar. Para ello utilizamos el daemon de kafka-topics y creamos el topic test, asignándole de replication-factor 1 (por estar trabajando de forma local).

Se ha escrito un pequeño script que levanta los servicios necesarios (Zookeeper y Kafka) para que se quede a la espera de las otras partes del proceso de Streaming, el script en Python, que genera los datos de muestra y levanta un productor al topic test previamente creado y Spark Streaming, que consume del topic test los datos que se están leyendo.

```
#Set up de variables
export CONFLUENT_HOME=/home/imoptero/TFM/kafka-setup/confluent-6.0.1;
export PATH=$CONFLUENT_HOME/bin:${PATH};

#Inicializamos servicios necesarios y eliminamos tmps anteriores
confluent local destroy;
confluent local services kafka start;
```

Figure 19: Script kafka.sh

8. Procesamiento de datos estáticos

Una vez hemos realizado la ingesta de los datos estáticos el siguiente paso es el procesamiento. Para ello se utilizará el framework de Apache Spark. Este framework es open source y tiene como principal característica la computación que puede realizar en un sistema de clusters, que puede ser realizado en memoria, lo cual agilizaría todo el proceso de filtrado de información. Posee una integración total con el entorno Hadoop, por lo que la lectura y escritura en nuestro sistema de almacenamiento está asegurada.

Mediante Apache Spark haremos una lectura y tratamiento de los ficheros y un posterior modelaje y exportado de resultados del dataset de testing.

El primer paso es la lectura de estos datos y la unión de los dataset de train_identity, train_transaction, test_identity y test_transaction respectivamente e imprimimos el schema del conjunto de datos de entrenamiento (para aplicarlo en la sección de Spark Streaming al flujo dinámico de datos).

Una vez hecho estos primeros pasos, procedemos a modificar el nombre de las columnas de test, haciendo un map que reemplaza los guiones por barras baja.

```
println("\n-----\n")
println("#####")
println("#     Modificamos columnas de test      #")
println("#####")

//Hacemos un mapeo de valores, cambiando "id-" por "id_"
val string_map = Map("id-" -> "id_")
string_map.foldLeft(test)((acc,ca) => acc.withColumnRenamed(ca._1,ca._2))

test.printSchema()
```

Figure 20: Limpieza de datos

A continuación eliminamos las columnas con valores nulos encontradas en la sección de EDA 6 y posteriormente aplicamos la función StringIndexer, la cual transforma nuestras columnas categóricas. Al no estar en una versión reciente de Apache Spark, esta función no permite añadir como parámetro una lista de columnas a las que aplicar esta transformación, por lo que la decisión elegida ha sido iterar la lista de variables categóricas y por cada una de ellas aplicar el StringIndexer, como se ve en la siguiente imagen.

```
//Lista de columnas a transformar
val cat_col = List("isFraud", "id_12", "id_15", "id_16", "id_28", "id_29", "id_31", "id_35", "id_36",
"id_37", "id_38", "DeviceType", "ProductCD", "card4",
"card6", "P_emailDomain", "R_emailDomain")

//Creamos el StringIndexer
val si = new StringIndexer()

//Creamos las variables temporales que irán almacenando los resultados de cada transformación
var tmp = train_filtered.select(col( colName = "#"))
var tmp2 = test_filtered.select(col( colName = "#"))

//Map
cat_col.foreach { s =>
    tmp = si.setInputCol(s)
        .setOutputCol("n_" + s)
        .setHandleInvalid("keep")
        .fit(tmp).transform(tmp)

    tmp2 = si.setInputCol(s)
        .setOutputCol("n_" + s)
        .setHandleInvalid("keep")
        .fit(tmp2).transform(tmp2)
}
```

Figure 21: Transformación de variables categóricas con StringIndexer

396

El siguiente paso previo a instanciar y entrenar el modelo GBDT es el uso de la función VectorAssembler. Para poder generar un modelo mediante Spark, es necesario proveer un vector de valores y la columna que corresponde a la etiqueta que queremos aprender. Por este motivo se aplica esta función para comprimir todas las columnas del dataframe en una sola denominada features.

Cabe destacar que todo este proceso se realiza también para el conjunto de datos de testing, para poder realizar una predicción correcta necesitamos hacer las mismas transformaciones en ambos conjuntos.

Una vez realizadas las transformaciones pertinentes y como paso previo al modelado, guardamos el conjunto de datos de entrenamiento que hemos procesado. El motivo es que este conjunto de

datos será utilizado por nuestro script de Python con el fin de generar muestras aleatorias y replicar un flujo dinámico.

Por último llegamos a la fase en la que aplicar el modelo. Añadimos los parámetros previamente obtenidos en la sección de EDA 6 y lanzamos el modelo.⁴⁰⁸

Para este proceso, realizamos una división del conjunto de datos de entrenamiento en entrenamiento y validación. De esta manera podremos evaluar al propio modelo (utilizando como métrica el AUC). Una vez tenemos el modelo y lo hemos evaluado, pasamos a predecir el fraude en los datos del conjunto de testing. Tanto el código utilizado como imágenes de la ejecución se muestran a continuación.

```
println("##### Modelado #####")
println("#")
println("#####")

//Dividimos train data en train y validation

val Array(training_data,valid_data) = featureDF.randomSplit(Array(0.8,0.2))

//Instanciamos el modelo

val gbt = new GBTCClassifier()
.setLabelCol("n_isFraud")
.setFeaturesCol("features")
.setLossType("logistic")
.setMaxBins(255)
.setMaxIter(24)
.setMaxDepth(8)
.setStepSize(0.05) //Learning rate
.setSubsamplingRate(0.7)

val model = gbt.fit(training_data)
println(model.toDebugString)

//Lo aplicamos a validation data
val validDF = model.transform(valid_data)

//Evaluamos la validez del modelo
val eval = new BinaryClassificationEvaluator()
.setLabelCol("n_isFraud")
.setMetricName("areaUnderROC")

val accuracy = eval.evaluate(validDF)
println(accuracy)

//Lo aplicamos a test data
val predictionDF = model.transform(testData)
predictionDF.show( numRows = 100)
```

Figure 22: Código utilizado para generar el modelo y predecir la etiqueta en el conjunto de testing

```

Else (feature 129 > 1.5)
If (feature 77 <= 2.5)
Predict: 0.5360118624967699
Else (feature 77 > 2.5)
Predict: 0.03980183099567571

21/01/09 20:09:40 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
21/01/09 20:09:40 WARN BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
0.8562735539621953

```

Figure 23: AUC correspondiente al modelo ejecutado

n_R_emaildomain	features	rawPrediction	probability	prediction
28.0 (246, [0,1,2,3,5,6... [0.71413305939089... [0.80663101322202... 0.0				
60.0 (246, [0,2,5,6,7,8... [0.71413305939089... [0.80663101322202... 0.0				
60.0 (246, [0,1,2,5,6,7... [0.71413305939089... [0.80663101322202... 0.0				
60.0 (246, [0,1,2,3,5,6... [0.71413305939089... [0.80663101322202... 0.0				
1.0 (246, [0,1,2,5,6,7... [0.65955531180860... [0.78903369941236... 0.0				
3.0 [2991797.0,0,0,61... [0.71413305939089... [0.80663101322202... 0.0				
60.0 (246, [0,1,2,5,6,7... [0.71413305939089... [0.80663101322202... 0.0				
15.0 (246, [0,1,2,3,4,5... [0.71413305939089... [0.80663101322202... 0.0				
20.0 [2993778.0,-5,0,2... [0.71413305939089... [0.80663101322202... 0.0				
60.0 (246, [0,1,2,3,4,5... [0.71413305939089... [0.80663101322202... 0.0				
10.0 (246, [0,2,5,6,7,8... [0.71413305939089... [0.80663101322202... 0.0				
0.0 [2994918.0,0,0,21... [0.71413305939089... [0.80663101322202... 0.0				
1.0 (246, [0,1,2,5,6,7... [1.08130847536202... [0.89684191121600... 0.0				
60.0 (246, [0,1,2,3,4,5... [0.71413305939089... [0.80663101322202... 0.0				
5.0 [2996904.0,0,0,14... [0.71413305939089... [0.80663101322202... 0.0				
3.0 (246, [0,2,5,6,7,8... [0.71413305939089... [0.80663101322202... 0.0				

Figure 24: Predicción obtenida al aplicar el modelo al conjunto de testing

9. Generar datos para el flujo dinámico

Una vez se ha procesado el flujo estático, ya podemos generar los datos de muestra para simular este procesamiento en tiempo real. El código necesario se muestra en las siguientes imágenes 25,26.

```

import pandas as pd
import numpy as np
from kafka import KafkaProducer
import os

#Kafka config

kafka_broker_hostname = "localhost"
kafka_consumer_portno = "9095"
kafka_broker = kafka_broker_hostname + ":" + kafka_consumer_portno
kafka_topic = "test"

producer = KafkaProducer(bootstrap_servers=kafka_broker)

#Leemos el dataset de train procesado
path = "/home/jmontero/TFM/Data/Dynamic/"

for files in os.listdir("/home/jmontero/TFM/Data/Dynamic/"):
    if str.endswith(files,".csv"):
        path+=files

#Train
train = pd.read_csv(path)

#Creamos los que van a ser los datasets generados cada X tiempo para simular
#un comportamiento dinamico en el proyecto

new_values = pd.DataFrame(columns = train.columns)

ids = train["TransactionID"].max() + 1

```

Figure 25: Script para generar datos de muestra en tiempo real 1

```

#Se crearan datos pseudo-aleatorios cada X segundos
import random
import time
n = 500
count = 0
while True:
    new_values_dict = {}

    new_values_dict["TransactionID"] = ids
    ids+=1

    for col in new_values.columns[1:]:
        new_values_dict[col] = train[col].iloc[random.randint(0,train.shape[0]-1)]

    new_values = new_values.append(new_values_dict, ignore_index = True)

    n-=1
    if(n<=0):

        count+=1
        print("Nuevos valores generados")
        print(new_values)
        n=500

        for _index in range(0, len(new_values)):

            json_values = new_values[new_values.index==_index].to_json(orient = 'records')

            producer.send(kafka_topic, bytes(json_values, 'utf-8'))

        time.sleep(20)

```

Figure 26: Script para generar datos de muestra en tiempo real 2

Utilizando el paquete de kafka-python, instanciamos un productor en Kafka, añadiendo las variables de conexión (nombre de host y puerto, así como el nombre del topic).

Una vez realizamos este paso, leemos el directorio donde se encuentran los valores del conjunto de entrenamiento procesados y recogemos el nombre del archivo csv para poder leerlo. El motivo de esto es que Spark no nombra de forma consistente a los ficheros que escribe, por lo que no es posible usar una expresión regular para recogerlo con Pandas.

Una vez hemos leído el fichero, creamos un nuevo dataframe con las mismas columnas que el conjunto de entrenamiento y guardamos el último ID del mismo.

Por último, realizamos un bucle while para generar de forma continua (hasta que se force su salida) 500 filas con valores aleatorios (excepto el identificador de la transacción, que es un valor incremental, para que no se repita). Cuando se hayan generado las 500 líneas, línea a línea del dataframe la formateamos a tipo json (para poder aplicar un schema de forma conveniente al leerlo con Spark Streaming) y mandamos la orden al productor de que lo mande, pasando a bytes con formato utf-8 su contenido.

Una vez se ha realizado el envío, el proceso duerme un tiempo de 20 segundos (para no colapsar el broker) y vuelve a empezar el bucle.

El dataframe de muestra se puede visualizar por consola al ejecutar el script 27.

[muestra] Redmine-Pro-de-Jirga-tarifa_prospe_precio_precio_minimo.csv	
Nuevos valores generados	
0	Transacciones: 14.22 14.25 14.26 14.21 14.17 14.17 14.19 14.20 ... n_id_36 n_id_37 n_id_38 n_DeviceType n_ProductID n_Cash n_Cash n_P_emaildomain n_R_emaildomain
1	30775056.0 -5.8 47748.0 0.0 -0.6 180.000000 52.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
2	30775057.0 -5.8 111687.0 9.0 -0.6 180.000000 52.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
3	30775058.0 -5.8 222398.0 0.0 0.0 180.000000 52.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
4	30775059.0 -5.8 222399.0 0.0 0.0 180.000000 52.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
400	30788010.0 -4.0 ... 336265.0 0.0 0.0 180.000000 427.0 266.0 51.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
401	30788011.0 -5.8 159657.0 0.0 0.0 180.000000 NaN 366.0 321.0 311.0 ... 1.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
402	30788012.0 -5.8 159658.0 0.0 0.0 180.000000 NaN 366.0 321.0 311.0 ... 1.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
403	30788013.0 -5.8 916479.0 0.0 0.0 180.000000 52.0 366.0 427.0 330.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
404	30788014.0 -5.8 2765779.0 4.0 -0.6 180.000000 51.0 366.0 427.0 330.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
1000 rows x 20 columns	
Nuevos valores generados	
0	Transacciones: 14.22 14.25 14.26 14.21 14.17 14.17 14.19 14.20 ... n_id_36 n_id_37 n_id_38 n_DeviceType n_ProductID n_Cash n_Cash n_P_emaildomain n_R_emaildomain
1	30775056.0 -5.8 47748.0 0.0 -0.6 180.000000 52.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
2	30775057.0 -5.8 111687.0 9.0 -0.6 180.000000 52.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
3	30775058.0 -5.8 222398.0 0.0 0.0 180.000000 52.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
4	30775059.0 -5.8 222399.0 0.0 0.0 180.000000 52.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
400	30788010.0 -4.0 ... 336265.0 0.0 0.0 180.000000 427.0 266.0 51.0 220.0 427.0 266.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
401	30788011.0 -5.8 159657.0 0.0 0.0 180.000000 NaN 366.0 321.0 311.0 ... 1.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
402	30788012.0 -5.8 159658.0 0.0 0.0 180.000000 NaN 366.0 321.0 311.0 ... 1.0 0.0 1.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
403	30788013.0 -5.8 916479.0 0.0 0.0 180.000000 52.0 366.0 427.0 330.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
404	30788014.0 -5.8 2765779.0 4.0 -0.6 180.000000 51.0 366.0 427.0 330.0 ... 0.0 0.0 0.0 1.0 1.0 0.0 0.0 0.0 0.0 0.0
1000 rows x 20 columns	

Figure 27: Visualización de la ejecución

10. Procesamiento en tiempo real

Una vez tenemos el flujo de datos en tiempo real, hay que realizar el análisis sobre el mismo. Para ello se utiliza Apache Spark, concretamente la librería de Spark Streaming, la cual permite realizar análisis sobre flujos dinámicos en formato de micro batches.

El primer paso es definir el schema que poseen nuestros datos y realizar la conexión con el topic test que hemos creado en Apache Kafka 28. La forma de lograr esto es crear una variable readStream con la opción de Kafka y añadir el nombre del host y el puerto en el que se encuentra el servicio, así como el topic al que suscribirse.

```

//Schema para los datos que vamos a leer
val schema = StructType(Seq(StructField("TransactionID",DoubleType,true), StructField("id_01",DoubleType,true), StructField("id_02",DoubleType,true),

//Conexión con Kafka
val kafka_broker_hostname = "localhost"
val kafka_consumer_portno = "9092"
val kafka_broker = kafka_broker_hostname + ":" + kafka_consumer_portno
val kafka_topic_input = "test"

val df_kafka = spark
    .readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", kafka_broker)
    .option("subscribe", kafka_topic_input)
    .load()

val df_kafka_string = df_kafka.selectExpr("CAST(value AS STRING) as value")

val df_kafka_string_parsed = df_kafka_string
    .select(from_json(colName = "value", schema).alias("data"))

val new_data = df_kafka_string_parsed.select("data.*")

new_data.printSchema()

```

Figure 28: Definición del schema y del consumidor de Kafka en Spark

Una vez hemos definido el stream de datos, si ejecutásemos (mientras estuviese el servicio de Zookeeper y Kafka levantados, así como el script de Python en funcionamiento), veríamos que irían llegando datos a Spark. Para poder recoger el valor, tenemos que acceder al campo de **value** de los mensajes de Kafka, casteándolo como String. Una vez hemos realizado esto, el siguiente paso es aplicar el schema previamente definido para poder tener nuestro dataframe.

Finalmente llegamos al análisis en tiempo real. Para realizar una predicción con el modelo previamente guardado en la parte de procesamiento de datos estáticos 8 tenemos que crear una columna que tenga el array de valores del resto de columnas. Para ello volvemos a utilizar la función VectorAssembler.

```

val assembler = new VectorAssembler()
    .setInputCols(new_data.columns)
    .setHandleInvalid("keep")
    .setOutputCol("features")

val data_df = assembler.transform(new_data)

val model = GBTClassificationModel.load(path = "/user/jmontero/TFM/Modelo")

val resultado = model.transform(data_df)

val res = resultado.select(col = "TransactionID", cols = "prediction")

res
    .writeStream
    .format("csv")
    .trigger(Trigger.ProcessingTime(intervalMs = 5000))
    .option("checkpointLocation", "/user/jmontero/TFM/Result/Dynamic")
    .option("path", "/user/jmontero/TFM/Result/Dynamic")
    .outputMode(outputMode = "append")
    .start()
    .awaitTermination()

```

Figure 29: Predicción de fraude en datos en tiempo real

Una vez hemos realizado este paso, podemos aplicar la función **transform** del modelo cargado

y seleccionar las variables que necesitamos, que son el identificador de la transacción y la etiqueta generada por el modelo al predecir si es fraude o no.

Una vez hecho esto, cada batch leído se guardará en la carpeta correspondiente de hdfs. A su vez, se ha marcado como directorio donde se guarda cada uno de los checkpoints una carpeta en el directorio resultado de hdfs. La última configuración a mencionar es el hecho de que este proceso se realiza cada 5 segundos, con la idea de que cada vez que Spark consuma del topic de Kafka, haya mensajes que recibir, y por lo tanto, no se guarden ficheros vacíos en la carpeta del resultado.

Durante la ejecución no se muestra por pantalla el dataframe, por lo que a continuación se muestran las imágenes de la conexión con Kafka y un vistazo a uno de los archivos del resultado.

```
21/01/09 17:17:59 INFO FileInputFormat: Total input paths to process : 1
21/01/09 17:18:02 INFO CodecPool: Got brand-new decompressor [.snappy]
21/01/09 17:18:04 INFO InternalParquetRecordReader: RecordReader initialized will read a total of 9022 records.
21/01/09 17:18:04 INFO InternalParquetRecordReader: at row 0. reading next block
21/01/09 17:18:04 INFO InternalParquetRecordReader: block read in memory in 4 ms. row count = 9022
21/01/09 17:18:05 INFO ConsumerConfig: ConsumerConfig values:
    allow.auto.create.topics = true
    auto.commit.interval.ms = 5000
    auto.offset.reset = earliest
    bootstrap.servers = [localhost:19092]
    check.crcs = true
    client.dns.lookup = default
    client.id =
    client.rack =
    connections.max.idle.ms = 540000
    default.api.timeout.ms = 60000
    enable.auto.commit = false
    exclude.internal.topics = true
    fetch.max.bytes = 52428800
    fetch.max.wait.ms = 500
    fetch.min.bytes = 1
    group.id = spark-kafka-source-9e7f745d-cc75-41c2-b4a9-83f1a9c6533b-143563625-driver-0
    group.instance.id = null
    heartbeat.interval.ms = 3000
    interceptor.classes = []
    internal.leave.group.on.close = true
    isolation.level = read_uncommitted
    key.deserializer = class org.apache.kafka.common.serialization.ByteArrayDeserializer
    max.partition.fetch.bytes = 1048576
    max.poll.interval.ms = 300000
```

Figure 30: Muestra de la conexión con Kafka durante la ejecución

```
(base) MacBook-Pro-de-Jorge:dinamicos jorge$ head part-00000-3ed4d581-c12c-4a30-a94a-f66530bcc2-c000.csv
3578342.0,0.0
3578343.0,0.0
3578344.0,0.0
3578345.0,0.0
3578346.0,0.0
3578347.0,0.0
3578348.0,0.0
3578349.0,0.0
3578350.0,0.0
3578351.0,0.0
```

Figure 31: Muestra del archivo resultado

11. Almacenamiento

El penúltimo paso en la arquitectura es el almacenamiento de la información en crudo y procesada para su posterior explotación. La tecnología utilizada para ello ha sido Apache Hive, debido a que en conjunto con tecnologías como Apache Impala o la Web UI Hue nos permite realizar

consultas sobre los datos en caso de necesidad. Además, los datos tienen formato de tabla, lo que hace que sea una buena decisión respecto a HBase o MongoDB.

La ingestión de las tablas Identity y Transaction será leyendo los ficheros CSV como si contuviesen String. Al no existir una manera para automatizar la creación de tablas desde el propio Apache Hive, la creación manual de estas tablas con más de 400 campos (de los cuales son conocidos aproximadamente 10 o 20) no se le ve sentido.

Una vez hecho esto, se crean las tablas finales, donde se extraen los campos de interés de cada una de ellas. El script utilizado y las tablas resultado se muestran a continuación.

```
--Creamos la base de datos si no existe y entramos en ella
CREATE DATABASE IF NOT EXISTS imontero_tfm;
USE imontero_tfm;

--Creamos la tabla de identidades si no existe ya
CREATE EXTERNAL TABLE IF NOT EXISTS identity_raw(data String)
LOCATION "/user/imontero/hive/data/identities"
TBLPROPERTIES("skip.header.line.count"="1");
LOAD DATA INPATH "/user/imontero/TFM/Raw/Static/Test/Identity/*" INTO TABLE identity_raw;
LOAD DATA INPATH "/user/imontero/TFM/Raw/Static/Train/Identity/*" INTO TABLE identity_raw;

--Creamos la tabla si no existiese en la que obtenemos los campos
--que nos interesan
CREATE TABLE IF NOT EXISTS identity_final AS
SELECT
split(data,",")[0] as TransactionID,
split(data,",")[39] as DeviceType,
split(data,",")[40] as DeviceInfo
from identity_raw;

--Creamos la tabla de transacciones si no existe ya
CREATE EXTERNAL TABLE IF NOT EXISTS transactions_raw(data String)
LOCATION "/user/imontero/hive/data/identities"
TBLPROPERTIES("skip.header.line.count"="1");
LOAD DATA INPATH "/user/imontero/TFM/Raw/Static/Train/Transaction/*" INTO TABLE transactions_raw;

--Creamos la tabla si existiese en la que obtenemos los campos
--que nos interesan
CREATE TABLE IF NOT EXISTS transactions_final AS
SELECT
split(data,",")[0] as TransactionID,
split(data,",")[1] as isFraud,
split(data,",")[3] as TransactionAmt,
split(data,",")[4] as ProductCD,
split(data,",")[8] as card4,
split(data,",")[10] as card6,
split(data,",")[15] as P_emaildomain,
split(data,",")[16] as R_emaildomain
from transactions_raw;
```

Figure 32: Creación de tablas Identity y Transaction

0: jdbc:hive2://follower-3.europe-west3-b.c.i> select * from identity_final limit 10;		
identity_final.transactionid	identity_final.devicetype	identity_final.deviceinfo
2987004	mobile	SAMSUNG SM-G892A Build/NRD90M
2987008	mobile	iOS Device
2987010	desktop	Windows
2987011	desktop	
2987016	desktop	MacOS
2987017	desktop	Windows
2987022		
2987038	mobile	Windows
2987040	desktop	
2987048	desktop	Windows

Figure 33: Tabla Identity

transactionid	isfraud	transactionamt	productcd	card4	card6	p_emaildomain	i_emaildomain
3481764	0	87.0	W	visa	debit	yahoo.com	
3481765	0	38.95	W	visa	debit	icloud.com	
3481766	0	117.0	W	visa	debit	gmail.com	
3481767	0	59.0	W	visa	debit		
3481768	0	177.0	W	mastercard	debit	yahoo.com	
3481769	0	77.0	W	visa	debit	gmail.com	
3481770	0	156.0	H	visa	credit	yahoo.com	
3481771	0	117.0	W	visa	debit	gmail.com	
3481772	0	108.0	R	american express	credit	anonymous.com	anonymous.com
3481773	0	58.95	W	mastercard	debit	yahoo.com	
3481774	0	21.0	W	visa	debit	gmail.com	
3481775	0	311.95	W	visa	debit	rocketmail.com	
3481776	0	117.0	W	mastercard	debit	icloud.com	
3481777	0	97.0	W	mastercard	debit	hotmail.com	
3481778	0	38.917	C	mastercard	credit	hotmail.com	hotmail.com
3481779	0	40.0	W	visa	debit		
3481780	0	445.0	W	visa	debit	yahoo.com	
3481781	0	186.0	R	visa	debit	comcast.net	comcast.net
3481782	0	206.0	W	visa	debit		
3481783	0	108.0	H	visa	credit	yahoo.com	gmail.com
3481784	0	64.0	W	visa	debit	att.net	
3481785	0	38.95	W	visa	debit	anonymous.com	
3481786	0	87.95	W	visa	debit	gmail.com	
3481787	0	29.0	W	mastercard	debit		
3481788	0	49.0	W	mastercard	debit	windstream.net	
3481789	0	59.0	W	visa	debit	aol.com	
3481790	0	77.0	W	visa	debit	yahoo.com	
3481791	0	82.0		mastercard	credit	gmail.com	

Figure 34: Tabla Transaction

En cuanto a la tabla de predicciones, esta se ha generado el schema manualmente, al tener sólo dos campos. Hay que añadir, que al igual que en el resto de tablas, se crea una tabla externa con una localización determinada para no mover los archivos de hdfs. Además, se añade a las dos primeras tablas la propiedad de tabla **skip.header.line.count** puesto que nuestros datos sin procesar ingestados por Flume contienen cabeceras.

A continuación el código para generar la tabla de predicciones así como una imagen de la propia tabla.

```
--Creamos la tabla de resultados si no existe ya
CREATE EXTERNAL TABLE IF NOT EXISTS predictions_raw(TransactionID int, isFraud double)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ","
LOCATION "/user/jmontero/hive/data/predictions"
TBLPROPERTIES("skip.header.line.count"="1");
LOAD DATA INPATH "/user/jmontero/TFM/Processed/Static/*" INTO TABLE predictions_raw;
LOAD DATA INPATH "/user/jmontero/TFM/Processed/Dynamic/*" INTO TABLE predictions_raw;

--Borramos la tabla si existiese y creamos una en la que obtenemos los campos
--que nos interesan
CREATE TABLE IF NOT EXISTS predictions_final AS
SELECT * from predictions_raw;
```

Figure 35: Creación de tabla Prediction

predictions_final.transactionid	predictions_final.isfraud
3424784	-5.0
3425230	-50.0
3425669	-5.0
3426034	-5.0
3428092	-5.0
3429047	-5.0
3429475	-5.0
3431036	-5.0
3432454	-5.0
3432685	-20.0

Figure 36: Tabla Prediction

480 Al realizar la explotación con PowerBI no se cree necesario realizar consultas sobre las tablas previamente mostradas.

12. Explotación

En esta sección haremos un informe acerca de las transacciones, el tipo de tarjeta que se utiliza en transacciones fraudulentas, su volumen, etc. La tecnología utilizada en esta sección es PowerBI. A continuación se muestran las imágenes, que son autoexplicativas.

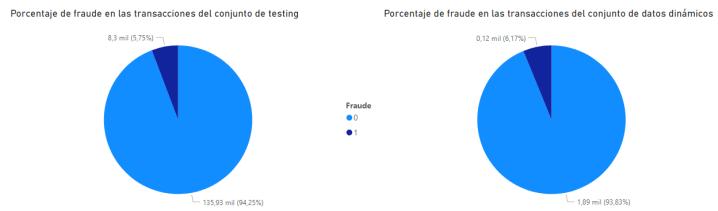


Figure 37: Informe en PowerBI

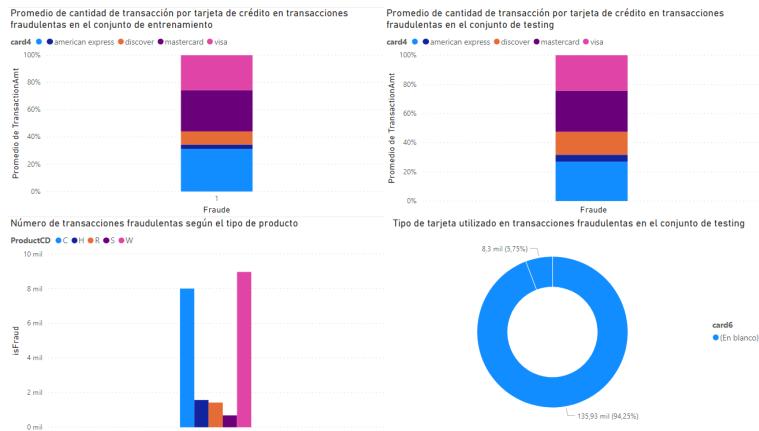


Figure 38: Informe en PowerBI

Se puede visualizar como la tendencia de valores se mantiene, y existe un volumen similar (en porcentaje) entre transacciones fraudulentas de los datos de testing y generados dinámicamente.

13. Monetización y coste del proyecto

El retorno de inversión de este proyecto no es algo evaluable a primera vista. Al fin y al cabo, el proyecto se constituye como una necesidad de nuestra start-up Fintech, y que el resto de empresas

de la competencia (y también del grupo empresarial al que pertenecemos) posee, aunque sea en menor medida.

El desarrollo de este proyecto garantiza una capa extra de seguridad en contra del fraude, lo que hace menos reacios a los inversores, aumenta la confianza que tienen los clientes con nosotros y por lo tanto nos provee de una publicidad indirecta (al estar el cliente satisfecho hay más probabilidades de que publique mediante el boca a boca o por redes sociales nuestra empresa).

Por último, este proyecto reduce costes fijos, como son las tarifas de los seguros a los que tiene estar sujetos la empresa, y podemos utilizar esos fondos como inversión de capital o como reserva del mismo.

Para calcular el coste del proyecto, primero tenemos que decidir que infraestructura final recomendar para una primera implementación. Esta es, una implementación de la infraestructura en la nube, la cual utilice las herramientas open source (y no las distribuciones de pago) de las tecnologías que hemos comentado a lo largo de todo el proyecto. El motivo de ello es que, al no proveer beneficios directos con este desarrollo, la directiva estará reacia a proveer un presupuesto importante para hacer el desarrollo de la infraestructura on-premise. Por ello, el hecho de montar la infraestructura en la nube permitirá probar resultados y discutir en un futuro donde desplegar la misma.

Los elementos a contratar en la nube se muestran en las siguientes imágenes.

Compute Engine		
8 x		
5,840 total hours per month		
VM class: regular		
Instance type: m1-megamem-96		
Region: Belgium		
Total available local SSD space 16x375 GiB		
Commitment term: 3 Years		
Estimated Component Cost: USD 22,318.98 per 1 month		
2 x		
1,460 total hours per month		
VM class: regular		
Instance type: m1-megamem-96		
Region: Belgium		
Commitment term: 3 Years		
Estimated Component Cost: USD 5,147.70 per 1 month		

Figure 39: Pricing Google Cloud : Máquinas maestro-esclavo

Persistent Disk	
Belgium	 
Zonal standard PD: 1,024 GiB	
Zonal SSD PD: 57,344 GiB	
Regional Standard PD: 1,024 GiB	
Snapshot storage: 256 GiB	
USD 9,878.02	
Internet Egress	
Source Region: Belgium	 
Premium Tier to North America: 1,024 GiB	
Premium Tier to Europe: 1,024 GiB	
Standard Tier: 1,024 GiB	
USD 317.44	
Total Estimated Cost: USD 37,662.14 per 1 month	

Figure 40: Pricing Google Cloud : Almacenamiento y red

Tal y como se ha calculado (con una permanencia de 3 años, lo que implica un descuento) el coste aproximado de esta infraestructura supone anualmente aproximadamente 360.000 euros. Este coste no se aproxima a lo que nos costaría replicar la infraestructura on premise debido al coste de licencias y por ese motivo se estudia primero la viabilidad del proyecto en la nube antes de replicarlo on premise.

Para no conseguir un sobrecoste de licencias en el ya amplio presupuesto que se va a necesitar, toda la tecnología utilizada es open source, y por lo tanto, lo único que habría que añadir es el coste del personal (puesto que el mantenimiento y costes fijos se desprecian en la nube).⁵¹⁶

14. Conclusiones

El desarrollo de este entorno de pruebas ha sido un éxito, y ha permitido tanto analizar los datos de fraude sobre el conjunto de datos de testing que poseíamos como tratar un flujo de datos en tiempo real, ingestarlo, procesarlo y predecir si es fraudulento y almacenarlo posteriormente.

Sin embargo, existen mejoras que se podrían realizar una vez se llevase a cabo este desarrollo en un entorno de producción, las cuales se van a enumerar en forma de lista a continuación.

- **EDA:** A la hora de realizar EDA, el único aspecto mejorable sería entender el resto de variables no documentadas, por lo que habría que solicitar esa información al origen de los datos por si puede ser facilitada.
- **Ingesta:** La ingesta de datos estáticos se ha realizado de forma correcta, pero el flujo de datos dinámicos es generado y no real, por lo que habría que desarrollar o utilizar los conectores necesarios para obtener los datos del origen desde Kafka. Existen numerosos conectores en Confluent para distintas plataformas y bases de datos, por lo que sería una solución rápida al problema encontrado. Además, quizás sería conveniente guardar los datos en crudo, por lo que habría que desplegar un sink de hdfs que escribiera el contenido del topic.
- **Procesamiento en tiempo real:** Se podría reentrenar el modelo cada x batches con los datos en tiempo real, evalúandolo y comparándolo con el modelo anterior. En el caso de que ofreciese un mejor rendimiento, se sobreescibiría.
- **Explotación:** Para la explotación hemos utilizado Microsoft PowerBI, pero no de la manera que se deseaba. No se ha podido realizar una conexión con el cluster mediante el puerto correspondiente a Hive usando el conector ODBC que proporciona Cloudera por no tener acceso a él, por lo que en la arquitectura a desarrollar en la nube como tarea hay que gestionar correctamente todas las interfaces y puertos de acceso a servicios. El primer motivo, para poder realizar explotación de los datos y así conectar con otros visualizadores como ElasticSearch, que hagan informes en tiempo real, como para conseguir un ahorro a la hora de controlar el tráfico de datos.

15. Lista de figuras

1	Arquitectura	4
2	Planificación del proyecto	6
3	Modelo de datos	7
4	Train Dataset	9
5	Descripción de los dataframes	10
6	Transaction ID	10
7	Transaction AMT	11
8	Product CD	11
9	Variables card	12
10	P_emaildomain y R_emaildomain	12
11	Variables Device	13
12	isFraud	13
13	Gráfico de correlaciones	14

14	Head(Train)	14
15	Hyperparameter Tuning	15
16	Dataframe resultado	16
17	Ejemplo de configuración de Agente en Flume	17
18	Script flume.sh	18
19	Script kafka.sh	19
20	Limpieza de datos	20
564	Transformación de variables categóricas con StringIndexer	20
22	Código utilizado para generar el modelo y predecir la etiqueta en el conjunto de testing	21
23	AUC correspondiente al modelo ejecutado	22
24	Predicción obtenida al aplicar el modelo al conjunto de testing	22
25	Script para generar datos de muestra en tiempo real 1	23
26	Script para generar datos de muestra en tiempo real 2	23
27	Visualización de la ejecución	24
28	Definición del schema y del consumidor de Kafka en Spark	25
29	Predicción de fraude en datos en tiempo real	25
30	Muestra de la conexión con Kafka durante la ejecución	26
31	Muestra del archivo resultado	26
32	Creación de tablas Identity y Transaction	27
576	Tabla Identity	27
34	Tabla Transaction	28
35	Creación de tabla Prediction	28
36	Tabla Prediction	28
37	Informe en PowerBI	29
38	Informe en PowerBI	29
39	Pricing Google Cloud : Máquinas maestro-esclavo	31
40	Pricing Google Cloud : Almacenamiento y red	32

16. Bibliografía

- [1] <https://www.expansion.com/expansion-empleo/2019/09/30/5d91d849e5fdea61628b46cd.html>.
- [2] <https://pypi.org/project/sweetviz/>.
- [3] A. Thakur, Approaching (Almost) Any Machine Learning Problem, 2020.
- 588 [4] <https://flume.apache.org/flumeuserguide.html>.