
**UNIVERSIDAD DE GRANADA**

Cloud Computing

# *Big Data* en *Cloud Computing*

Práctica 4

Víctor Vázquez Rodríguez
victorvazrod@correo.ugr.es
76664636R

Máster universitario en Ingeniería Informática

Curso 2019/20

# Índice

# 1. Introducción

En esta práctica, se va resolver un problema de clasificación con una gran cantidad de datos sobre un clúster de nodos para aprender cómo se trabaja con *Big Data* en el *cloud*. El procesamiento se realiza con Spark sobre el sistema de archivos distribuido de Hadoop. He implementado el flujo de trabajo en Python haciendo uso de la librería `pyspark`.

En las siguientes secciones de este documento, se explica cómo se han llevado a cabo las distintas tareas necesarias, desde la carga de datos hasta el entrenamiento de los modelos para los distintos clasificadores. Todo ello estará acompañado de pequeños fragmentos de código para las partes más relevantes. El código completo se puede encontrar en el repositorio de GitHub creado para esta práctica [1].

[1] Repositorio de GitHub: https://github.com/Varrrro/spark-ml

## 2. Lectura y filtrado de datos

El conjunto de datos a utilizar para el problema de clasificación posee más de 600 atributos, de los cuáles solo tengo que usar los 6 que me han sido asignados para la práctica, que son: *PSSM_r2_-3_S*, *PSSM_central_0_P*, *PSSM_r1_1_Y*, *PSSM_r1_-3_R*, *PSSM_r1_0_R*, *PredSS_central_2*. Estos atributos, además de la clase (columna *class*) deben ser filtrados del conjunto de datos y colocados en un fichero llamado `filteredC.small.training`.

El conjunto de datos inicial se encuentra en formato ARFF y dividido en dos archivos: uno con la cabecera y las etiquetas y otro con los datos como tal. El proceso que se ha seguido para la extracción de los datos deseados es el siguiente:

1. Leer el archivo con las cabeceras.
2. Extraer las filas que empiezan con las etiquetas *@inputs* o *@outputs*.
3. Eliminar estas etiquetas, las comas y separar por palabras (nombres de los atributos).
4. Leer el archivo de los datos.
5. Poner a cada columna el nombre de su atributo.
6. Seleccionar las columnas deseadas y escribir al archivo destino.

El código encargado de hacer este filtrado se puede ver a continuación:

```
headers =
    sc.textFile("/user/datasets/ecbdl14/ECBDL14_IR2.header")
    .filter(lambda line: "@inputs" in line or "@outputs" in line)
    .flatMap(lambda line: line.replace(",", "").split())
    .filter(lambda word: word != "@inputs" and word != "@outputs")
    .collect()

data = sqlc.read.csv(
    "/user/datasets/ecbdl14/ECBDL14_IR2.data",
    header=False,
    inferSchema=True
)

for i, colname in enumerate(data.columns):
    data = data.withColumnRenamed(colname, headers[i])

data = data.select("PSSM_r2_-3_S", "PSSM_central_0_P",
    "PSSM_r1_1_Y", "PSSM_r1_-3_R", "PSSM_r1_0_R",
    "PredSS_central_2", "class")

data.write.csv(path, header=True)
```

Como se puede ver, se hace uso de los RDD de Spark para hacer el filtrado de los atributos del fichero de cabeceras. Además, cabe destacar que, aunque no se vea en el fragmento de código mostrado, al realizar la carga de datos se comprueba si existe el fichero `filteredC.small.training`, en cuyo caso no se realiza el filtrado para no gastar tiempo de procesamiento.

## 3. Pre-procesamiento

Antes de poder entrenar los clasificadores con los datos, se deben realizar algunas tareas de pre-procesamiento. Para empezar, observamos que las clases no están balanceadas, teniendo unos 680.000 positivos y alrededor de 1.300.000 negativos. Por ello, aplicamos *undersampling* para balancear las clases obteniendo la proporción de registros positivos sobre el total y haciendo `sample` de los registros negativos para obtener la misma proporción, tal y como se puede ver en el fragmento de código siguiente:

```
positives = data.filter(data["class"] == 1)
negatives = data.filter(data["class"] == 0)
ratio = float(positives.count()) / float(data.count())
sampled_negatives = negatives.sample(False, ratio)
data = positives.union(sampled_negatives)
```

También tenemos que dividir el conjunto de datos en las particiones para entrenamiento y prueba, que conseguimos con la función `randomSplit` de Spark. En mi caso, he decidido usar un 70 % de los registros para entrenamiento y el resto para prueba.

Una vez realizadas estas dos tareas, ya podríamos entrenar los modelos. No obstante, tenemos que tener en cuenta que los clasificadores toman como entrada un vector de características, no una serie de columnas. Spark nos proporciona `VectorAssembler` para crear juntar las columnas deseadas en un vector y colocarlo en una nueva columna del conjunto de datos. Hay otro impedimento, y es que la columna *PredSS_central_2* contiene caracteres de texto, los cuales no acepta `VectorAssembler`. Usamos entonces `StringIndexer` para convertir estos caracteres en valores numéricos que si pueden incorporarse a los vectores de características. Tanto `StringIndexer` como `VectorAssembler` se colocan en un `Pipeline` de Spark al que luego se añaden los distintos clasificadores. Estos *pipelines* permiten entrenar los flujos de procesamiento completos para los datos, obteniendo modelos que trabajan directamente sobre los conjuntos de datos sin tener que realizar cada etapa de forma manual.

```
def create_preprocess_pipeline():
    si = StringIndexer(
        inputCol="PredSS_central_2",
        outputCol="PredSS_central_2_indexed"
    )

    va = VectorAssembler(
        inputCols=["PSSM_r2_-3_S", "PSSM_central_0_P",
            "PSSM_r1_1_Y", "PSSM_r1_-3_R", "PSSM_r1_0_R",
            "PredSS_central_2_indexed"],
        outputCol="features"
    )
```

```
    return Pipeline(stages=[si, va])
```

# 4. Clasificadores

Se han entrenado modelos para 3 clasificadores diferentes: *Random Forest*, perceptrón multicapa y regresión logística. El entrenamiento de estos modelos es idéntico para los 3 clasificadores:

1. Definición del clasificador.
2. Definición del *grid* de parámetros.
3. Definición del `CrossValidator`.
4. Entrenamiento del modelo usando el conjunto de entrenamiento.

En el *grid* de parámetros (`ParamGrid`) se definen una serie de valores posibles para los parámetros del clasificador (o de cualquier fase del *pipeline*) para que el `CrossValidator` pruebe todas las combinaciones posibles. Este `CrossValidator` divide el conjunto de datos dado en entrenamiento y validación según los *folds* o pliegues que se especifiquen (en nuestro caso, 3) y comprueba el modelo entrenado para cada combinación de parámetros con los datos de validación. Al final, la combinación con un mayor valor de AUC es considerada como el mejor modelo y se usa para realizar una predicción sobre el conjunto de prueba.

En las siguientes subsecciones se van a presentar los fragmentos de código encargados de la definición y entrenamiento de estos modelos, así como los resultados obtenidos para las distintas combinaciones de parámetros y los resultados finales sobre el conjunto de prueba para el mejor modelo de cada clasificador.

## 4.1. *Random Forest*

Para este clasificador, los parámetros cuyos valores variamos son el número de árboles y la medida de impureza usada. En el siguiente fragmento de código, se pueden apreciar todos los pasos que ya se han indicado en la introducción anterior.

```
def train_random_forest(prepro, train):
    rf = RandomForestClassifier(
        featuresCol="features",
        labelCol="class",
    )

    rf_grid = ParamGridBuilder() \
        .addGrid(rf.numTrees, [10, 50, 100]) \
        .addGrid(rf.impurity, ["gini", "entropy"]) \
        .build()

    rf_cv = CrossValidator(
        estimator=Pipeline(stages=[prepro, rf]),
```

```
        estimatorParamMaps=rf_grid,
        evaluator=BinaryClassificationEvaluator(labelCol="class"),
        numFolds=3
    )

    return rf_cv.fit(train)
```

Los resultados obtenidos se pueden ver en el cuadro 1, en el que podemos apreciar que ambas métricas de impureza arrojan resultados muy similares y que el aumento del número de árboles sí que incrementa el valor del AUC, aunque de manera mínima.

Cuadro 1: Resultados de los modelos RF en la validación cruzada

| **numTrees** | **impurity** | **AUC** |
|---|---|---|
| 10 | *gini* | 0.5942809 |
| 50 | *gini* | 0.5949606 |
| 100 | *gini* | 0.595012 |
| 10 | *entropy* | 0.5942139 |
| 50 | *entropy* | 0.5949326 |
| 100 | *entropy* | 0.59501994 |

La mejor combinación de las que hemos probado es la de la última fila del cuadro 1, con 100 árboles y usando entropía como medida de impureza. La predicción realizada por este modelo para el conjunto de prueba arrojó un valor de AUC de 0.59517723.

## 4.2. Perceptrón multicapa

El segundo clasificador es una red neuronal sencilla. En este caso, el *grid* de parámetros define una serie de arquitecturas (número de capas y neuronas) diferentes para entrenar y probar. Todas estas arquitecturas tienen una capa de entrada con 6 neuronas (una por característica) y otra capa de salida con 2 neuronas (clasificación binaria).

```
def train_multilayer_perceptron(prepro, train):
    mlp = MultilayerPerceptronClassifier(
        featuresCol="features",
        labelCol="class",
        predictionCol="rawPrediction",
        maxIter=100
    )

    mlp_grid = ParamGridBuilder() \
```

```
        .addGrid(mlp.layers, [[6, 4, 2], [6, 8, 4, 2], [6, 8,
            2]]) \
        .build()

    mlp_cv = CrossValidator(
        estimator=Pipeline(stages=[prepro, mlp]),
        estimatorParamMaps=mlp_grid,
        evaluator=BinaryClassificationEvaluator(labelCol="class"),
        numFolds=3
    )

    return mlp_cv.fit(train)
```

Los resultados obtenidos se pueden ver en el cuadro 2, donde apreciamos que la mejor arquitectura ha sido la de [6, 8, 2]. Posteriormente, este modelo ganador ha realizado una predicción sobre el conjunto de prueba con un AUC de 0.52218163.

Cuadro 2: Resultados de los modelos MLP en la validación cruzada

| **layers** | **AUC** |
|---|---|
| [6, 4, 2] | 0.5102396 |
| [6, 8, 4, 2] | 0.52180004 |
| [6, 8, 2] | 0.5255682 |

## 4.3. Regresión logística

El último clasificador es el de regresión logística multinomial. En este caso, se prueban distintas combinaciones de parámetros de regularización y elasticidad.

```
def train_logistic_regression(prepro, train):
    lr = LogisticRegression(
        featuresCol="features",
        labelCol="class",
        maxIter=100,
        family="multinomial"
    )

    lr_grid = ParamGridBuilder() \
        .addGrid(lr.regParam, [0.1, 0.01, 0.001]) \
        .addGrid(lr.elasticNetParam, [0.5, 0.6, 0.8]) \
        .build()

    lr_cv = CrossValidator(
```

```
        estimator=Pipeline(stages=[prepro, lr]),
        estimatorParamMaps=lr_grid,
        evaluator=BinaryClassificationEvaluator(labelCol="class"),
        numFolds=3
    )

    return lr_cv.fit(train)
```

Los resultados de la validación cruzada de estos modelos se puede ver en el cuadro 3. Como se puede ver, los resultados van siendo mejores al reducir el valor de la regularización, siendo la mejor combinación la de la última fila del cuadro, con 0.001 de regularización y 0.8 de elasticidad. Este modelo ganador ha obtenido un AUC de 0.5817315 en su predicción sobre el conjunto de prueba final.

Cuadro 3: Resultados de los modelos LR en la validación cruzada

| **regParam** | **elasticNetParam** | **AUC** |
|---|---|---|
| 0.1 | 0.5 | 0.57934934 |
| 0,1 | 0.6 | 0.5 |
| 0.1 | 0.8 | 0.5 |
| 0.01 | 0.5 | 0.58033985 |
| 0.01 | 0.6 | 0.5803268 |
| 0.01 | 0.8 | 0.58027285 |
| 0.001 | 0.5 | 0.5812404 |
| 0.001 | 0.6 | 0.58124447 |
| 0.001 | 0.8 | 0.5812488 |

# 5. Conclusiones

Para facilitar la comparación de los modelos entrenados, se han recopilado en el cuadro 4 los resultados obtenidos en las predicciones realizadas sobre el conjunto de prueba que ya se han comentado a lo largo de la sección anterior.

Cuadro 4: Resultados sobre el conjunto de prueba

| **Clasificador** | **AUC** |
|---|---|
| *Random Forest* | 0.59517723 |
| Perceptrón multicapa | 0.52218163 |
| Regersión logística | 0.5817315 |

El mejor modelo que hemos entrenado es, como se puede ver, el *Random Forest*, con un AUC de 0.59517723, seguido de la regresión logística y con el perceptrón multicapa en último lugar. El resultado del clasificador *Random Forest* podría ser incluso mejor usando un número de árboles mayor a los que se han probado.

En general, los resultados son bastante malos, sin llegar ninguno de los modelos siquiera al 0.6 de AUC y creo que esto se debe a que tan solo se usan 6 atributos de los más de 600 disponibles en el conjunto de datos inicial, así que pienso que la calidad que es posible obtener está limitada.