Nombre: José Antonio Laserna Belrán
Email: jalb0002@correo.ugr.es
DNI: 26510180N

Práctica 3: Problema de clasificación con Hadoop, Spark

Descripción de la práctica

Se pide resolver un problema de clasificación mediante su implementación con las utilidades que ofrece Spark. Como lenguaje de programación he elegido Python por mi experiencia con él y porque es un lenguaje habitual para tratar este tipo de problemas.

En sintonía con las sesiones de prácticas 9, 10 y 11, se ha desplegado Spark en una serie de contenedores mediante Docker y Docker compose.

El dataset asignado para esta tarea es una colección de datos del Apache Point Observatory en Nuevo México, EE. UU. Este dataset contiene imágenes y datos espectroscópicos de millones de objetos celestes.

Despliegue de los contenedores

Se ha partido de la estructura utilizada en las prácticas 9, 10 y 11. Sin embargo, en mi caso al resolver la práctica en mi máquina local, he tenido que realizar algunas modificaciones.

En primer lugar, he tenido que especificar valores de **ulimit** suficientes para poder lanzar los contenedores namenode y datanode:

```
16      ulimits:
17        nofile:
18          soft: 65536
19          hard: 65536
```

Por otro lado, he tenido que lanzar contenedores de Spark específicamente como *workers* para poder ejecutar los scripts de Python ya que, si no se lanzan, Spark queda a la espera de encontrar estos workers. En mi caso he lanzado 2 workers y los he definido en el docker-compose.yaml de la siguiente forma:

```
Run Service
spark-worker-1:
  build: ./spark
  container_name: spark-worker-1
  environment:
    - SPARK_MODE=worker
    - SPARK_MASTER_URL=spark://spark:7077
    - CORE_CONF_fs_defaultFS=hdfs://namenode:8020
  depends_on:
    - spark
  networks:
    - hadoop

Run Service
spark-worker-2:
  build: ./spark
  container_name: spark-worker-2
  environment:
    - SPARK_MODE=worker
    - SPARK_MASTER_URL=spark://spark:7077
    - CORE_CONF_fs_defaultFS=hdfs://namenode:8020
  depends_on:
    - spark
  networks:
    - hadoop
```

Otra cuestión que me ha dado problemas es como trasladar los archivos necesarios a los contenedores, pues no he conseguido que el contenido del directorio que se monta como volumen para los contenedores de Hadoop aparezca en el sistema de archivos del contenedor al lanzarlo. Como solución, he creado un script de bash muy simple para automatizar este proceso y, adicionalmente, automatizar el proceso de despliegue:

```
#!/bin/bash

systemctl start docker
docker compose up -d
sleep 1

#Namenode
docker container ls -a
docker cp dataset/small_celestial.csv namenode:/data/small_celestial.csv
docker exec namenode hdfs dfs -mkdir /user
docker exec namenode hdfs dfs -mkdir /user/spark
docker exec namenode hdfs dfs -chown spark:spark /user/spark
docker exec namenode hdfs dfs -ls /user/spark
docker exec namenode hdfs dfs -put /data/small_celestial.csv /user/spark/

#Spark
docker cp pyspark_clasificacion_v2.py spark:pyspark_clasificacion_v2.py
```

Hay que hacer un proceso similar un proceso similar para el contenedor de spark a la hora de proporcionarle el script de python y recuperar los resultados:

```
docker cp pyspark_clasificacion_v2.py spark:pyspark_clasificacion_v2.py
docker exec spark spark-submit --master spark://spark:7077 --total-executor-cores 6 --executor-memory 1g /pyspark_clasificacion_v2.py
docker cp spark:/opt/bitnami/spark/resultados_prac3_cc.txt resultados_prac3_cc.txt
chown lassy:lassy resultados_prac3_cc.txt
```

Clasificación

Para resolver el problema se ha utilizado la biblioteca Mllib de Spark con Python que viene referenciada en el guion de la práctica.

En primer lugar se crea una sesión de Spark y se cargan los datos desde hdfs:

```
spark = SparkSession.builder.appName("Prac3_CC").getOrCreate()

df = spark.read.csv(
    "hdfs://namenode:8020/user/spark/small_celestial.csv",
    sep=";", header=True, inferSchema=True
).dropna()
```

Habría que tratar el balanceo de los datos utilizados pero en este dataset los datos ya están balanceados:

```
+------+-----+
|  type|count|
+------+-----+
|galaxy|50000|
|  star|50000|
+------+-----+
```

El aspecto de los datos es el siguiente:

```
+---------+--------+----------+-----------+----------+----------+-----------+--------+-----------------+--------+------+
|  expAB_z|       i|       q_r|modelFlux_r|   expAB_i|  expRad_u|        q_g|psfMag_z|              dec|psfMag_r|  type|
+---------+--------+----------+-----------+----------+----------+-----------+--------+-----------------+--------+------+
|0.4802158| 21.9757|-0.1255715|   1.421841| 0.1488244|0.09189734| -0.2031111|22.95979|0.0202495652658788|22.56184|galaxy|
|     0.05|21.61484| -0.120134|    1.59757|      0.05|   4.21148|-0.08888569|21.71544| 0.727337177932732|22.31075|galaxy|
|0.1927271|21.88879|-0.1287684|   1.318119|0.09904385|  2.314147|-0.02433589|22.06033|-0.460815701136998|22.27711|galaxy|
|0.5904964|22.07951|-0.2159682|   1.424122|      0.05|0.01768703|  0.3914998|21.82783|0.0918372890303701|22.33795|galaxy|
|0.3282258| 21.0175|-0.1552096|   2.560175|      0.05| 0.3770643| -0.5457523|21.04544| 0.892359363199922|22.06569|galaxy|
+---------+--------+----------+-----------+----------+----------+-----------+--------+-----------------+--------+------+
only showing top 5 rows
```

Y el objetivo de la clasificación es el tipo de cuerpo celestial. Para ello hay que preparar los datos:

- Se convierte la columna **type** de tipo texto a índices.
- Se filtra la columna **type**, separándola de los datos de entrenamiento.
- Generar una columna vectorial para entrenar los modelos.

```
51
52      label_indexer = StringIndexer(inputCol="type", outputCol="indexedLabel")
53      feature_cols = [c for c in df.columns if c != "type"]
54      assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
```

A continuación se crea la partición de test y entrenamiento

```
56      train, test = df.randomSplit([0.8, 0.2], seed=42)
57      evaluator = MulticlassClassificationEvaluator(
58          labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy"
59      )
```

Para realizar el entrenamiento y la predicción se crea el un **pipeline** con los datos filtrados para realizar el entrenamiento junto a un **grid** para poder probar distintos parámetros del modelo automáticamente. Además, se ha implementado **cross validation** para mejorar el entrenamiento:

```python
def entrenar_y_evaluar(nombre, classifier_stage, param_grid, usar_minmax=False, num_folds=5):

    stages = [label_indexer, assembler]
    if usar_minmax:
        scaler = MinMaxScaler(inputCol="features", outputCol="scaledFeatures")
        stages.append(scaler)
    stages.append(classifier_stage)
    pipeline = Pipeline(stages=stages)

    builder = ParamGridBuilder()
    for param_name, valores in param_grid.items():
        param_obj = classifier_stage.getParam(param_name)
        builder = builder.addGrid(param_obj, valores)
    grid = builder.build()

    cv = CrossValidator(
        estimator=pipeline,
        estimatorParamMaps=grid,
        evaluator=evaluator,
        numFolds=num_folds
    )

    start_train = time.time()
    cvModel = cv.fit(train)
    end_train = time.time()
    train_elapsed = end_train - start_train
    train_times.append((nombre, train_elapsed))

    estimatorParamMaps = cv.getEstimatorParamMaps()
    avgMetrics = cvModel.avgMetrics
    for param_map, metric in zip(estimatorParamMaps, avgMetrics):
        params = {param.name: param_map[param] for param in param_map}
        resultados.append((nombre, params, metric))

    best_models.append((nombre, cvModel.bestModel))
```

Por último se llama a la función para cada modelo a evaluar. En mi caso he utilizado DecisionTree, RandomForest y LogisticRegresion:

```python
dt = DecisionTreeClassifier(labelCol="indexedLabel", featuresCol="features")
entrenar_y_evaluar(
    "Decision Tree",
    dt,
    {"maxDepth": [3, 5, 7, 10]},
    usar_minmax=False,
    num_folds=5
)

rf = RandomForestClassifier(labelCol="indexedLabel", featuresCol="features")
entrenar_y_evaluar(
    "Random Forest",
    rf,
    {"numTrees": [3, 5, 7, 10]},
    usar_minmax=False,
    num_folds=5
)

lr = LogisticRegression(labelCol="indexedLabel", featuresCol="features")
entrenar_y_evaluar(
    "Logistic Regression",
    lr,
    {"regParam": [0.01, 0.05, 0.1, 0.3]},
    usar_minmax=False,
    num_folds=5
)
```

Resultados

Los resultados obtenidos para el conjunto de entrenamiento más pequeño han sido los siguientes:

Modelo	F1 Score	Precisión	Recall	Accuracy	Entrenamiento	Predicción
Decision Tree	89.76%	89.91%	89.77%	89.77%	38.30s	0.0262s
Random Forest	81.50%	82.66%	81.64%	81.64%	27.66s	0.0327s
Logistic Regression	80.00%	80.27%	80.03%	80.03%	21.35s	0.0280s

Como se puede comprobar, Logistic Regression y Random Forest tienen un desempeño similar, siendo Logistic Regression más rápido.

El mejor modelo ha sido Decision Tree pero también es bastante más lento.

En conclusión, si no hay una restricción de tiempo para el entrenamiento, Decision Tree mejora en rendimiento a los otros dos. Si el conjunto de datos tuviese un tamaño tal que el tiempo fuese un problema, probablemente elegiría Logistic Regressión pues es bastante más rápido que Random Forest con unos resultados similares.