UNIVERSIDAD DE GRANADA

PRÁCTICA 3

Cloud Computing: Servicios y Aplicaciones

Alumna: Cristina del Águila Martín, cristinadam@correo.ugr.es
DNI: 20100057N

ÍNDICE

1. Configuración del entorno con Docker

Primero he creado mi estructura de carpetas con los siguientes comandos

```
mkdir practica3
cd practica3
mkdir spark
```

1.1. Fichero docker-compose.yaml

A continuación creo el archivo *docker-compose.yaml*, en el que voy a explicar cómo levantar varios contenedores y cómo se comunican entre ellos.
En este archivo defino lo siguiente:

- 1 contenedor NameNode (HDFS)
- 1 contenedor DataNode (HDFS)
- 1 contenedor Spark, con Python instalado

1.2. Dockerfile

A continuación creo el archivo Dockerfile en la carpeta `/spark (spark/Dockerfile)`, aquí le explico a Docker cómo construir el contenedor Spark con Python y las librerías que van a hacer falta.

1. Parto de una imagen con Spark ya instalado
2. Añado Python 3 y pip
3. Instalo numpy

1.3. Levantar los contenedores

Desde la carpeta practica3 utilizo el siguiente comando:

```
docker compose up -d
```

Lo cual va a levantar los siguientes contenedores:

- namenode: donde se gestiona el sistema de archivos HDFS
- datanode: almacén de los bloques de datos en el sistema HDFS
- spark: entorno para ejecutar tus scripts de análisis con PySpark

1.4. Acceso a los contenedores

Para conectarme al contenedor spark utilizo:

```
docker exec -it spark bash
```

Para conectarme al contenedor namenode utilizo:

```
docker exec -it namenode bash
```

2. Subida del dataset a HDFS

Descargo el dataset y lo copio en `~/data`

```
cp half_celestial.csv ~/data/
```

En el contenedor namenode hago lo siguiente:

```
hdfs dfs -mkdir -p /user/CCSA2425/cristinadam
hdfs dfs -put /data/half_celestial.csv /user/CCSA2425/cristinadam/
hdfs dfs -ls /user/CCSA2425/cristinadam
```

3. Clasificador con PySpark

En primer lugar, creo el archivo `clasificador.py`

1. Inicio la sesión de Spark

Ahora creo una sesión de Spark con el nombre "`Practica3-MLlib`", para usar las funcionalidades de procesamiento distribuido de PySpark. También uso la siguiente línea para reducir el nivel de los mensajes del sistema a solo errores para evitar información innecesaria durante la ejecución.

```
# 1. Crear sesión de Spark
spark = SparkSession.builder.appName("Practica3-MLlib").getOrCreate()
spark.sparkContext.setLogLevel("ERROR")
```

2. Carga del da

Leo el archivo CSV que había almacenado en HDFS, con cabecera y separación por punto y coma.

```
# 2. Cargar el dataset desde HDFS
path = "hdfs://namenode:8020/user/CCSA2425/cristinadam/half_celestial.csv"
df = spark.read.csv(path, header=True, inferSchema=True, sep=';')
```

3. Exploración inicial

Imprimo la estructura del DataFrame para conocer los tipos de cada columna. También calculo los valores nulos por columna y se enseño la distribución de clases (galaxy y star) en la variable objetivo.

```
# 3. Exploración de datos
print("\n Estructura del dataset:")
df.printSchema()

print("\n Valores nulos por columna:")
df.select([count(when(isnan(c) | col(c).isNull(), c)).alias(c) for c in df.columns]).show()

print("\n Distribución de clases:")
df.groupBy("type").count().show()
```

4. Preprocesamiento

Transformo la variable `type` a formato numérico con `StringIndexer`. Selecciono 10 variables numéricas como características, las agrupo en un único vector con `VectorAssembler (features_raw)`, y luego aplico un tipo de escalado según el modelo:

Para los modelos que toleran valores negativos (como regresión logística o Random Forest), aplico `StandardScaler`, que centra los datos en media cero y varianza uno. En cambio, para Naive Bayes (que exige entradas no negativas), uso `MinMaxScaler`, que transforma los datos al rango [0,1].

```
# 4. Preprocesamiento
# Convertir la variable de salida (galaxy/star) a numérica
indexer = StringIndexer(inputCol="type", outputCol="label")

# Columnas numéricas que uso como features
features = [
    "expAB_z", "i", "q_r", "modelFlux_r", "expAB_i",
    "expRad_u", "q_g", "psfMag_z", "dec", "psfMag_r"
]

# Ensamblo y normalizo features
assembler = VectorAssembler(inputCols=features, outputCol="features_raw")
scaler = StandardScaler(inputCol="features_raw", outputCol="features")

# Pipeline para modelos que admiten negativos
pipeline_std = Pipeline(stages=[indexer, assembler, StandardScaler(inputCol="features_raw", outputCol="features")])
data_std = pipeline_std.fit(df).transform(df).select("features", "label")

# Pipeline para NaiveBayes: features no negativas
pipeline_minmax = Pipeline(stages=[indexer, assembler, MinMaxScaler(inputCol="features_raw", outputCol="features")])
data_minmax = pipeline_minmax.fit(df).transform(df).select("features", "label")
```

Así, genero dos conjuntos distintos: `data_std` para los dos modelos y `data_minmax` exclusivamente para Naive Bayes.

5. División del conjunto de datos

Divido los dos conjuntos (`data_std` y `data_minmax`) en entrenamiento (80%) y prueba (20%) como se indica en el guión de prácticas, usando una semilla fija para garantizar reproducibilidad.

```
# 5. Partición de datos
# Para modelos estándar
train_std, test_std = data_std.randomSplit([0.8, 0.2], seed=42)

# Para NaiveBayes
train_minmax, test_minmax = data_minmax.randomSplit([0.8, 0.2], seed=42)
```

6. Definición de modelos

Defino seis modelos en total, agrupados en tres técnicas distintas:

- Regresión logística con dos configuraciones distintas de regularización (regParam).
- Random Forest, variando el número de árboles y la profundidad máxima.
- Naive Bayes, con dos valores de suavizado (smoothing).

```
modelos = {
    "LogisticRegression_1": LogisticRegression(maxIter=10, regParam=0.01),
    "LogisticRegression_2": LogisticRegression(maxIter=20, regParam=0.1),
    "RandomForest_1": RandomForestClassifier(numTrees=10, maxDepth=5),
    "RandomForest_2": RandomForestClassifier(numTrees=30, maxDepth=10),
    "NaiveBayes_1": NaiveBayes(smoothing=1.0),
    "NaiveBayes_2": NaiveBayes(smoothing=0.5)
}
```

7. Entrenamiento y evaluación

Recorro cada modelo y lo entreno sobre el conjunto de datos correspondiente (train_std o train_minmax). Luego, calculo la métrica AUC sobre el conjunto de prueba.

```
# 7. Entrenamiento y evaluación
print("\n Resultados:")
for nombre, modelo in modelos.items():
    if "NaiveBayes" in nombre:
        m = modelo.fit(train_minmax)
        predicciones = m.transform(test_minmax)
    else:
        m = modelo.fit(train_std)
        predicciones = m.transform(test_std)

    auc = evaluator.evaluate(predicciones)
    print(f"{nombre}: AUC = {auc:.4f}")
```

8. Cierre de Spark

Cierro la sesión de Spark para liberar los recursos utilizados

```
spark.stop()
```

4. Ejecución del script

Como no quería tener que estar copiando el archivo "`clasificador.py`" cada vez que lo modificaba, he optado por montar mi carpeta en el contenedor.
Para esto hago lo siguiente:

1. En mi docker-compose.yaml añado la siguiente línea

```
volumes:
    - ./:/workspace
```

Esto hace que todo lo que tengo en `practica3/` (mi carpeta local) aparezca en el contenedor Spark como `/workspace`.

2. Reinicio los contenedores

```
docker compose down
docker compose up -d
```

Esto aplica los cambios del `docker-compose.yaml`

3. Ejecuto el script directamente desde la carpeta montada
Entro al contenedor Spark

```
docker exec -it spark bash
```

y ejecuto el script desde `/workspace`

De esta forma, cada vez que edite `clasificador.py` en mi máquina local, el contenedor verá los cambios automáticamente, sin necesidad de volver a copiarlo.

5. Evaluación de los modelos y resultados obtenidos

El conjunto de datos utilizado contiene 1.000.000 instancias, distribuidas casi perfectamente entre las dos clases objetivo: galaxias (500.002) y estrellas (499.998). Todas las variables numéricas están completas, sin presencia de valores nulos, lo que ha facilitado el preprocesamiento sin necesidad de imputaciones.

```
Distribución de clases:
+------+------+
|  type| count|
+------+------+
|galaxy|500002|
|  star|499998|
+------+------+
```

Para abordar el problema de clasificación binaria, he usado 3 técnicas de construcción de clasificadores, probando 2 parametrizaciones de cada uno de los algoritmos. He usado como métrica principal el Área Bajo la Curva ROC (AUC), que permite comparar el rendimiento de los clasificadores teniendo en cuenta tanto la sensibilidad como la especificidad del modelo.

```
modelos = {
    "LogisticRegression_1": LogisticRegression(maxIter=10, regParam=0.01),
    "LogisticRegression_2": LogisticRegression(maxIter=20, regParam=0.1),
    "RandomForest_1": RandomForestClassifier(numTrees=10, maxDepth=5),
    "RandomForest_2": RandomForestClassifier(numTrees=30, maxDepth=10),
    "NaiveBayes_1": NaiveBayes(smoothing=1.0),
    "NaiveBayes_2": NaiveBayes(smoothing=0.5)
}
```

Los resultados obtenidos han los siguientes:

```
LogisticRegression_1: AUC = 0.8785
LogisticRegression_2: AUC = 0.8372
RandomForest_1: AUC = 0.9129
RandomForest_2: AUC = 0.9686
NaiveBayes_1: AUC = 0.6109
NaiveBayes_2: AUC = 0.6109
```

He probado también ejecutando

```
"DecisionTree_1": DecisionTreeClassifier(maxDepth=20),
"DecisionTree_2": DecisionTreeClassifier(maxDepth=10)
```

por separado, para comprobar que me resultados daba este clasificador y el resultado ha sido:

```
DecisionTree_1: AUC = 0.9385
DecisionTree_2: AUC = 0.8684
```

Con estos resultados, podemos decir que:

1. Los modelos de Random Forest son los que tienen mejor rendimiento. El segundo modelo (RandomForest_2), con más árboles (30) y mayor profundidad (10), ha alcanzado un AUC de 0.9686, lo que indica una capacidad muy alta para distinguir entre galaxias y estrellas. El incremento en el número de árboles y la profundidad ha mejorado claramente la capacidad de generalización del modelo.

2. La regresión logística también ha tenido un buen rendimiento, especialmente en su configuración más simple (LogisticRegression_1). Sin embargo, aumentar la regularización en LogisticRegression_2 ha

disminuido el rendimiento, lo que sugiere que un exceso de penalización ha limitado la capacidad del modelo para ajustar los datos.

3. Los modelos de Naive Bayes son los que han tenido los peores resultados, con un AUC de apenas 0.6109 en ambas configuraciones. A pesar de usar MinMaxScaler para asegurarme de que las variables tuvieran valores no negativos, la suposición de independencia entre variables que realiza este clasificador no se ajusta adecuadamente a la complejidad del conjunto de datos.

Así que, como resumen puedo afirmar que el modelo RandomForest_2 es el más eficaz en este caso, seguido por LogisticRegression_1. Naive Bayes, por su parte, no es el más adecuado para este problema en particular.

6. Problemas encontrados

Problema 1:

En el clasificador estaba obteniendo error por no separar bien las columanas. Para solucionarlo añado en la línea 15:

```
sep=';'
```

Problema 2:

No encontraba los errores porque me aparecía demasiada información en la terminal.
Para solucionarlo añado la siguiente línea de código a `clasificador.py` spark.

```
sparkContext.setLogLevel("ERROR")
```

Problema 3:

Al ejecutar el clasificador acababa en "Killed" esto era debido a que se estaba quedando sin memoria justo durante el entrenamiento de uno de los modelos.
Para solucionarlo ejecuto el clasificador con este comando:

```
spark-submit --master local[2] --executor-memory 4g
/workspace/clasificador.py
```

- `local[2]`: usa solo 2 núcleos (evita sobrecargar).
- `--executor-memory 4g`: da más RAM al proceso.

Además, también me he dado cuenta de que `RandomForest_2` (50 árboles, maxDepth=10) avanzaba bastante y casi terminaba, pero acababa fallando con el mensaje Killed, los mensajes indicaban un problema de recursos. Para solucionaro disminuyo el número de árboles a 30

Problema 4:

"requirement failed: Naive Bayes requires nonnegative feature values"

Este problema está relacionado con que el modelo Naive Bayes, exige que todas las features sean >= 0
En mi caso, esto aparecía porque estaba usando `StandardScaler` para todos los modelos.
Para solucionarlo, hago que NaiveBayes use `MinMaxScaler` en lugar de `StandardScaler`, mientras el resto de modelos (LogisticRegression y RandomForest) siguen usando `StandardScaler`