Procesamiento de Big Data con SparkR y Hadoop

Cloud Computing: Servicios y aplicaciones

Pedro Manuel Gómez-Portillo López

gomezportillo@correo.ugr.es

71722388Q

4 de junio de 2019

Índice

1. Parte 1. Hadoop....3
2. Parte 2. Algoritmos de clasificación con SparkR....3
2.1. Desarrollo....4
2.2. Resultados....6
2.2.2. Regresión logística....6
2.2.3. Random forest....6
2.2.4. K-means....6
2.2.5. Mejor algoritmo....7
3. Conclusiones....7
4. Anexos....8
4.1. Código en R de la práctica....8

1. Parte 1. Hadoop

No ha podido realizarse la parte de Hadoop ya que no disponíamos de los permisos necesarios en el servidor.

2. Parte 2. Algoritmos de clasificación con SparkR

En esta parte se usará SparkR para realizar en R el procesamiento de un dataset en el servidor Hadoop y se guardarán los resultados también en él, para luego ejecutar varios algoritmos de predicción y comparar sus resultados, siguiendo el guion de prácticas.

- Procesar el fichero de datos para crear uno nuevo
  - Consta de as 5 primeras columnas y la variable de clase (la última columna). En total tendrá 6 columnas.
  - Se llamará */user/tuusuario/ECDB-2012.small.training*
- Equilibrar el fichero resultante de datos para que tenga el mismo número de registros de las clases *0* y *1*.
- Aplicar al menos tres clasificadores de la MLlib al conjunto de entrenamiento nuevo creado.
- Aplicar el modelo creado al conjunto de entrenamiento:
- Obtener los resultados clasificación para cada uno de los modelos utilizando al menos dos variaciones en los hiperparámetros de los algoritmos de clasificación.
- Crear una tabla con los resultados para los tres algoritmos y las variantes de parámetros aplicadas, para conocer la efectividad de la clasificación aplicada.
- Identificar el algoritmo que ha obtenido los mejores resultados.

2.1. Desarrollo

El primer problema que se encontró al empezar a trabajar en la práctica fue no saber qué dataset utilizar ya que, aunque en el guión pone que utilicemos */user/mp2019/ECDB-2012.training* para el entrenamiento y */user/mp2019/ECDB-2012.test* para las pruebas solo pudieron encontrarse los que se ven a continuación.

```
mp71722388@hadoop-master: $ hdfs dfs -ls /user/mp2019
19/06/04 11:49:29 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Found 28 items
-rw-r--r--   2 root         supergroup       711174 2019-05-13 18:13 /user/mp2019/20000_ECBDL14_10tst.data
-rw-r--r--   2 root         supergroup     17683919 2019-05-13 18:14 /user/mp2019/500000_ECBDL14_10tst.data
-rw-r--r--   2 root         supergroup       177876 2019-05-20 19:09 /user/mp2019/5000_ECBDL14_10tst.data
-rw-r--r--   2 root         supergroup    102747181 2019-05-13 18:14 /user/mp2019/ECBDL14_10tst.data
-rw-r--r--   2 root         supergroup         2093 2019-05-13 18:04 /user/mp2019/WordCount.java
```

Por ello, se ha decidido trabajar con */user/mp2019/ECBDL14_10tst.data* y dividirlo en dos conjuntos, entrenamiento y prueba, como se verá a continuación.

Tras configurar el entorno de trabajo como se indica en el guión y como puede verse en el código de la práctica (anexo 1) se leyó el fichero indicado, se seleccionaron sus columnas *f1-f5* y *class* con `small_tr <- select(tttm, f1, f2, f3, f4, f5, class)` y se volvió a guardar.

Como en el enunciado pedía que se balancearan los datos con la clase *class*, se utilizó,

```
num_regs_pre <- as.integer(collect(count(small_tr)))
summarize( group_by(small_tr, class), count = n(), percent= n()/num_regs_pre*100.0)
```

para ver cuántos valores había de cada clase, obteniendo

```
> summarize( group_by(small_tr, class), count = n(), percent= n()/num_regs_pre*100.0)
# Source: spark<?> [?? x 3]
  class   count percent
  <int>   <dbl>   <dbl>
1     0 2849280   98.3
2     1   48637    1.68
```

Es decir, había 2849280 filas de la clase 0 (un 98,3%) y 48637 de la clase 1 (1,68%), por lo que para igualarlas se utilizó,

```
small_tr_eq <- ovun.sample(class ~ ., data = small_tr, p = 0.5, seed = 1, method =
"under")$data
```

obteniendo

```
> small_tr_eq <- ovun.sample(class ~ ., data = small_tr, p = 0.5, seed = 1, method = "under")$data
> num_regs_post <- as.integer(collect(count(small_tr_eq)))
> summarize( group_by(small_tr_eq, class), count = n(), percent= n()/num_regs_post*100.0)
# A tibble: 2 x 3
  class count percent
  <int> <int>   <dbl>
1     0 48645    50.0
2     1 48637    50.0
```

Es decir, 48637 filas de cada clase, o un 50% del total cada una, tras lo que se guardaron los datos con

```
spark_write_csv(small_tr_eq_tbl, path=BALANCED_FILEPATH, delimiter = ",", header=TRUE, mode = 'overwrite')
```

Aunque antes fue necesario convertir los datos del formato *data.frame* a *tbl_frame* para que SparkR pudiera trabajar con ellos con la función *copy_to()*.

Luego se realizaron dos particiones, entrenamiento y test, con el 75% y el 25% de los datos, respectivamente, utilizando

```
partitions <- small_tr_eq_tbl %>% sdf_random_split(training = 0.75, test = 0.25, seed = 123)
```

Una vez los datos estaban preprocesados y preparados se pudo comenzar a ejecutar los algoritmos de predicción. Como el código de estos algoritmos ya está en el anexo 1, no se volverá a copiar a continuación y se pasa a presentar los resultados.

Por último, para crear una tabla con los resultados se han utilizado las funciones *print()* y *paste()* de R.

```
# Mostrar los resultados

base = "Accuray de la ejecución del modelo de"
base_tiempo = "y ha tardado"

print(paste(base, "Regresión Lineal", RELI_ACCURACY, base_tiempo, tiempo_reli))
print(paste(base, "Regresión Logística", RELO_ACCURACY, base_tiempo, tiempo_relo))
print(paste(base, "Random Forest", RF_ACCURACY, base_tiempo, tiempo_rf))
print(paste(base, "K-means", K_MEANS_ACCURACY, base_tiempo, tiempo_k_means))
```

2.2. Resultados

2.2.1. Regresión lineal

Fue el primer algoritmo utilizado, pero la ejecución falló con el error

```
# Error: java.lang.IllegalArgumentException: Field "label" does not exist
```

y aunque se consultó su API[1] y varios foros no se consiguió solucionar, por lo que se pasó al siguiente algoritmo.

2.2.2. Regresión logística

Este algoritimo pudo ejecutarse correctamente, para lo que tardó **~4.72 segundos** y se obtuvo un *accuracy* del **~53,73%**.

2.2.3. Random forest

Los resultados de este algoritmo fueron un tiempo de ejecución de **~6,35 segundos** y un *accuracy* del **~57,66%**.

2.2.4. K-means

Por último también se intentó ejecutar el algoritmo k-means, aunque con resultados similares al de regresión lineal; el siguiente error no permitió terminar su ejecución, y aunque se intentó no ha podido solucionarse.

```
# Error: java.lang.IllegalArgumentException: requirement failed: Column prediction
must be of type DoubleType but was actually IntegerType.
```

1 https://www.rdocumentation.org/packages/sparklyr/versions/0.6.4/topics/ml_linear_regression

2.2.5. Mejor algoritmo

Por tanto, teniendo en cuenta tanto el tiempo de ejecución como el *accuracy* obtenido, el algoritmo con los mejores resultados ha sido el **Random Forest** ya que, aunque ha ha invertido un 50% más de tiempo de ejecución, ha obtenido resultados un **~5% mejores**, lo que en estos algoritmos es preferible, ya que es su principal objetivo

3. Conclusiones

Esta práctica me ha servido como primera toma de contacto a Hadoop y Spark en un lenguaje con el que no me siento muy códmo, R, por lo que me ha requerido un esfuero extra por aprenderlo.

En conclusión, me alegro mucho de haberla realizado, ya que he adquirido conocimientos que pueden serme necesarios en mi futura vida laboral.

4. Anexos

4.1. Código en R de la práctica

```
# PRINCIPALES REFERENCIAS
# https://github.com/manuparra/taller_SparkR/blob/master/Parte
%202.%20S06.%20Mineria%20de%20datos%20y%20Machine%20Learning%20con%20sparklyr.ipynb
# https://github.com/manuparra/TallerH2S#create-the-spark-environment

# install.packages('sparklyr')
# install.packages('dplyr')
# install.packages('ROSE')

if (nchar(Sys.getenv("SPARK_HOME")) < 1) {
  Sys.setenv(SPARK_HOME = "/opt/spark-2.2.0/")
}

library(SparkR, lib.loc = c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib")))
sparkR.session(master = "local[*]", sparkConfig = list(spark.driver.memory = "1g"),
enableHiveSupport=FALSE)

# sparkR.version()
# spark_install(version = "2.2.0")

library(sparklyr)
library(dplyr)
library(ROSE)

sc <- spark_connect(master = "local", version = "2.2.0")

# FILE = "5000_ECBDL14_10tst.data"
# FILE = "20000_ECBDL14_10tst.data"
# FILE = "500000_ECBDL14_10tst.data"
FILE = "ECBDL14_10tst.data"
FILEPATH = paste("hdfs://hadoop-master/user/mp2019/", FILE, sep="")
tttm <- spark_read_csv(sc, name="tttm", path=FILEPATH, delimiter = ",",
header=TRUE, overwrite = TRUE)
# df5000 <- read.df("hdfs://hadoop-master/user/mp2019/5000_ECBDL14_10tst.data",
source="csv")

small_tr <- select(tttm, f1, f2, f3, f4, f5, class)
spark_write_csv(small_tr, path="hdfs://hadoop-master/user/mp2019/mp71722388/ECDB-
2012.small.training", delimiter = ",", header=TRUE,  mode = 'overwrite')

# Ahora equilibramos el fichero para que  tenga el mismo número de registros de las
clases 0 y 1

num_regs_pre <- as.integer(collect(count(small_tr)))
summarize( group_by(small_tr, class), count = n(), percent= n()/num_regs_pre*100.0)
# Source: spark<?> [?? x 3]
#   class   count percent
#   <int>   <dbl>   <dbl>
# 1     0 2849280   98.3
# 2     1   48637    1.68
```

```r
small_tr_eq <- ovun.sample(class ~ ., data = small_tr, p = 0.5, seed = 1, method =
"under")$data

num_regs_post <- as.integer(collect(count(small_tr_eq)))
summarize( group_by(small_tr_eq, class), count = n(), percent=
n()/num_regs_post*100.0)
# A tibble: 2 x 3
#   class count percent
#   <int> <int>   <dbl>
# 1     0 48645    50.0
# 2     1 48637    50.0

small_tr_eq_tbl <- copy_to(sc, small_tr_eq)

## Guardar los resultados sobreescribiendo el fichero
BALANCED_FILEPATH = "hdfs://hadoop-master/user/mp2019/mp71722388/ECDB-
2012.small.training"
spark_write_csv(small_tr_eq_tbl, path=BALANCED_FILEPATH, delimiter = ",",
header=TRUE,  mode = 'overwrite')

## Si ya tenemos el fichero guardado podemos saltarnos todo lo anterior y empezar
desde aqui tras leerlo
# small_tr_eq_tbl <- spark_read_csv(sc, name="small_tr_eq_tbl",
path="hdfs://hadoop-master/user/mp2019/mp71722388/ECDB-2012.small.training",
delimiter = ",", header=TRUE, overwrite = TRUE)

# Creamos un conjunto de entrenamiento con el 75% de los datos

# REF. https://stackoverflow.com/a/35343912/3594238
partitions <- small_tr_eq_tbl %>% sdf_random_split(training = 0.75, test = 0.25,
seed = 123)

# Aplicamos 3 clasificadores
my_features = c("f1","f2","f3","f4","f5")

## Regresion Lineal
tiempo_pre = Sys.time()
reli_model <- ml_linear_regression(partitions$training, f1~f5, label_col="class")
reli_predicted <- ml_predict(reli_model, newdata = partitions$test)
RELI_ACCURACY = ml_multiclass_classification_evaluator(reli_predicted,
metric_name="accuracy")
tiempo_reli = Sys.time() - tiempo_pre
# Error: java.lang.IllegalArgumentException: Field "label" does not exist

## Regresion Logistica
tiempo_pre = Sys.time()
relo_model <- partitions$training %>% ml_logistic_regression(response = "class",
features = my_features)
relo_predicted <- ml_predict(relo_model, newdata = partitions$test)
RELO_ACCURACY = ml_multiclass_classification_evaluator(relo_predicted,
metric_name="accuracy")
tiempo_relo = Sys.time() - tiempo_pre
# [1] 0.6018519

## Random Forest
tiempo_pre = Sys.time()
training_cv <- partitions$training %>% select(f1,f2,f3,f4,f5,class) %>%
mutate(class1=as.character(class)) %>% select(f1,f2,f3,f4,f5,class=class1)
```

```
randfor_model <- ml_random_forest(training_cv, response="class",
features=my_features)
randfor_predicted <- ml_predict(randfor_model, partitions$test)
RF_ACCURACY = ml_multiclass_classification_evaluator(randfor_predicted,
metric_name="accuracy")
tiempo_rf = Sys.time() - tiempo_pre
# [1] 0.6315789

## K-means
tiempo_pre = Sys.time()
k_means_model <- partitions$training %>% select(f1,f2,f3,f4,f5,class) %>%
ml_kmeans(features = "class", centers = 3)
k_means_predicted <- ml_predict(k_means_model, newdata = partitions$test)
K_MEANS_ACCURACY = ml_multiclass_classification_evaluator(k_means_predicted,
metric_name="accuracy")
tiempo_k_means = Sys.time() - tiempo_pre
# Error: java.lang.IllegalArgumentException: requirement failed: Column prediction
must be of type DoubleType but was actually IntegerType.


# Mostrar los resultados
base = "Accuray de la ejecución del modelo de"
base_tiempo = "y ha tardado"

# print(paste(base, "Regresión Lineal", RELI_ACCURACY, base_tiempo, tiempo_reli))
print(paste(base, "Regresión Logística", RELO_ACCURACY, base_tiempo, tiempo_relo))
print(paste(base, "Random Forest", RF_ACCURACY, base_tiempo, tiempo_rf))
# print(paste(base, "K-means", K_MEANS_ACCURACY, base_tiempo, tiempo_k_means))

# [1] "Accuray de la ejecución del modelo de Regresión Logística 0.537398507340277
y ha tardado 4.71406221389771"
# [1] "Accuray de la ejecución del modelo de Random Forest 0.576645664068977 y ha
tardado 6.35683274269104"

sparkR.session.stop()
```