UNIVERSIDAD
DE GRANADA

Cloud Computing: Servicios y Aplicaciones
Memoria Práctica 4

Procesamiento de datos masivos en plataformas de cloud computing.

Carlos Enríquez López
celopez@correo.ugr.es

Introducción:

Para lograr completar todos los objetivos de esta práctica se han realizado las siguiente tareas

En primer lugar, tras investigar el fichero de datos y su cabecera en el servidor provisto por la UGR se ha creado un script de python para spark destinado a obtener las columnas asignadas. El dataset resultante a usar para machine learning se ha guardado en un solo fichero y se ha descargado del servidor para continuar el procesamiento en local.

La opción de continuar el procesamiento en local se ha tomado debido a la baja disponibilidad del cluster de la UGR. No obstante, se ha intenta seguir usando exactamente las mismas herramientas que este provee, es decir, seguir usando spark con sus bibliotecas para python. Para lograrlo, en lugar de instalar spark en local se ha usado un contenedor docker con spark y pyspark listos para usar. HDFS tampoco se ha instalado ya que una vez en local y haciendo uso de ficheros pequeños tenía poco sentido usarlo y no alteraba apenas el flujo de trabajo o el contenido del código.

Haciendo uso del contenedor se ha realizado el preprocesamiento al dataset obtenido del servidor y se le han aplicado las técnicas de machine learning oportunas. En todo momento se ha intentado emular el uso del cluster de la UGR.

Para realizar todas las tareas se han creado un total de 3 scripts de python. Se podría haber realizado uno solo con un solo flujo de trabajo si se hubiese querido ejecutarlo completamente en el servidor de hadoop de la UGR.

En el repositorio de la práctica se pueden encontrar los datasets usados, los scripts ejecutados y algunos resultados obtenidos en el procesamiento y la aplicación de clasificadores.

Repositorio: https://github.com/carlos-el/UGR_CC2_P4_Spark

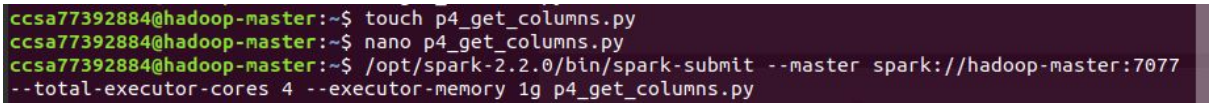
Tareas:

Obtención del dataset básico:

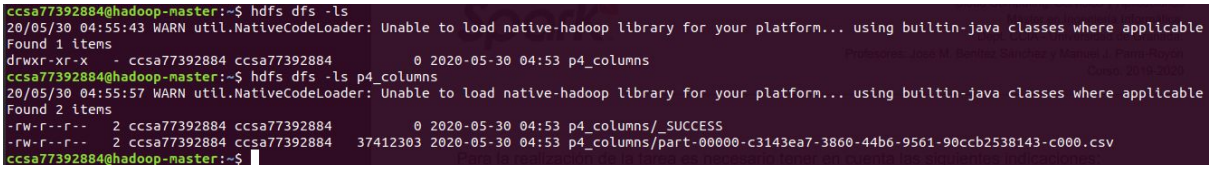
Una vez conocidas las variables concretas asignadas se ha investigado en los archivos .data y .header del servidor usando el sistema de ficheros HDFS. De aquí se ha extraído el nombre de la variable objetivo y se ha procedido a crear el script de python para reunir las columnas necesarias en un solo dataset.

El script (p4_get_columns.py) se encarga, en primer lugar, de leer el fichero de cabeceras y obtener el número columna de cada variable que necesitamos para crear el dataset. Una vez tenemos el número de columna de cada variable se lee el fichero de datos que contiene las columnas y se extraen las adecuadas en un dataset. Este dataset finalmente se guarda en HDFS como un solo fichero. Se podía haber guardado como lo hace Spark por defecto, en varios fichero, y luego haberse unido en uno sol usando el comando -getmerge de HDFS pero como el dataset no era excesivamente grande se ha optado por la primera opción.

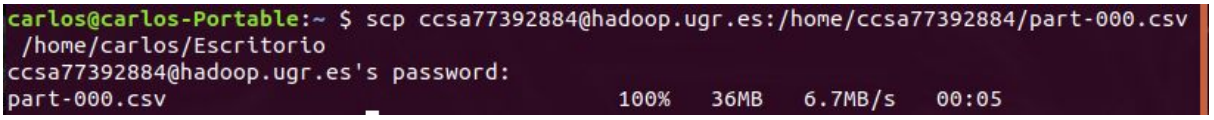
Por último se ha llevado el archivo con el dataset al directorio local del servidor desde HDFS con el comando -get y se ha descargado a la máquina del alumno con scp.



Creación y ejecución del script.



Comprobación del estado del fichero resultante



Descarga del dataset

Despliegue del contenedor de Spark en local:

Como ya se ha mencionado, para correr spark en local se ha usado un contenedor, en realidad un orquestación de contenedores con docker-compose. En entorno es el credo por el usuario 'gettyimages' y disponible en su repositorio: gettyimages/docker-spark: Docker build for Apache Spark.

Usarlo es tan sencillo como clonar el repositorio y hacer un pull de la imagen de spark. Después podemos modificar el docker-compose.yml que viene en el repositorio para levantar el número de esclavos que queramos y con las opciones configuración deseadas en el nodo máster y en sus esclavos. En nuestro caso se ha usado la configuración por defecto, un solo esclavo con dos cores y 1GB de memoria.

Una vez configurado solo tenemos que ejecutar los siguientes comandos:

```
docker-compose up
docker exec -it docker-spark_master_1 /bin/bash
```

y ya estamos listos para lanzar tareas en Spark con:
```
bin/spark-submit --master spark://master:7077
--total-executor-cores 1 --executor-memory 1g <tarea.py>
```

Si se va ha hacer uso de la MlLib es necesario instalar previamente en el contenedor numpy. Esto se puede hacer directamente desde dentro o modificando el dockerfile que integra el repositorio clonado.
También es importante destacar que la versión de Spark incluida en el contendor es la 2.4.1 y no la 2.2.0 como en el cluster de la UGR.

A éste método se le ha visto mucho potencial ya que facilita enormemente el poder desplegar una máquina virtual potente en un proveedor cloud y disponer de Spark de manera rápida, sencilla y bastante configurable.

Preprocesamiento del dataset:

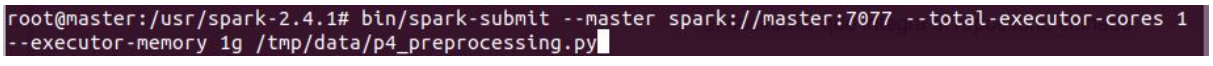
Con spark disponible y el dataset en local se ha realizado un preprocesamiento simple del dataset, de nuevo, mediante un script de python (p4_preprocessing.py).

El funcionamiento es simple. Leemos el dataset de un volumen añadido al contenedor de Spark. Como el dataset está desbalanceado se identifica la clase mayoritaria (0) y se obtiene el ratio de desbalance con respecto a la minoritaria. El ratio obtenido ha sido de 2. Después se ha procede a realizar un undersampling de la clase mayoritaria en función del ratio de desbalance.

Por último se divide el dataset resultante en dos conjunto uno para entrenamiento y otro para validación y se crean los ficheros correspondientes.

Un ‘summary’ del dataset original, del reducido y de los dataset de entrenamiento y validación pueden encontrarse en la carpeta ‘preprocessing_data’ del fichero .zip adjunto a la memoria.

La ejecución se ha realizado con el siguiente comando:



Aplicación de clasificadores:

Los clasificadores aplicado al conjunto de datos han sido ‘LogisticRegression’, ‘NaiveBayes’, y ‘LinearSVC’. No obstante, tras leer los conjuntos de entrenamiento y test ha sido necesario modificar los dataframes resultantes ya que MlLib usa un

formato concreto que no se puede guardar de manera normal en un fichero. este pequeño preprocesamiento ha constado únicamente de modificar el nombre de alguna variable, agrupar todas las variables no objetivo en una única columna de vectores y crear otra columna copia de la anterior pero con los valores normalizados (necesaria para ejecutar Naive Bayes). Después se ha procedido con los experimentos.

Las métricas obtenidas para cada modelo también se encuentran dentro de la carpeta 'ml_data' del archivo .zip de la práctica.

Regresión logística:

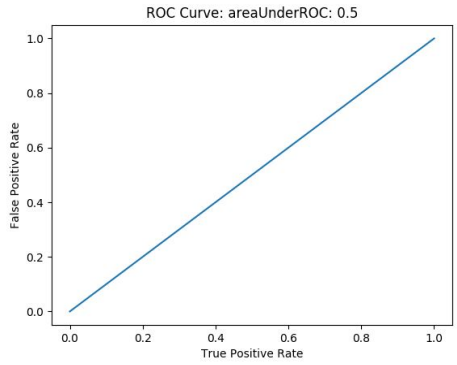
Modelo 1:

Parametrización:

```
LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8, family="binomial")
```

Precisión: `0.49954876099555284`

Área bajo la curva ROC: `0.5`



En este modelo hemos sacado la gráfica de la curva ROC, no obstante tiene poco sentido repetirlo ya que para el resto de modelos va a ser prácticamente la misma gráfica.

Modelo 2:

Parametrización:

```
LogisticRegression(maxIter=30, regParam=0.2, elasticNetParam=0.9, family="binomial")
```

Precisión: `0.49954876099555284`

Área bajo la curva ROC: `0.5`

Podemos comprobar que la bondad de ambos modelos es catastrófica, siendo prácticamente iguales, la clasificación es equivalente o tal vez incluso peor que una clasificación aleatoria y esto se refleja en su curva ROC.

Naive Bayes:

Modelo 1:

Parametrización:

```
NaiveBayes(smoothing=1.0, modelType="multinomial")
```

Precisión: `0.5293124132406934`

Área bajo la curva ROC: `0.521562733523721`

Modelo 2:

Parametrización:

```
NaiveBayes(smoothing=5.0, modelType="multinomial")
```

Precisión: `0.5293160376503677`

Área bajo la curva ROC: `0.5215627287420872`

Aunque con respecto a los anteriores podemos decir que estos modelos son algo mejores, siguen siendo muy malos. Entre ellos la diferencia es mínima a pesar de un cambio en el valor ‘smoothing’ muy grande entre ellos.

Máquinas de vectores de soporte:

Modelo 1:

Parametrización: `LinearSVC(maxIter=10, regParam=0.1)`

Precisión: `0.5300227975368512`

Área bajo la curva ROC: `0.5478002746684064`

Modelo 2:

Parametrización: `LinearSVC(maxIter=50, regParam=0.02)`

Precisión: `0.5311536133552248`

Área bajo la curva ROC: `0.5476990204231116`

A pesar de la poca variación entre ellos y con los modelos anteriores, podemos decir que estos modelos son los mejores (aun siendo bastante malos). De entre ellos el modelo número 2 es algo mejor en cuanto a precisión siendo por tanto el mejor de todos los modelos probados.

Conclusión:

Los resultados para todos los métodos son bastante malos. Los únicos que muestran un indicio de poder predecir ‘algo’ son Naives Bayes y LinearSVC con el mayor valor de precisión y de área bajo la curva ROC aunque podríamos decir que ambos también son un fracaso. La poca diferencia entre todos ellos probablemente es causada por su baja bondad ya que si todos los modelos son muy malos muy difícilmente puedan mostrar diferencias en sus clasificaciones. A pesar de todo el resultado era esperado a causa de que las columnas asignadas han sido aleatorias y de un dataset de 600 variables por lo que cualquier buen resultado habría sido fruto del azar en la asignación de unas columnas mejores. Además tampoco se

tenía ninguna información de las variables por lo que resulta difícil realizar un preprocesamiento a las mismas.