



# UNIVERSIDAD DE GRANADA

## PRÁCTICA 3:

Procesamiento y minería de datos con Spark

---

Cloud Computing: Servicios y Aplicaciones

### Autor

Pablo Valenzuela Álvarez ([pvalenzuela@correo.ugr.es](mailto:pvalenzuela@correo.ugr.es))



# ÍNDICE

<b>Introducción.....</b>	<b>3</b>
<b>Resolución del problema.....</b>	<b>4</b>
Preprocesado.....	7
Resultados.....	10
Conclusiones.....	16

# Introducción

En esta práctica, nos adentraremos en el uso de Apache Spark, una potente herramienta de procesamiento de datos en tiempo real que nos permitirá manejar y analizar grandes volúmenes de información de manera eficiente. Nuestro primer proyecto consistirá en trabajar con el dataset "Celestial Objects" disponible en [Kaggle](#), el cual contiene información detallada sobre diversos objetos celestes, como estrellas, planetas y galaxias.

A través de esta práctica, aprenderemos a cargar, procesar y analizar datos utilizando Spark, aprovechando sus capacidades de paralelización y manejo de grandes conjuntos de datos.

# Resolución del problema

Para resolver el problema vamos a simular un entorno spark, ya que no disponemos de acceso a ninguno. Necesitamos crear varias imágenes para simular un proceso **master** con varios **workers** para entrenar uno o varios modelos de clasificación, y también, una **API** para mostrar el resultado de la predicción. Vamos a crear varios archivos Dockerfile para las imágenes anteriores (ver figura 1).

```
You, 3 days ago | 2 authors (Pablo Valenzuela and others)
1 # builder para descargar y configurar el entorno spark
2 FROM openjdk:11.0.11-jre-slim-buster as builder
3
4 # dependencias para pyspark
5 RUN apt-get update && apt-get install -y curl vim wget software-properties-common
6
7 RUN update-alternatives --install "/usr/bin/python" "python" "$(which python3)" 1
8
9 ENV SPARK_VERSION=3.5.1 \
10 HADOOP_VERSION=3 \
11 SPARK_HOME=/opt/spark \
12 PYTHONHASHSEED=1
13
14 RUN apt-get update && \
15     apt-get install -y python3-pip && \
16     pip3 install pyspark
17
18 # descarga de spark desde el archivo de apache
19 RUN wget --no-verbose -O apache-spark.tgz "https://archive.apache.org/dist/spark/spark-$SPARK_VERSION/apache-spark-$SPARK_VERSION-bin-hadoop$HADOOP_VERSION.tgz"
20 && mkdir -p /opt/spark \
21 && tar -xf apache-spark.tgz -C /opt/spark --strip-components=1 \
22 && rm apache-spark.tgz
23
24
25 # entorno de apache spark
26 FROM builder as apache-spark
27
28 WORKDIR /opt/spark
29
30 ENV SPARK_MASTER_PORT=7077 \
31 SPARK_MASTER_WEBUI_PORT=8080 \
32 SPARK_LOG_DIR=/opt/spark/logs \
33 SPARK_MASTER_LOG=/opt/spark/logs/spark-master.out \
34 SPARK_WORKER_LOG=/opt/spark/logs/spark-worker.out \
35 SPARK_WORKER_WEBUI_PORT=8080 \
36 SPARK_WORKER_PORT=7000 \
37 SPARK_MASTER="spark://spark-master:7077" \
38 SPARK_WORKLOAD="master"
39
40 RUN mkdir -p $SPARK_LOG_DIR && \
41 touch $SPARK_MASTER_LOG && \
42 touch $SPARK_WORKER_LOG && \
43 ln -sf /dev/stdout $SPARK_MASTER_LOG && \
44 ln -sf /dev/stdout $SPARK_WORKER_LOG
45
46 COPY start-spark.sh /
47
48 RUN chmod +x /start-spark.sh
49
50 CMD ["/bin/bash", "/start-spark.sh"]
```

Figura 1. Ejemplo de Dockerfile master.

**Nota:** Como estoy trabajando en Windows necesito ejecutar el siguiente comando: “`sed -i 's/\r$//' start-spark.sh`”. Básicamente elimina los saltos de línea y permite la ejecución en las imágenes del archivo *start-spark.sh*. El fichero *start-spark.sh* crea el proceso de spark y clasifica en master y worker según el servicio que lo llame.

Una vez creadas las imágenes, ejecutamos el fichero *compose.yaml* para crear los servicios (ver figura 2).

```

1  ...
2  services:
3    spark-master:
4      image: spark-master:3.5.1
5      container_name: spark-master
6      ports:
7        - "9090:8080"
8        - "7077:7077"
9      volumes:
10       - ./apps:/opt/spark-apps
11       - ./data:/opt/spark-data
12       - ./output:/opt/spark-output
13     environment:
14       - SPARK_LOCAL_IP=spark-master
15       - SPARK_WORKLOAD=master
16
17   spark-worker-1:
18     image: spark-worker:3.5.1
19     container_name: spark-worker-1
20     ports:
21       - "9091:8080"
22       - "7000:7000"
23     depends_on:
24       - spark-master
25     environment:
26       - SPARK_MASTER=spark://spark-master:7077
27       - SPARK_WORKER_CORES=2
28       - SPARK_WORKER_MEMORY=1G
29       - SPARK_DRIVER_MEMORY=1G
30       - SPARK_EXECUTOR_MEMORY=1G
31       - SPARK_WORKLOAD=worker
32       - SPARK_LOCAL_IP=spark-worker-1
33
34   spark-worker-2:
35     image: spark-worker:3.5.1
36     container_name: spark-worker-2
37     ports:
38       - "9092:8080"
39       - "7001:7000"
40     depends_on:
41       - spark-master
42     environment:
43       - SPARK_MASTER=spark://spark-master:7077
44       - SPARK_WORKER_CORES=2
45       - SPARK_WORKER_MEMORY=1G
46       - SPARK_DRIVER_MEMORY=1G
47       - SPARK_EXECUTOR_MEMORY=1G
48       - SPARK_WORKLOAD=worker
49       - SPARK_LOCAL_IP=spark-worker-2
50
51   spark-api:
52     image: spark-api:3.5.1
53     container_name: spark-api
54     ports:
55       - "5000:5000"
56     depends_on:
57       - spark-master
58     environment:
59       - SPARK_MASTER=spark://spark-master:7077
60     volumes:
61       - ./api:/opt/spark-api
62       - ./output:/opt/spark-output
63     command: ["python", "/opt/spark-api/celestial_api.py"]

```

Figura 2. Fichero compose.yaml.

Si nos fijamos en los servicios, se crea un servicio master, dos workers y una API. El servicio master ejecutará los ficheros y se encargará de distribuirlo entre los workers y recoger los resultados. Por último, en la API podemos realizar peticiones de predicción.

El fichero que ejecutaremos en el servicio master contendrá todo lo necesario para cargar el dataset, preprocesar los datos y crear varios modelos de clasificadores. En la figura 3, se muestran los algoritmos de clasificación (DecisionTree, RandomForest y GradientBoosting) y los hiperparámetros que se van a usar para cada uno.

```

# Definimos un Árbol de Decisión
dt = DecisionTreeClassifier(labelCol="indexed_label", featuresCol="scaled_features")

# Con ayuda de ParamGridBuilder construimos una malla (grid) con los hiperparámetros
# del modelo.
dt_param_grid = ParamGridBuilder()\
    .addGrid(dt.maxDepth, [3, 10])\
    .addGrid(dt.maxBins, [10, 50])\
    .addGrid(dt.minInstancesPerNode, [5, 10])\
    .addGrid(dt.impurity, ["gini", "entropy"])\
    .build()

# Aquí definimos otro modelo, en esta ocasión, un Random Forest.
rf = RandomForestClassifier(labelCol="indexed_label", featuresCol="scaled_features")

# Y esta es la malla correspondiente al Random Forest
rf_param_grid = ParamGridBuilder()\
    .addGrid(rf.maxDepth, [3, 10])\
    .addGrid(rf.maxBins, [10, 30])\
    .addGrid(rf.minInstancesPerNode, [5, 10])\
    .addGrid(rf.impurity, ["gini", "entropy"])\
    .addGrid(rf.featureSubsetStrategy, ["auto", "sqrt", "log2"])\
    .build()

# Aquí definimos el último modelo, un Gradient Boosting.
gb = GBClassifier(labelCol="indexed_label", featuresCol="scaled_features")

# Y su malla correspondiente
gb_param_grid = ParamGridBuilder()\
    .addGrid(gb.maxDepth, [3, 10])\
    .addGrid(gb.minInstancesPerNode, [5, 10])\
    .addGrid(gb.subsamplingRate, [0.6, 0.8])\
    .build()

```

Figura 3. Algoritmos de entrenamiento.

## Preprocesado

Empezamos haciendo una exploración de los datos. Tenemos que acceder al servicio master (*docker exec -it spark-master /bin/bash*) y ejecutar la consola **pyspark**.

Si ejecutamos los siguientes comandos:

- `df = spark.read.csv("/opt/spark-data/small_celestial.csv",header=True,sep=";",inferSchema=True);`
- `df.describe();`

Podemos ver los campos del dataset y analizarlos, el resultado de la salida es el siguiente:

DataFrame[summary: string, expAB\_z: string, i: string, q\_r: string, modelFlux\_r: string, expAB\_i: string, expRad\_u: string, q\_g: string, psfMag\_z: string, dec: string, psfMag\_r: string, type: string]

Si ejecutamos los siguientes, vemos una representación de los datos que contienen todos los campos (ver figura 4):

- `df.createOrReplaceTempView("dataset");`
- `sql = spark.sql("select * from dataset");`

```
>>> sql.show()
```

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
0.05	23.20646	-0.241973	0.8345672	0.05	12.91812	0.3144629	22.59644	0.229343090133464	22.87204	galaxy
0.5874656	19.59161	0.02346599	8.683687	0.9609066	2.737795	0.1174305	19.92542	-1.00231304975531	20.82201	galaxy
0.05	21.86053	0.3565305	0.9458704	0.05	51.41937	0.4425624	22.62458	0.028629512064739	22.83852	galaxy
0.05	20.24973	-0.08548556	6.038634	0.1993707	24.64536	0.2871795	21.96052	14.6264464963057	21.96932	galaxy
0.05	22.38403	0.02455264	1.18002	0.05	0.5878582	0.1845807	22.92211	-0.64869835405967	22.27636	galaxy
0.05	22.05696	0.02236104	6.401645	0.05	15.88138	-0.2687832	19.61789	65.7589609307604	20.47844	galaxy
0.05	21.59862	0.008893777	1.812271	0.1204406	2.736085	-0.01891627	21.40633	-0.459534425907587	22.42269	galaxy
0.05	20.98391	0.03827773	3.184495	0.8001134	0.5445185	-0.0628657	20.87324	14.8905130066439	21.61997	galaxy
0.169554	20.47739	-0.1468553	5.038363	0.2533349	0.7323152	-0.1498862	20.80894	1.21168123198785	21.34776	galaxy
0.05	21.25115	-0.01915844	2.36242	0.4884966	0.1673803	-0.09078356	21.50078	0.295758780451645	21.7166	galaxy
0.05	21.79118	-0.1678272	1.337815	0.2391625	6.594543	0.37391	22.134	12.0412931353591	22.49362	galaxy
0.173639	18.07745	0.09755554	33.28561	0.6066637	0.08446308	0.03473249	18.1015	51.3224916971324	18.71255	galaxy
0.7993075	18.03049	0.04409927	40.23423	0.8102384	1.503553	-0.03456844	18.49508	0.0509400807800082	19.23211	galaxy
0.1499711	21.57549	0.03147343	1.4845	0.4154454	1.211273	0.003889297	21.19818	-0.21807204097463	22.04378	galaxy
0.09997772	21.37227	0.06344061	1.385579	0.2290707	7.405845E-4	0.01286124	21.61116	-0.802291729222016	22.77431	galaxy

only showing top 20 rows

Figura 4. Muestra de los datos del dataset

Observamos que todos los datos son numéricos y que se mueven en rangos distintos, por lo que habrá que normalizarlos entre 0 y 1. La siguiente figura muestra una descripción de los datos, aquí podemos ver los rangos entre los que se mueven.

```
>>> df.summary()
DataFrame[summary: string, expAB_z: string, i: string, q_r: string, modelFlux_r: string, expAB_i: string, expRad_u: string, q_g: string, psfMag_z: string, dec: string, psfMag_r: string, type: string]
>>> df.summary().show()
24/05/22 10:39:20 WARN SparkStringUtils: Truncated the string representation of a plan since it was too large. This behavior can be adjusted by setting 'spark.sql.debug.maxToStringFields'.
```

summary	expAB_z	i	q_r	modelFlux_r	expAB_i	expRad_u	q_g	psfMag_z	dec	psfMag_r	type
count	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000	100000
mean	0.2833732901238045	20.26422049840006	-0.03464701281830368	37.44998410227312	0.40623806708417726	4.256249439594138	-0.17534578134092843	20.142073942499994	1.284481017192383	21.040297936600048	NULL
stddev	0.28083588765448875	1.7340563728259963	0.8380916507624272	192.04811896385797	0.2997675455524942	9.906412587013232	57.32981696269636	1.7837927426240056	7.124205369648828	1.7737590374641898	NULL
min	0.05	11.81408	-70.44912	-0.7927552	0.05	0.0	-17255.0	10.47052	-9.88502209738354	12.17984	galaxy
25%	0.05	19.39212	-0.02674623	1.584243	0.126449	0.03759531	-0.1079832	19.21378	-0.517865193627807	20.26007	NULL
50%	0.1535061	20.63059	-0.0354932	3.127704	0.3500139	0.6655853	-0.01787298	20.48006	0.102490977890749	21.63221	NULL
75%	0.424195	21.42811	0.01961469	10.25441	0.6403865	3.345292	0.06711205	21.40648	0.743352548383233	22.34121	NULL
max	1.0	33.21816	158.5264	13510.94	0.9999999	59.40614	4847.074	27.84727	74.9551450207025	26.40547	star

Figura 5. Descripción de los datos

Si queremos ver si el dataset contiene valores perdidos, tenemos que ejecutar el comando de la figura 6. Como se ve en la figura, no contiene ninguno, por lo que no tenemos que preocuparnos por este problema.

```
>>> from pyspark.sql.functions import isnan, when, count, col
>>>
>>> df.select([count(when(isnan(c), c)).alias(c) for c in df.columns]).show()
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|expAB_z| i | q_r | modelFlux_r | expAB_i | expRad_u | q_g | psfMag_z | dec | psfMag_r | type |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
```

Figura 6. mostrar los valores perdidos.

Como último paso de la exploración, vamos a ver las clases con las que cuenta el dataset. En la figura 7 se muestra que tiene dos: galaxy y star.

```
>>> df = spark.read.csv("/opt/spark-data/small_celestial.csv", sep=";", header=True, inferSchema=True)
>>> df.select("type").distinct().show()
+-----+
| type |
+-----+
| galaxy |
| star |
+-----+
```

Figura 7. Clases del dataset.

Terminada la exploración inicial, procedemos a aplicar el preprocesado a los datos. En la figura 8 se pueden observar los pasos que se han seguido:

1. Identificar los atributos que serán las **características**.
2. Aplicar la **normalización**.
3. Identificar la variable **objetivo**.
4. Y crear el **pipeline** con todas las transformaciones.



```

# Detectamos los atributos que son numéricos para transformarlos (normalizarlos)
numeric_features = [col for col, dtype in data.dtypes if dtype == "int" or dtype == "double"]

# Etiquetamos a los atributos numéricos como "features"
assembler = VectorAssembler(inputCols=numeric_features, outputCol="features")

# Los normalizamos y los etiquetamos como "scaled_features"
scaler = StandardScaler(inputCol="features", outputCol="scaled_features")

# Indexamos al atributo clase y lo etiquetamos como "indexed_label"
labelIndexer = StringIndexer(inputCol="type", outputCol="indexed_label")

# Creamos un pipeline para realizar todas estas transformaciones en orden
pipeline = Pipeline(stages=[labelIndexer, assembler, scaler])

# Compilamos el pipeline con los datos
prep_model = pipeline.fit(data)

prep_model.save(output_folder + "/pipeline")

# Transformamos los datos con el pipeline compilado y
# los guardamos en una nueva variable "prep_data"
prep_data = prep_model.transform(data)

```

Figura 8. Preprocesado.

## Resultados

Para obtener los resultados, antes tenemos que generar los modelos. Para ello, debemos acceder al servicio master y ejecutar nuestro fichero con la aplicación que usará procesamiento spark.

Accedemos al servicio master con:

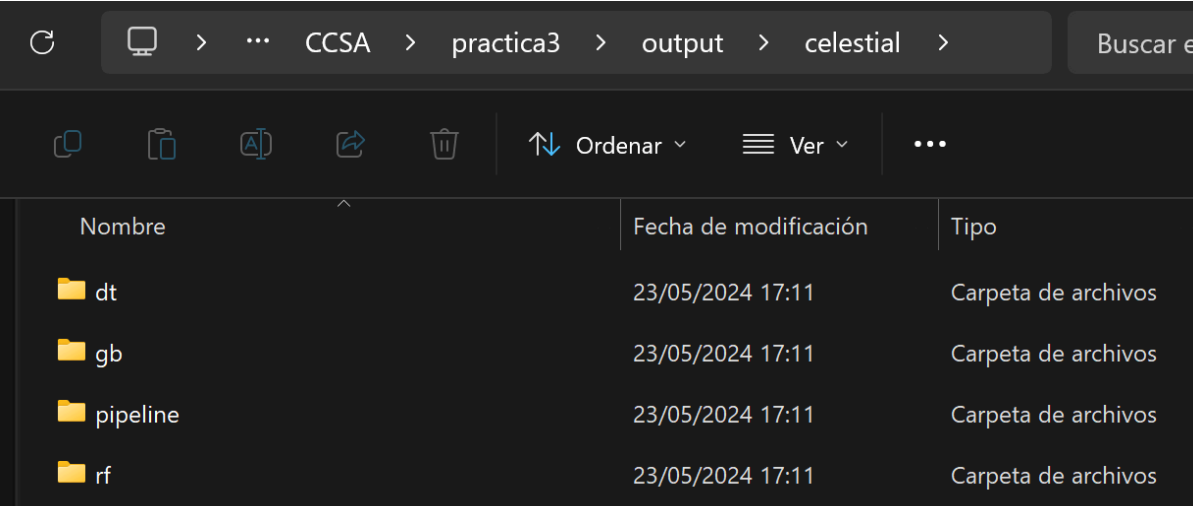
- `docker exec -it spark-master /bin/bash`

Y ejecutamos:

- `/opt/spark/bin/spark-submit /opt/spark-apps/celestial.py --input /opt/spark-data/small_celestial.csv --output /opt/spark-output/celestial`

De esta manera se ejecutará nuestro fichero `celestial.py` sobre el dataset `small_celestial.csv`, y generará una salida recogida en `/opt/spark-output/celestial`.

Como hemos definido tres algoritmos en el fichero `celestial.py`, obtendremos unos resultados parecidos a los de la siguiente figura. En los cuales tendremos los tres mejores modelos creados usando los hiperparámetros vistos en la figura 3 y el pipeline.



Nombre	Fecha de modificación	Tipo
dt	23/05/2024 17:11	Carpeta de archivos
gb	23/05/2024 17:11	Carpeta de archivos
pipeline	23/05/2024 17:11	Carpeta de archivos
rf	23/05/2024 17:11	Carpeta de archivos

Figura 9. Resultados de entrenamiento.

Una vez termine el procesamiento de spark, nos dirigiremos a la API. En el código de esta cargamos los modelos como se muestra en la figura 10. La API espera una petición sobre la dirección `/predict` con un archivo json por lo que es necesario procesarlo como se muestra en la figura 11. Y por último, obtener los resultados de la predicción (ver figura 12).

```

# Crear una sesión de Spark
spark = SparkSession.builder.appName('MLModelPrediction').getOrCreate()

# Ruta al modelo MLlib
MODEL_DT_PATH = "/opt/spark-output/celestial/dt"
MODEL_RF_PATH = "/opt/spark-output/celestial/rf"
MODEL_GB_PATH = "/opt/spark-output/celestial/gb"
PIPELINE_PATH = "/opt/spark-output/celestial/pipeline"

# Crear la aplicación Flask
app = Flask(__name__)

# Función para cargar los modelos y el pipeline
def load_models_and_pipeline():
    global pre_pipeline, model_dt, model_rf, model_gb
    while True:
        if os.path.exists(MODEL_DT_PATH) and os.path.exists(MODEL_RF_PATH) and os.path.exists(MODEL_GB_PATH) and os.path.exists(PIPELINE_PATH):
            pre_pipeline = PipelineModel.load(PIPELINE_PATH)
            model_dt = DecisionTreeClassificationModel.load(MODEL_DT_PATH)
            model_rf = RandomForestClassificationModel.load(MODEL_RF_PATH)
            model_gb = GBTClassificationModel.load(MODEL_GB_PATH)
            break
        else:
            print("Models or pipeline not found. Waiting for them to become available...")
            time.sleep(30) # Espera 30 segundos antes de volver a verificar

```

Figura 10. Carga de modelos en la API.

```

@app.route('/predict', methods=['POST'])
def predict():
    try:
        # Obtener los datos del request
        data = request.get_json()

        # Definir el esquema de los datos esperados
        schema = StructType([
            StructField("expAB_z", FloatType(), True),
            StructField("i", FloatType(), True),
            StructField("q_r", FloatType(), True),
            StructField("modelFlux_r", FloatType(), True),
            StructField("expAB_i", FloatType(), True),
            StructField("expRad_u", FloatType(), True),
            StructField("q_g", FloatType(), True),
            StructField("psfMag_z", FloatType(), True),
            StructField("dec", FloatType(), True),
            StructField("psfMag_r", FloatType(), True)
        ])
    
```

Figura 11. Petición y procesamiento de entradas.

```

# Crear un DataFrame de Spark con los datos recibidos
input_data = spark.createDataFrame([data], schema)

prep_data = pre_pipeline.transform(input_data)

# Hacer la predicción
predictions_dt = model_dt.transform(prep_data)
predictions_rf = model_rf.transform(prep_data)
predictions_gb = model_gb.transform(prep_data)

# Seleccionar las columnas de predicción y probabilidad para cada modelo
prediction_dt = predictions_dt.select("prediction").collect()[0][0]
probabilities_dt = predictions_dt.select("probability").collect()[0][0]

prediction_rf = predictions_rf.select("prediction").collect()[0][0]
probabilities_rf = predictions_rf.select("probability").collect()[0][0]

prediction_gb = predictions_gb.select("prediction").collect()[0][0]
probabilities_gb = predictions_gb.select("probability").collect()[0][0]

```

Figura 12. Recogida de resultados de la predicción.

Una vez activa la API, podemos realizar las peticiones para ver cómo funciona. A través de la aplicación **Postman** hemos realizado dicha tarea obteniendo los resultados de las siguientes figuras (figuras 13, 14, 15 y 16).

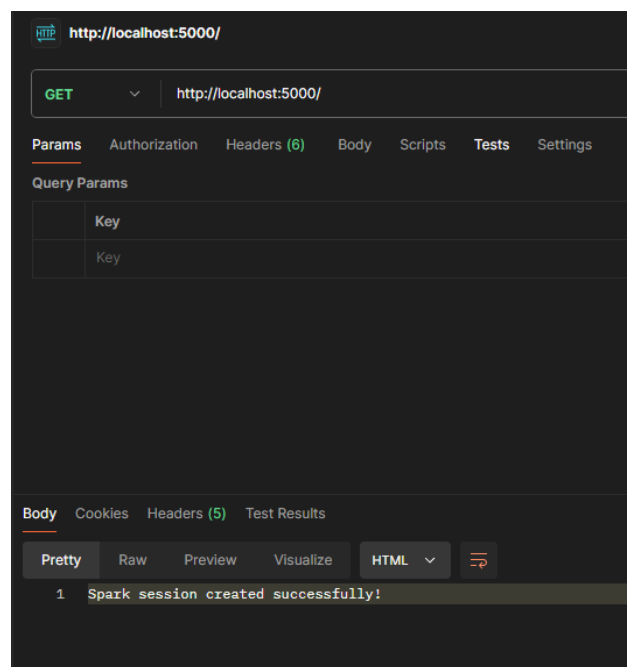


Figura 13. Petición GET a la API.

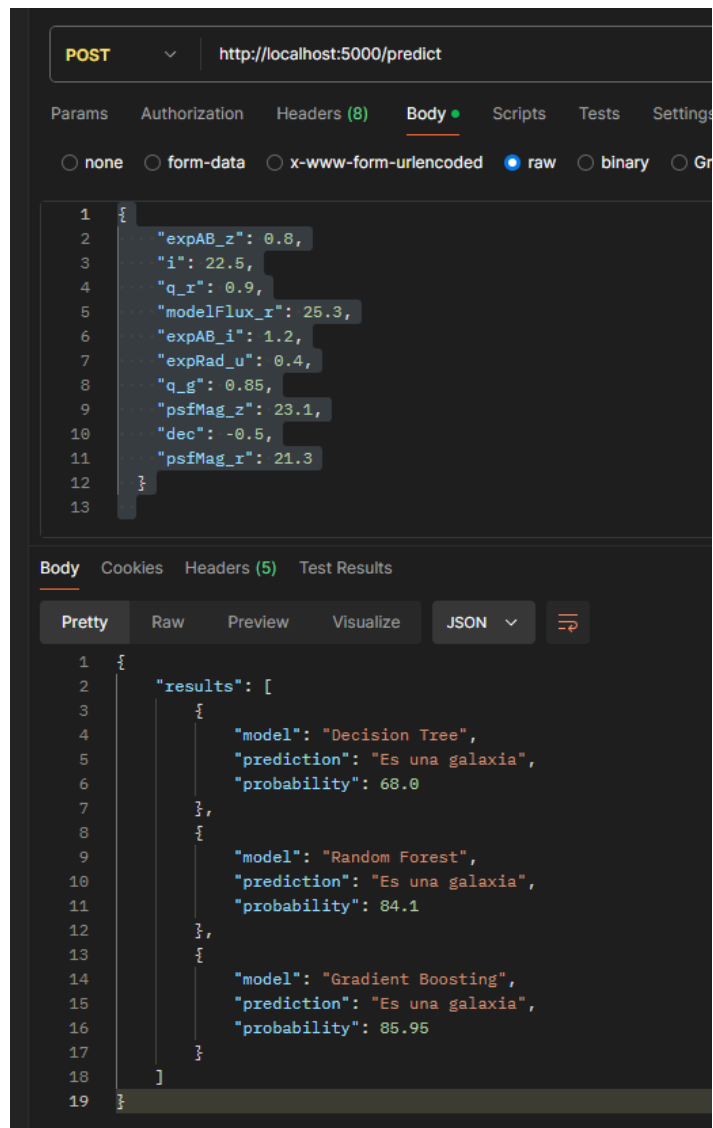


Figura 14. Petición /predict con el primer json.

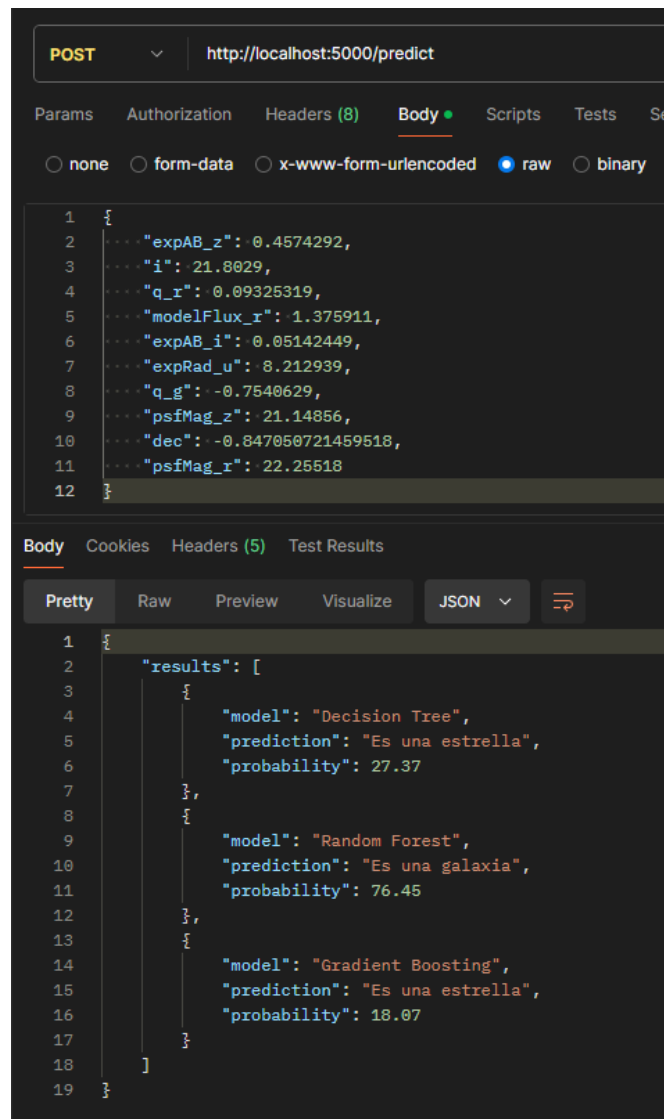


Figura 15. Petición `/predict` con el segundo json.

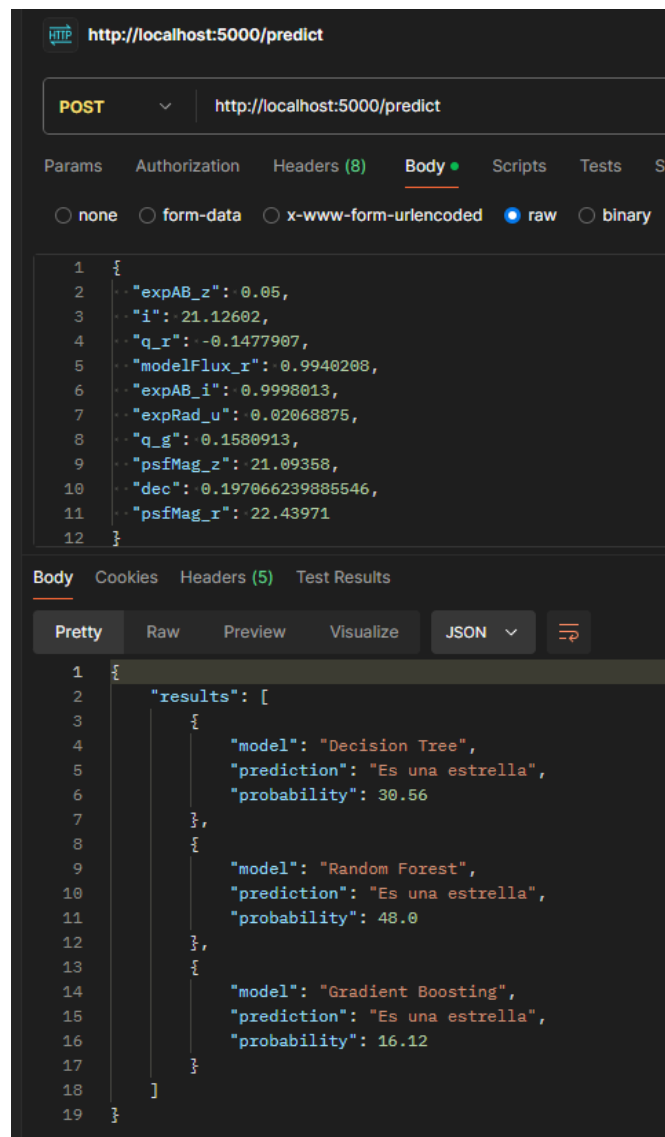


Figura 16. Petición /predict con el tercer json.

Como podemos observar en las figuras 14, 15 y 16, obtenemos resultados diferentes según el algoritmo que usemos e incluso totalmente diferentes como en el caso de la figura 15. Si queremos escoger uno sobre todos, podemos guiarnos por el parámetro AUC (área bajo la curva) del json *params.json* ubicado en las carpetas de los modelos:

- DT = 0.8826
- RF = 0.9654
- GB = 0.9801

En teoría, el modelo proporcionado por el algoritmo **Gradient Boosting** es mejor. Sin embargo en el caso de la figura 16 es el que peor resultado obtiene pero en la figura 14 el mejor.

## Conclusiones

En esta práctica, hemos demostrado la potencia y versatilidad de Apache Spark para el procesamiento de grandes volúmenes de datos y la creación de modelos de machine learning en un entorno simulado.

Los resultados obtenidos de los tres algoritmos de clasificación (Decision Tree, Random Forest y Gradient Boosting) han sido variados, con el Gradient Boosting mostrando el mejor rendimiento en términos de área bajo la curva. Sin embargo, también hemos observado que el rendimiento puede fluctuar dependiendo de los datos de entrada específicos, lo cual es una lección crucial para cualquier proyecto de análisis de datos: no hay un modelo perfecto para todas las situaciones, y la validación constante es esencial.

La implementación de la API para realizar predicciones en tiempo real ha sido un paso significativo hacia la aplicación práctica de nuestros modelos. Esto no solo muestra la flexibilidad de Spark en integrarse con aplicaciones de tiempo real, sino también la importancia de crear interfaces que permitan la fácil interacción con los modelos de machine learning.

En conclusión, esta práctica ha confirmado que Apache Spark es una herramienta formidable para el procesamiento y análisis de grandes volúmenes de datos. Sin embargo, el éxito en su implementación depende de una comprensión profunda de los datos, una cuidadosa selección y validación de modelos, y una infraestructura robusta que pueda manejar la carga de trabajo distribuida. La experiencia adquirida aquí será invaluable para futuros proyectos en análisis de datos y machine learning, proporcionando una base sólida para abordar problemas complejos con confianza y eficacia.

En conclusión, esta práctica ha demostrado que Apache Spark es una herramienta poderosa para el procesamiento de grandes volúmenes de datos. La experiencia adquirida será invaluable para futuros proyectos de análisis de datos y machine learning, proporcionando una base sólida para abordar problemas complejos con eficacia.