Cloud Computing: Servicios y Aplicaciones


# UNIVERSIDAD DE GRANADA

## Práctica 4
## Procesamiento y minería de datos en Big Data con Spark sobre plataformas cloud

**Autor**
Juan Manuel Castillo Nievas


Máster Profesional en Ingeniería Informática 2020-2021

—

Granada, 6 de junio de 2021

# Índice

# 1. Introducción

El objetivo de esta práctica es consolidar el conocimiento adquirido durante el curso sobre la infraestructura de cómputo para Big Data y su despliegue en plataformas Cloud Computing.

En esta práctica se ha propuesto hacer manejo del sistema **HDFS** (Hadoop Distributed File System) [1] para obtener unos datos y hacer operaciones básicas para trabajar con esos ficheros.

Cuando se obtenga el conjunto de datos desde HDFS, se usará la plataforma **Spark** [2] para poner en uso diferentes métodos y técnicas de preprocesamiento de datos para grandes volúmenes de datos. Para ello, se propone el uso de la biblioteca **MLLib** (disponible en Spark) [3], la cual dispone de distintos métodos y modelos de programación para dichas técnicas de preprocesamiento.

Se explorarán 3 técnicas de construcción de clasificadores usando MLLib, cada técnica con 2 parametrizaciones distintas y obteniendo una evaluación para cada una de ellas. Dichas evaluaciones servirán para un estudio final que servirá como conclusión para averiguar qué modelos y parámetros han botenido mejores resultados. Las 3 técnicas de construcción de clasificadores que han sido usadas han sido:

- Random Forest
- Decision Trees
- Gradient Boosted Trees

Como lenguaje de programación se ha usado **Python 3** [4], tal y como se ha venido usando a lo largo de las prácticas de esta asignatura.

# 2. Conjunto de datos

El conjunto de datos sobre el que se van a aplicar las técnicas de clasificación se encuentra en dos directorios de HDFS:

- **hdfs://user/datasets/ecbdl14/ECBDL14_IR2.header** aquí se encuentran las cabeceras del conjunto de datos.
- **hdfs://user/datasets/ecbdl14/ECBDL14_IR2.data**: aquí se encuentran los datos.

Las características de este conjunto de datos son:

- 600 atributos
- 1 atributo de clasificación binaria
- 2060000 registros

- 4Gbytes

A cada alumno se le ha asignado un total de **6 columnas** (más la columna con la variable clasificadora). En este caso, las 6 columnas asignadas han sido las siguientes:

- PredSA_central
- PSSM_r2_0_K
- PSSM_r2_1_G
- PSSM_r1_1_H
- PSSM_r2_0_E
- PredRCH_r1_-4

En la Figura 1 se muestra una visualización del conjunto de datos.

| | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 1 | PredSA_central | PSSM_r2_0_K | PSSM_r2_1_G | PSSM_r1_1_H | PSSM_r2_0_E | PredRCH_r1_-4 | class |
| 2 | 0 | 1 | -4 | -1 | 0 | 0 | 0 |
| 3 | 3 | -5 | 6 | -2 | -5 | 3 | 1 |
| 4 | 4 | 0 | -4 | 0 | 3 | 1 | 0 |
| 5 | 3 | -4 | -7 | 3 | -3 | 0 | 0 |
| 6 | 3 | 0 | -3 | -1 | 2 | 0 | 0 |
| 7 | 3 | -2 | -3 | -2 | -2 | 3 | 1 |
| 8 | 0 | 0 | -3 | -1 | -2 | 3 | 1 |
| 9 | 3 | 1 | -5 | -6 | 3 | 4 | 0 |
| 10 | 4 | -4 | -1 | 0 | -4 | 0 | 0 |
| 11 | 2 | -5 | -4 | 0 | 4 | 3 | 1 |
| 12 | 3 | -4 | 7 | -5 | -2 | 3 | 1 |
| 13 | 4 | 1 | -2 | -3 | 2 | 1 | 0 |
| 14 | 4 | 1 | -1 | 0 | 4 | 0 | 0 |
| 15 | 0 | -2 | -2 | -5 | -2 | 3 | 0 |
| 16 | 0 | -5 | -5 | 2 | -5 | 0 | 0 |
| 17 | 3 | 1 | -2 | -5 | 4 | 2 | 0 |
| 18 | 0 | -3 | -2 | -3 | -3 | 0 | 0 |
| 19 | 3 | -3 | -4 | 0 | -3 | 3 | 1 |
| 20 | 0 | -7 | -3 | -4 | -7 | 3 | 0 |

Figura 1: Visualización del conjunto de datos

Algo importante a destacar de estas columnas es que todas ellas son de tipo **Integer**, salvo la columna **PredRCH_r1_-4** que es de tipo **String**. En el código implementado se ha convertido esta variable a tipo **Integer** para que no suponga ningún problema a la hora de aplicar las técnicas de clasificación.

# 3. Preprocesamiento de datos

## 3.1. Eliminación de valores perdidos

Se ha detectado que la variable **PredRCH_r1_-4** contiene valores perdidos. Esto se ha detectado porque tal y como se ha comentado, esta variable inicialmente era de tipo **String**. Las columnas que contienen el valor $X$, son en realidad valores perdidos.

Se ha decidido **eliminar las filas que contengan valores perdidos**. Para ello, se ha usado la siguiente función:

```
def delete_null_rows(df):

    return df.dropna()
```

## 3.2. Undersampling

Este problema de clasificación está **desbalanceado**. Esto quiere decir que hay más ejemplos que pertenecen una clase que a la otra. Para paliar con este problema se ha decidido aplicar **undersampling**. Esta técnica lo que hace es balancear el problema de clasificación disminuyendo el número de ejemplos de la clase mayoritaria de forma que su porcentaje sea el mismo que el número de ejemplos en la clase minoritaria.

Dependiendo de si la clase mayoritaria es la clase positiva o negativa, se balanceará una clase u otra, tal y como se muestra en el siguiente código:

```
def undersampling(df):
    pos = df.filter(df['class']==0)
    neg = df.filter(df['class']==1)

    total_pos = pos.count()
    total_neg = neg.count()

    if total_pos > total_neg:
        fr = float(total_neg)/float(total_pos)
        sampled = pos.sample(withReplacement=False, fraction=fr)
        return sampled.union(neg)
    else:
        fr = float(total_pos)/float(total_neg)
        sampled = neg.sample(withReplacement=False, fraction=fr)
        return sampled.union(pos)
```

En la Tabla 1 se muestra el número de ejemplos en ambas clases antes y después del preprocesamiento.

| | **Antes** | **Después** |
|---|---|---|
| **Clases positivas** | 1375458 | 662492 |
| **Clases negativas** | 687729 | 662493 |

Tabla 1: Balanceo de clases antes y después del preprocesamiento

# 4. Técnicas de clasificación

A continuación se van a explicar las 3 técnicas de clasificación implementadas. Para cada técnica de clasificación, el particionamiento de los datos es el mismo: **70 % de entrenamiento** y **30 % de test**.

Las métricas usadas para evaluar la calidad de las técnicas han sido **accuracy** (precisión) y **AUC** (área bajo la curva).

Cabe destacar que se ha creado una función exclusivamente para **realizar las predicciones** (código mostrado en la siguiente página) y obtener las métricas de cada técnica de clasificación. Esta función evalúa un determinado modelo usando los parámetros correspondientes. Para esta función se ha usado las métricas de evlauación que proporciona MLLib [5]. La función creada recibe 4 argumentos:

- **estimator:** la técnica de clasificación utilizada
- **paramGrid:** los parámetros usados
- **train:** conjunto de entrenamiento
- **test:** conjunto de test

Esta función utiliza la función **print()** para mostrar en la salida tanto la precisión como el área bajo la curva.

```
def predictions(estimator, paramGrid, train, test):
    train_validator = TrainValidationSplit(estimator=estimator,
                                           estimatorParamMaps=paramGrid,
                                           evaluator=BinaryClassificationEvaluator(),
                                           trainRatio=train_sample,
                                           seed=5000)
    model = train_validator.fit(train)
    predictions = model.transform(test)

    # Convierto ‘prediction‘ y ‘label‘ a float, de lo contrario no funciona
    pred_and_label = predictions.select("prediction","label")
    pred_and_label = pred_and_label.withColumn("prediction",
        func.round(pred_and_label['prediction']).cast('float'))
    pred_and_label = pred_and_label.withColumn("label",
        func.round(pred_and_label['label']).cast('float'))

    metrics=MulticlassMetrics(pred_and_label.rdd.map(tuple))

    evaluator = BinaryClassificationEvaluator()

    auc = evaluator.evaluate(predictions)
    accuracy = round(metrics.accuracy*100, 3)

    print("Accuracy %s" % accuracy)
    print("AUC %s" % auc)
```

## 4.1. Random Forest

La documentación de esta técnica se puede encontrar en el siguiente enlace:

https://spark.apache.org/docs/latest/ml-classification-regression.html#random-forest-classifier

Los parámetros que se han tenido en cuenta para esta técnica han sido los siguientes:

- **MaxDepth:** indica la profundidad máxima que alcanza el árbol
- **MaxBins:** indica el número de segmentos en los que se van a dividir las variables continuas cuando se aplique la discretización.
- **NumTrees:** indica el número de árboles a entrenar.

#### 4.1.1. Primera versión

La configuración aplicada a esta primera versión de esta técnica de clasificación es la siguiente:

```
rf = RandomForestClassifier(labelCol="label", featuresCol="features")

paramGridRF = (ParamGridBuilder()
               .addGrid(rf.maxDepth, [2, 5, 10])
               .addGrid(rf.maxBins, [5, 10, 20])
               .addGrid(rf.numTrees, [5, 20, 50])
               .build())
```

Evaluación del modelo creado:

- **Accuracy:** 58.78 %
- **AUC:** 0.62

#### 4.1.2. Segunda versión

La configuración aplicada a esta segunda versión de esta técnica de clasificación es la siguiente:

```
rf = RandomForestClassifier(labelCol="label", featuresCol="features")

paramGridRF = (ParamGridBuilder()
               .addGrid(rf.maxDepth, [2, 4, 6])
               .addGrid(rf.maxBins, [10, 20, 30])
               .addGrid(rf.numTrees, [10, 30, 50])
               .build())
```

Evaluación del modelo creado:

- **Accuracy:** 58.35 %
- **AUC:** 0.61

## 4.2. Decision Trees

La documentación de esta técnica se puede encontrar en el siguiente enlace:

`https://spark.apache.org/docs/latest/ml-classification-regression.html#decision-tree-classifier`

Los parámetros que se han tenido en cuenta para esta técnica han sido los siguientes:

- **MaxDepth:** indica la profundidad máxima que alcanza el árbol
- **MaxBins:** indica el número de segmentos en los que se van a dividir las variables continuas cuando se aplique la discretización.

### 4.2.1. Primera versión

La configuración aplicada a esta primera versión de esta técnica de clasificación es la siguiente:

```
dt = DecisionTreeClassifier(labelCol="label", featuresCol="features")

dtparamGrid = (ParamGridBuilder()
               .addGrid(dt.maxDepth, [2, 5, 10, 15, 20])
               .addGrid(dt.maxBins, [10, 20, 30, 40, 50])
               .build())
```

Evaluación del modelo creado:

- **Accuracy:** 57.85 %
- **AUC:** 0.58

### 4.2.2. Segunda versión

La configuración aplicada a esta segunda versión de esta técnica de clasificación es la siguiente:

```
dt = DecisionTreeClassifier(labelCol="label", featuresCol="features")

dtparamGrid = (ParamGridBuilder()
               .addGrid(dt.maxDepth, [2, 5, 10, 15, 20])
               .addGrid(dt.maxBins, [10, 20, 30, 40, 50])
               .build())
```

Evaluación del modelo creado:

- **Accuracy:** 57.82 %
- **AUC:** 0.58

## 4.3. Gradient-Boosted Trees

La documentación de esta técnica se puede encontrar en el siguiente enlace:

`https://spark.apache.org/docs/latest/ml-classification-regression.html#gradient-boosted-tree-classifier`

Los parámetros que se han tenido en cuenta para esta técnica han sido los siguientes:

- **MaxDepth:** indica la profundidad máxima que alcanza el árbol
- **MaxBins:** indica el número de segmentos en los que se van a dividir las variables continuas cuando se aplique la discretización.
- **MaxIter:** indica el número máximo de iteraciones que se van a hacer.

### 4.3.1. Primera versión

La configuración aplicada a esta primera versión de esta técnica de clasificación es la siguiente:

```
gb = GBTClassifier(labelCol="label", featuresCol="features")

gbparamGrid = (ParamGridBuilder()
               .addGrid(gb.maxDepth, [2, 5, 10])
               .addGrid(gb.maxBins, [10, 20, 40])
               .addGrid(gb.maxIter, [5, 10, 20])
               .build())
```

Evaluación del modelo creado:

- **Accuracy:** 58.56 %
- **AUC:** 0.62

### 4.3.2. Segunda versión

La configuración aplicada a esta segunda versión de esta técnica de clasificación es la siguiente:

```
gb = GBTClassifier(labelCol="label", featuresCol="features")

gbparamGrid = (ParamGridBuilder()
                .addGrid(gb.maxDepth, [2, 3, 5])
                .addGrid(gb.maxBins, [10, 20, 30])
                .addGrid(gb.maxIter, [2, 5, 10])
                .build())
```

Evaluación del modelo creado:

- **Accuracy:** 58.39 %
- **AUC:** 0.62

# 5. Conclusiones

En esta práctica se han aplicado distintas técnicas de clasificación usando la librería **MLLib** disponible en **Spark**. Además, se ha usado **HDFS** para la obtención del conjunto de datos.

En la Tabla 2 se muestran los resultados de las métricas obtenidas en cada técnica de clasificación.

| | **Accuracy** | **AUC** |
|---|---|---|
| **Random Forest V1** | 58.78 % | 0.62 |
| **Random Forest V2** | 58.35 % | 0.61 |
| **Decision Tree V1** | 57.85 % | 0.58 |
| **Decision Tree V2** | 57.82 % | 0.58 |
| **Gradient-Boosted Tree V1** | 58.65 % | 0.62 |
| **Gradient-Boosted Tree V2** | 58.39 % | 0.62 |

Tabla 2: Comparación de los modelos de clasificación

Se pueden sacar las siguientes conclusiones:

- El **mejor modelo de clasificación** es el **Random Forest** con los **parámetros de la versión 1**. Los resultados son un **58.78 %** de precisión y un área bajo la curva de **0.62**
- El **peor modelo de clasificación** es el **Decision Tree**, tanto para la versión 1 como para la versión 2. Ninguna de las dos versiones llega a alcanzar el 58 % de precisión.
- Realmente no se debe hacer mucho caso a que el mejor porcentaje obtenido sea inferior a un 60 %, pues el conjunto de datos posee 40 columnas y se ha hecho una predicción basándose solamente en 6 columnas. El preprocesamiento de datos es muy importante y en este conjunto de datos no se ha prestado demasiada atención, pues solo se han eliminado valores perdidos y se ha aplicado undersampling.
- Los modelos de clasificación **Random Forest** y **Gradient-Boosted Tree** obtienen resultados bastante parecidos, aunque este último tarda muchísimo más tiempo en terminar su ejecución, por lo que se puede decir que **Random Forest es mejor en relación resultado-tiempo de ejecución.**

# 6. Bibliografía


[1] HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[2] Spark. https://spark.apache.org/.

[3] MLLib. https://spark.apache.org/mllib/.

[4] Python. https://www.python.org/.

[5] MLLib: Binary Classification. https://spark.apache.org/docs/latest/mllib-evaluation-metrics.html#binary-classification.