

Fecha: 13/07/2025

Presentado por: Leonard Jose Cuenca Roa.

Ingeniería para el Procesado Masivo de Datos

Actividad 1. Actividad 1: Uso de HDFS, Spark SQL y MLi

Presentado por: Leonard Jose Cuenca Roa.

Fecha: 06/07/2025

Parte 1: Manejo de HDFS. Tras acceder a la terminal de Linux en Jupyter Lab.

Como parte de la preparación de la clase, el profesor Abel nos proporcionó un paquete optimizado para el uso de HDFS y su entorno de software para el análisis de **Big Data**. Este ejercicio se ejecutará en un ambiente de Windows y, a continuación, mostraré el avance de la actividad con capturas de pantalla:

Paso 1: Instalación de ambiente

Descripción: Se ha creado un directorio en la ruta C:\BDP que contiene el software necesario para la práctica. El procedimiento de instalación consiste en descomprimir los archivos y ejecutar los scripts .bat en la secuencia especificada.

El objetivo de este ejercicio es utilizar exclusivamente software libre, prescindiendo de servicios web comerciales que requieren un pago para su uso ó pro cargar una tarjeta de crédito, como puedo señalar se ve la ejecución del ambiente y el uso de los puertos en completa ejecución cada herramienta.

Evidencia:

Big Data Portable Launcher v3.5

Launcher Estado de Puertos

Controles Principales

☒ Iniciar TODO (Servicios + Jupyter)

Monitor del Sistema

CPU: CPU: 4.5%

RAM: RAM: 54.7%

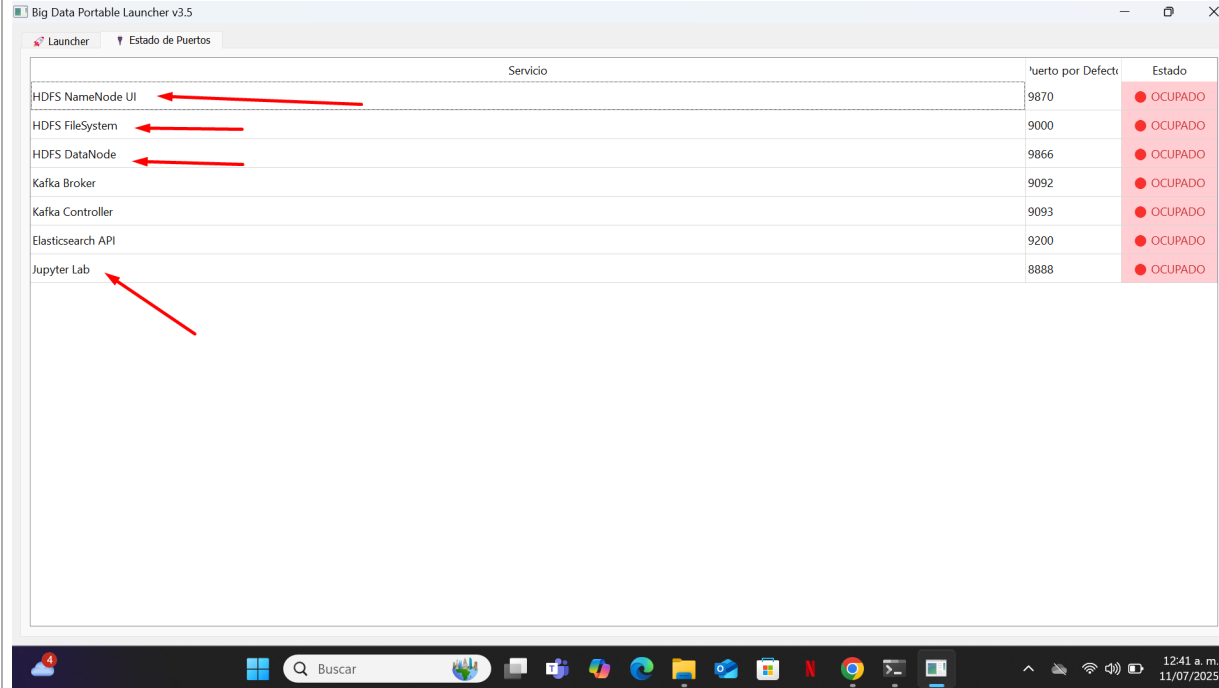
Explorador HDFS

Nombre	Tamaño (bytes)	Modificación
--------	----------------	--------------

Logs del Launcher

```
[00:06:08] HDFS DataNode iniciado con PID: 8692
[00:06:09] -> Lanzando Kafka KRaft...
[00:06:09] Kafka KRaft iniciado con PID: 1676
[00:06:09] -> Lanzando Elasticsearch...
[00:06:09] Elasticsearch iniciado con PID: 7956
[00:06:09] [INFO] Iniciando Jupyter Lab...
[00:06:11] [INFO] Puerto por defecto de Jupyter (8888) está libre.
[00:06:11] -> Lanzando Jupyter Lab...
[00:06:11] Jupyter Lab iniciado con PID: 10696
[00:06:11] =====
```

☐ Mantener siempre al frente

Paso 1: Instalación de ambiente**Paso 2: Ejercicio parte 1**

Descripción: Como parte del ejercicio de conocer los diferentes comandos básicos de hdfs, se ejecuta el siguiente comando para crear un directorio y se valida su generación

Evidencia:

```
Entorno de Big Data Portatil ( X ) + - v

=====
      ENTORNO DE BIG DATA PORTATIL - CONSOLA LISTA
=====

Variables de entorno configuradas para esta sesion:
- JAVA_HOME:    C:\BDP\common_jdk
- HADOOP_HOME:  C:\BDP\hadoop
- SPARK_HOME:   C:\BDP\spark
- KAFKA_HOME:   C:\BDP\kafka_kraft
- ES_HOME:      C:\BDP\elasticsearch

Ahora puedes usar todos los comandos directamente.

=====

C:\BDP>hdfs dfs -mkdir /CuencaRoaLeonardJose

C:\BDP>hdfs dfs -ls /
Found 2 items
drwxr-xr-x  - Kennyta supergroup          0 2025-07-11 00:54 /CuencaRoaLeonardJose
```

Paso 3: Ejercicio parte 1

Descripción: Como parte del seguimiento de la tarea de la parte 1, desde la interfaz local de Jupyter se sube el .csv que viene como documento adjunto de la actividad, se deje evidencia.

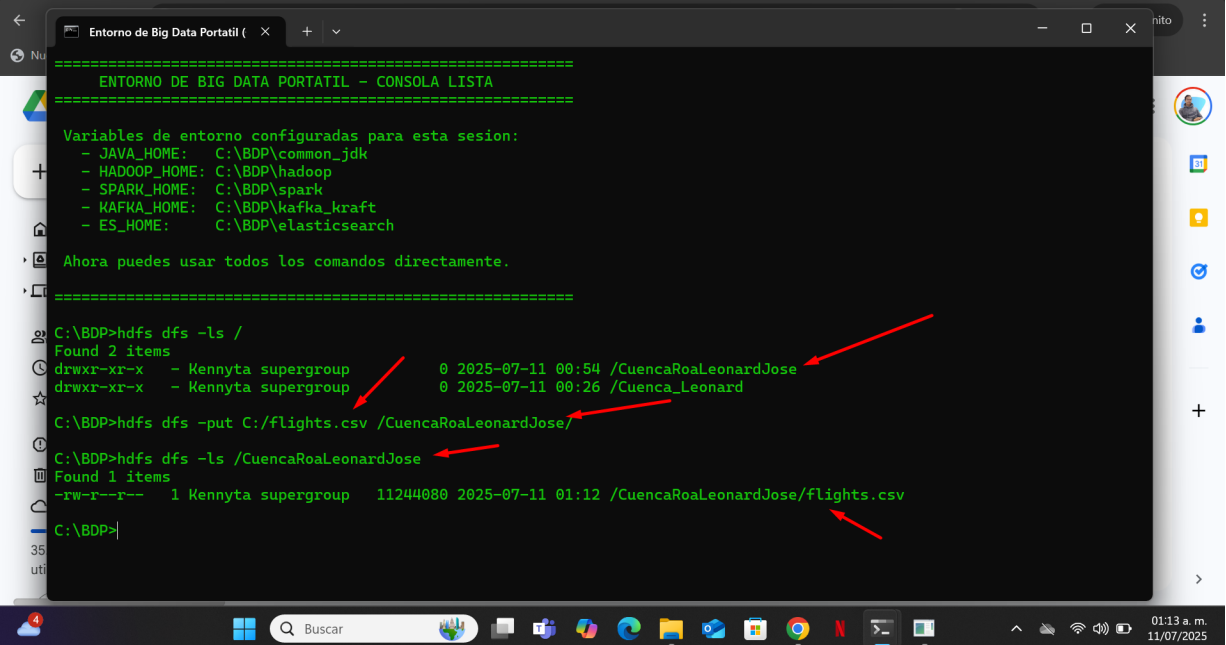
Evidencia:

The screenshot shows the JupyterLab interface with a file explorer on the left and a table view of the 'flights.csv' file. The table contains the following data:

	year	month	day	dep_time	dep_delay	arr_time
1	2014	1	1	1	96	235
2	2014	1	1	4	-6	738
3	2014	1	1	8	13	548
4	2014	1	1	28	-2	800
5	2014	1	1	34	44	325
6	2014	1	1	37	82	747
7	2014	1	1	346	227	936
8	2014	1	1	526	-4	1148
9	2014	1	1	527	7	917
10	2014	1	1	536	1	1334
11	2014	1	1	541	1	911
12	2014	1	1	549	24	907
13	2014	1	1	550	0	837
14	2014	1	1	557	-3	1134
15	2014	1	1	557	-3	825
16	2014	1	1	558	-2	801

Paso 4: Ejercicio parte 1

Descripción: Ahora para continuar con la actividad se cargó el archivo “**flights.csv**” al directorio HDFS, usando el comando “**hdfs dfs -put C:/flights.csv /CuencaRoaLeonardJose/**” permite cargar archivos al destino seleccionado en este caso **CuencaRoaLeonardJose** y usamos el comando “**hdfs dfs -put C:/flights.csv /CuencaRoaLeonardJose/**” se deja evidencia de la operación. Y para terminar se ejecutó el comando para validar que se cargo el archivo en el directorio creado para la practica para este caso el comando utilizado fue “**hdfs dfs -ls /CuencaRoaLeonardJose**”

Evidencia:

```
Entorno de Big Data Portatil ( x ) + - v
=====
ENTORNO DE BIG DATA PORTATIL - CONSOLA LISTA
=====
Variables de entorno configuradas para esta sesion:
- JAVA_HOME: C:\BDP\common_jdk
- HADOOP_HOME: C:\BDP\hadoop
- SPARK_HOME: C:\BDP\spark
- KAFKA_HOME: C:\BDP\kafka_kraft
- ES_HOME: C:\BDP\elasticsearch

Ahora puedes usar todos los comandos directamente.

=====
C:\BDP>hdfs dfs -ls /
Found 2 items
drwxr-xr-x - Kennyta supergroup 0 2025-07-11 00:54 /CuencaRoaLeonardJose
drwxr-xr-x - Kennyta supergroup 0 2025-07-11 00:26 /Cuenca_Leonard
C:\BDP>hdfs dfs -put C:/flights.csv /CuencaRoaLeonardJose/
C:\BDP>hdfs dfs -ls /CuencaRoaLeonardJose
Found 1 items
-rw-r--r-- 1 Kennyta supergroup 11244080 2025-07-11 01:12 /CuencaRoaLeonardJose/flights.csv
C:\BDP>
```

Parte 2: Manejo de Apache Spark con *notebooks* de Jupyter.

Para esta sección usaremos el JupyterLab suministrado en la actividad para mostrar el conocimiento adquirido.

Caso 1: Inferir de manera correcta o no los datos

Descripción: Para resolver este primer caso podremos usar dos metodos de spark, podremos usar el `.printSchema()` y `.show()`.

Método `printSchema`: Nos permite imprimir el esquema del DataFrame es decir la estructura o plano esto nos muestra en consola o pantalla el nombre de cada columna y el tipo de datos que Spark le asigno, lo podemos usar para hacer validaciones ya que los muestra en un formato de árbol fácil de leer.

Método `.show()`: Nos permite mostrar los datos reales del DataFrame generado recibe como parámetro la cantidad y esta se muestra en pantalla o en consola dependiendo el medio que se este usando.

Método `.count()`: Este metodo me permite mostrar la cantidad de registros nos ayuda en validar si existen un total de datos en conjunto.

Metodo de consulta: Tambien podemos validar si tenemos valores nulos Spark nos permite realizar una previa consulta y validar por ciertos campos si son nulos para este caso valores nulos son NA Para este caso debemos importar `pypark.sql` para poderlo usar.

Podemos aclarar lo siguiente `printSchema` me permite entender la estructura y `show` ver los datos manos me permite validar, analizar y depurar datos en Spark.

Para organizar esta sección organizare el código en un solo bloque para comprender y avanzar en los Tres ejercicios que se deben resolver:

```
Jupyter actividad_1 Last Checkpoint: 33 minutes ago
File Edit View Run Kernel Settings Help
[18]: from pyspark.sql import SparkSession
      from pyspark.sql import functions as F
      from pyspark.sql.types import IntegerType, DoubleType

      spark = SparkSession.builder.appName("MiPrimerAppSpark").getOrCreate()

      ruta_hdfs = "/CuencaRoLeonardJose/Flights.csv"
      # Descargar estas líneas
      flightsDF = spark.read.csv(ruta_hdfs, header="true", inferSchema="true")

      # 1. Imprimi el esquema del DataFrame para revisar los tipos de datos
      print("Esquema del DataFrame:")
      flightsDF.printSchema()

      # 2. Muestra las primeras 5 filas para inspeccionar los datos
      print("\nPrimeras 5 filas del DataFrame:")
      flightsDF.show(5)

      # 3. Mostramos el número de filas que tiene el DataFrame para hacernos una idea de su tamaño:
      flightsDF.count()

      # 4. Vamos a averiguar cuántas filas tienen el valor "NA" (como string) en la columna dep_time:
      cuantos_NA = flightsDF\
        .where(F.col("dep_time") == "NA")\
        .count()

      cuantos_NA

      # 5. En nuestro caso, como tenemos un número considerable de filas, vamos a quitar todas las filas donde hay un NA en cualquiera de las columnas.
      columnas_limpiar = ["dep_time", "dep_delay", "arr_time", "arr_delay", "air_time", "hour", "minute"]

      flightslimpiado = flightsDF
      for nombreColumna in columnas_limpiar: # para cada columna, nos quedamos con las filas que no tienen NA en esa columna
          flightslimpiado = flightslimpiado.where(F.col(nombreColumna) != "NA")

      flightslimpiado.cache()

      # 6. Si ahora mostramos el número de filas que tiene el DataFrame flightslimpiado tras eliminar todas esas filas, vemos que ha disminuido ligeramente
      flightslimpiado.count()

      # 7. Una vez que hemos eliminado los NA, vamos a convertir a tipo entero cada una de esas columnas que eran de tipo string.

      flightsConvertido = flightslimpiado

      for c in columnas_limpiar:
          # método que crea una columna o reemplaza una existente
          flightsConvertido = flightsConvertido.withColumn(c, F.col(c).cast(IntegerType()))

      flightsConvertido = flightsConvertido.withColumn("arr_delay", F.col("arr_delay").cast(DoubleType()))
      flightsConvertido.cache()

      flightsConvertido.printSchema()

      flightsConvertido.show(5)
```

Ejercicio 1 : Crear nuevo DataFrame AeropuertoOrigenDF y RutasDistintasDF

Descripción: Para este primer ejercicio se aplica conocimientos previos, es decir tengo un poco de experiencia en consultas SQL por lo que describir mi solución mostrando el código y dejando el resultado como evidencia.

Código:

```
# Contar aeropuertos de origen distintos
aeropuertosOrigenDF = flightsConvertido.select("origin").distinct()
n_origen = aeropuertosOrigenDF.count()
print(f"Número de aeropuertos de origen distintos: {n_origen}")
aeropuertosOrigenDF.show()

# Contar rutas distintas (combinaciones de origen y destino)
rutasDistintasDF = flightsConvertido.select("origin", "dest").distinct()
n_rutas = rutasDistintasDF.count()

print(f"Número de rutas distintas: {n_rutas}")
rutasDistintasDF.show()
```

Evidencia:

```
[42]: # Contar aeropuertos de origen distintos
aeropuertosOrigenDF = flightsConvertido.select("origin").distinct()
n_origen = aeropuertosOrigenDF.count()
print(f"Número de aeropuertos de origen distintos: {n_origen}")
aeropuertosOrigenDF.show()

# Contar rutas distintas (combinaciones de origen y destino)
rutasDistintasDF = flightsConvertido.select("origin", "dest").distinct()
n_rutas = rutasDistintasDF.count()

print(f"Número de rutas distintas: {n_rutas}")
rutasDistintasDF.show()
```

Número de aeropuertos de origen distintos: 2

```
+-----+
|origin|
+-----+
| SEA  |
| PDX  |
+-----+
```

Número de rutas distintas: 115

```
+-----+
|origin|dest|
+-----+
| SEA  |RND|
| SEA  |DTW|
| SEA  |CLE|
| SEA  |LAX|
| PDX  |SEA|
| SEA  |BLI|
| PDX  |IAH|
| PDX  |PHX|
| SEA  |SLC|
| SEA  |SBA|
| SEA  |BWI|
| PDX  |IAD|
| PDX  |SFO|
| SEA  |KOA|
| SEA  |JAC|
| PDX  |MCI|
| SEA  |SJC|
| SEA  |ABQ|
| SEA  |SAT|
| PDX  |ONT|
+-----+
```

only showing top 20 rows

```
[45]: assert(n_origen == 2)
assert(n_rutas == 115)
assert(aeropuertosOrigenDF.count() == n_origen)
assert(rutasDistintasDF.count() == n_rutas)
```

test - OK

Ejercicio 2: Retrasos

Descripción: Como lo indica el enunciado se generó una función que en Python que permita validar y resolver las incógnitas, para este caso divido el problema en 5 operaciones cruciales que se describen en el código.

Código:

```
def retrasoMedio(df):  
    # 1. Primero Filtro vuelos con retraso positivo  
    vuelos_con_retraso_positivo = df.where(F.col("arr_delay") > 0)  
  
    # 2. Segundo: Agrupo por aeropuerto de destino y  
    # 3. Tercero: Calculo el retraso medio  
    # 4. Cuarto: Renombro la columna del retraso medio  
    retraso_medio_df = vuelos_con_retraso_positivo.groupBy("dest") \  
        .agg(F.avg("arr_delay").alias("retraso_medio"))  
  
    # 5. Quinto: Ordeno el DataFrame resultante de mayor a menor retraso medio  
    df_final_ordenado = retraso_medio_df.orderBy(F.col("retraso_medio").desc())  
  
    return df_final_ordenado
```

Evidencia:

```
[55]: # Invocar La función con tu DataFrame flightsConvertido  
retrasoMedioDF = retrasoMedio(flightsConvertido)  
  
# Mostrar Los tres primeros resultados como esta ordenado podremos mostrar Los primeros tres de esta manera  
print("Los tres aeropuertos con mayor retraso medio y sus retrasos:")  
retrasoMedioDF.show(3)
```

Los tres aeropuertos con mayor retraso medio y sus retrasos:

```
+-----+  
|dest|      retraso_medio|  
+-----+  
| BOI|           64.75|  
| HDN|           46.8|  
| SFO|41.193768844221104|  
+-----+  
only showing top 3 rows
```



Ejercicio 3: Ajustar un modelo de DecisionTree de Spark para predecir si un vuelo vendrá o no con retraso

Descripción:

Código:

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.evaluation import BinaryClassificationEvaluator
from pyspark.sql import functions as F

# 0. Preparo la columna 'label'
flights_con_label = flightsConvertido.withColumn("label", F.when(F.col("arr_delay") > 0, 1).otherwise(0))

# 1. Defino los StringIndexers
indexerMonth = StringIndexer(inputCol="month", outputCol="monthIndexed")
indexerCarrier = StringIndexer(inputCol="carrier", outputCol="carrierIndexed")

# 2. Defino las columnas a usar en el VectorAssembler (incluyendo las indexadas)
feature_cols = ["monthIndexed", "day", "dep_time", "arr_time", "carrierIndexed", "distance", "air_time"]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")

# 3. Defino el modelo Decision Tree
dt = DecisionTreeClassifier(labelCol="label", featuresCol="features", seed=42)

# 4. Genero el Pipeline (orden de ejecución: indexers -> assembler -> dt)
pipeline = Pipeline(stages=[indexerMonth, indexerCarrier, assembler, dt])

# 5. Divido los datos (antes de fit_pipeline si el pipeline incluye el modelo)
train_data, test_data = flights_con_label.randomSplit([0.7, 0.3], seed=42)

# 6. Ajusto y entreno el Pipeline completo con los datos de entrenamiento
model = pipeline.fit(train_data)

# 7. Realizo las predicciones en el conjunto de prueba
predicciones = model.transform(test_data)

# 8. Evaluo el modelo
evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction",
metricName="areaUnderROC")
auc = evaluator.evaluate(predicciones)

# Muestro las predicciones
print(f"Área bajo la curva ROC (AUC) = {auc}")
predicciones.select("label", "prediction", "probability", "rawPrediction", "arr_delay").show(5)
```


Evidencia:

PREVALUABLES,SEAT,CL,CLONES,PREVALUATION,PROBABILITY,CARRIERS,ARR_DELAY,STATUS

Área bajo la curva ROC (AUC) = 0.4876462417378798

label	prediction	probability	rawPrediction	arr_delay
0	0.0	[0.73400176938956...]	[7467.0,2706.0]	-4.0
1	0.0	[0.73400176938956...]	[7467.0,2706.0]	219.0
1	1.0	[0.48260869565217...]	[444.0,476.0]	24.0
0	0.0	[0.73400176938956...]	[7467.0,2706.0]	-6.0
0	0.0	[0.65965814551505...]	[8722.0,4500.0]	-25.0

only showing top 5 rows

```
[58]: assert(isinstance(indexerMonth, StringIndexer))
      assert(isinstance(indexerCarrier, StringIndexer))
      assert(indexerMonth.getInputCol() == "month")
      assert(indexerMonth.getOutputCol() == "monthIndexed")
      assert(indexerCarrier.getInputCol() == "carrier")
      assert(indexerCarrier.getOutputCol() == "carrierIndexed")
```

test OK

Ejercicio 3 Parte 1: Crear una variable llamada vectorAssembler

Descripción: Para satisfacer la petición requerida, se creó una lista que contiene los nombres de las columnas del registro. Posteriormente, se importó el VectorAssembler de la librería pyspark.ml.feature y se le pasó esta lista como parámetro.

En resumen, el **VectorAssembler** es un objeto fundamental en el **Machine Learning** con **Spark**. Su propósito es tomar las columnas definidas en su parámetro **inputCols** (la lista que se generó) y concatenarlas en una única columna de tipo Vector. Esta columna de salida, que se define con el parámetro **outputCol** y se ha nombrado **features**, es esencial para el entrenamiento. El objetivo final de este proceso es preparar los datos para la generación de un clasificador de árbol de decisiones con ayuda de Spark.

Código:

```
from pyspark.ml.feature import VectorAssembler
```

```
# Paso 1: Genero la lista de variables de entrada (strings)
```

```
columnas_ensamblar = [
    "monthIndexed", # Columna con index
    "day",
    "dep_time",
    "arr_time",
    "carrierIndexed", # Columna con index
    "distance",
    "air_time"
]
```

```
# Paso 2: Genero el VectorAssembler
```

```
vectorAssembler = VectorAssembler(inputCols=columnas_ensamblar, outputCol="features")
```

```
[21]: from pyspark.ml.feature import VectorAssembler

# Paso 1: Crear La lista de variables de entrada (strings)
# Asegúrate de usar las columnas indexadas para 'month' y 'carrier'
columnas_ensamblar = [
    "monthIndexed", # Columna indexada
    "day",
    "dep_time",
    "arr_time",
    "carrierIndexed", # Columna indexada
    "distance",
    "air_time"
]

# Paso 2: Crear el VectorAssembler
# La columna de salida se llamará "features"
vectorAssembler = VectorAssembler(inputCols=columnas_ensamblar, outputCol="features")

print(vectorAssembler.getOutputCol())
print(vectorAssembler.getInputCols())

features
['monthIndexed', 'day', 'dep_time', 'arr_time', 'carrierIndexed', 'distance', 'air_time']

[22]: assert(isinstance(vectorAssembler, VectorAssembler))
assert(vectorAssembler.getOutputCol() == "features")
input_cols = vectorAssembler.getInputCols()
assert(len(input_cols) == 7)
assert("arr_delay" not in input_cols)
```

tests ok

Ejercicio 3 Parte 2: Utilizar Binarizador en Spark

Descripción: Validando lo que se desea realizar a petición del ejercicio, puedo considerar que usar Binarizador de Spark es transformar nuestro problema a un clasificador binario con el propósito de predecir si hay retraso significativo o no, ya que la columna arr_delay_binary es mi label para el modelo.

Código:

```
from pyspark.ml.feature import Binarizer
from pyspark.sql import functions as F

# 1. Definí la variable delayBinarizer
# inputCol: La columna original de retraso ('arr_delay').
# outputCol: El nombre de la nueva columna binaria ('arr_delay_binary').
# hreshold: El umbral. Valores > threshold serán 1.0, valores <= threshold serán 0.0.
delayBinarizer = Binarizer(inputCol="arr_delay", outputCol="arr_delay_binary", threshold=15.0)

# Aplicamos el Binarizer en el DataFrame flightsConvertido solo para validarlo

df_con_binario = delayBinarizer.transform(flightsConvertido)
df_con_binario.select("arr_delay", "arr_delay_binary").show(5)
```

```
[25]: from pyspark.ml.feature import Binarizer
      from pyspark.sql import functions as F

      # 1. Definó la variable delayBinarizer
      # inputCol: La columna original de retraso ('arr_delay').
      # outputCol: El nombre de la nueva columna binaria ('arr_delay_binary').
      # threshold: El umbral. Valores > threshold serán 1.0, valores <= threshold serán 0.0.
      delayBinarizer = Binarizer(inputCol="arr_delay", outputCol="arr_delay_binary", threshold=15.0)

      # Aplicamos el Binarizer en el DataFrame flightsConvertido solo para validarlo

      df_con_binario = delayBinarizer.transform(flightsConvertido)
      df_con_binario.select("arr_delay", "arr_delay_binary").show(5)
```

arr_delay	arr_delay_binary
70.0	1.0
-23.0	0.0
-4.0	0.0
-23.0	0.0
43.0	1.0

only showing top 5 rows

```
[26]: assert(isinstance(delayBinarizer, Binarizer))
      assert(delayBinarizer.getThreshold() == 15)
      assert(delayBinarizer.getInputCol() == "arr_delay")
      assert(delayBinarizer.getOutputCol() == "arr_delay_binary")
```

test -> OK

Ejercicio 3 Parte 3: El NoteBook nos indica generar un Modelo de Clasificación

Descripción: Quedo muy sorprendido Spark tiene un sin fin de cualidades y unas de ellas es que también se puede usar El modelo de clasificación de árbol de decisión, para este caso lo vi sencillo su implementación, primero lo importamos del paquete `pyspark.ml.classification`, como nos apoyamos en generar un **features**, le estamos indicando al clasificador aquí están todas mis variables predictorias ya están combinadas y listas para usarse gracias al generar el **VectorAssembles**, EL **labelCol** le indicamos que debe aprender a predecir si un vuelo se retrasa o no, con el apoyo de **Binarizer** y generar mi **columnas model**.

Código:

```
from pyspark.ml.classification import DecisionTreeClassifier
```

```
# Por aprendizaje de otras materias en el uso de arbol de decisiones usare semillas para evitar un poco el sesgo
```

```
decisionTree = DecisionTreeClassifier(
    featuresCol="features",
    labelCol="arr_delay_binary",
    seed=42
)
```

```
print("--- Información del Estimador DecisionTreeClassifier ---")
print("El objeto 'decisionTree' es un estimador listo para aprender.")
print("Está configurado con los siguientes parámetros:")
print(decisionTree)
print("-----")
```

```
[33]: from pyspark.ml.classification import DecisionTreeClassifier

      # Por aprendizaje de otras materias en el uso de arbol de decisiones usare semillas para evitar un poco el sesgo

      decisionTree = DecisionTreeClassifier(
          featuresCol="features",
          labelCol="arr_delay_binary",
          seed=42
      )

      print("--- Información del Estimator DecisionTreeClassifier ---")
      print("El objeto 'decisionTree' es un estimador listo para aprender.")
      print("Está configurado con los siguientes parámetros:")
      print(decisionTree)
      print("-----")

      --- Información del Estimator DecisionTreeClassifier ---
      El objeto 'decisionTree' es un estimador listo para aprender.
      Está configurado con los siguientes parámetros:
      DecisionTreeClassifier_943ccb633789
      -----

[34]: assert(isinstance(decisionTree, DecisionTreeClassifier))
      assert(decisionTree.getFeaturesCol() == "features")
      assert(decisionTree.getLabelCol() == "arr_delay_binary")
```

test -> OK

Ejercicio 3 Parte 4: Generamos un Pipeline

Descripción: Para finalizar se generó el Pipeline Spark exige ser organizado para crear los Pipeline, el código diseñado permite explicar cada paso, sinceramente me costo entender un poco todo el concepto por lo que en el código logrado genero una secuencia de pasos y la ayuda del NoteBook en sus validaciones permite saber si estamos en el camino correcto.

Código:

```
# Paso 1: genero el objeto Pipeline
# Se encadeno todas las etapas en el orden lógico de procesamiento:
# StringIndexer -> Binarizer (para la etiqueta) -> VectorAssembler -> DecisionTreeClassifier
pipeline = Pipeline(stages=[
    indexerMonth,
    indexerCarrier,
    delayBinarizer,
    vectorAssembler,
    decisionTree
])

print(":D Pipeline creado exitosamente.")
print(f"Etapas del Pipeline: {[stage.uid for stage in pipeline.getStages()]}")

# Paso 2: :D Se Entrenó el Pipeline
# Se invoco el método 'fit()' sobre el pipeline, pasándole el DataFrame de entrenamiento.
# En este caso, por simplicidad, usamos el DataFrame completo 'flightsConvertido' como
entrenamiento.
pipelineModel = pipeline.fit(flightsConvertido)

print("\n :D Pipeline entrenado (ajustado a los datos) con éxito.")
print("El objeto 'pipelineModel' ahora contiene todos los transformadores ajustados y el modelo de
clasificación entrenado.")

# Paso 3: :D Aplicar el Pipeline entrenado para transformar (predecir) los datos
# Se invoco el método 'transform()' sobre el pipelineModel para generar predicciones.
flightsPredictions = pipelineModel.transform(flightsConvertido)

print("\n :D Predicciones generadas y guardadas en 'flightsPredictions' DataFrame.")
print("Este DataFrame ahora incluye las columnas de las características ensambladas,")
print("la etiqueta real ('arr_delay_binary'), la predicción ('prediction'),")
print("y las probabilidades ('probability') del modelo para cada clase.")

# Paso 4: XD Mostrar algunas de las columnas relevantes para verificar las predicciones
print("\n--- Primeras 5 filas del DataFrame de Predicciones ---")
flightsPredictions.select("arr_delay", "arr_delay_binary", "prediction", "probability", "features").show(5,
truncate=False)
print("-----")
```

C:\WINDOWS\System32\WinRmX

actividad_1.ipynb

+

Notebook Python 3 (ipykernel)

```
[54]: # Paso 1: genero el objeto Pipeline
# Se encadenan todas las etapas en el orden lógico de procesamiento:
# StringIndexer -> Binarizer (para la etiqueta) -> VectorAssembler -> DecisionTreeClassifier
pipeline = Pipeline(stages=[
    StringIndexer,
    Binarizer,
    VectorAssembler,
    DecisionTreeClassifier
])

print(" :D Pipeline creado exitosamente.")
print(f"Etapas del Pipeline: {[stage.uid for stage in pipeline.getStages()]}")

# Paso 2: :D Se Entrenó el Pipeline
# Se invoca el método 'fit()' sobre el pipeline, pasándole el DataFrame de entrenamiento.
# En este caso, por simplicidad, usamos el DataFrame completo 'flightsConvertido' como entrenamiento.
pipelineModel = pipeline.fit(flightsConvertido)

print("\n :D Pipeline entrenado (ajustado a los datos) con éxito.")
print("El objeto 'pipelineModel' ahora contiene todos los transformadores ajustados y el modelo de clasificación entrenado.")

# Paso 3: :D Aplicar el Pipeline entrenado para transformar (predecir) los datos
# Se invoca el método 'transform()' sobre el pipelineModel para generar predicciones.
flightsPredictions = pipelineModel.transform(flightsConvertido)

print("\n :D Predicciones generadas y guardadas en 'flightsPredictions' DataFrame.")
print("Este DataFrame ahora incluye las columnas de las características ensambladas,")
print("la etiqueta real ('arr_delay_binary'), la predicción ('prediction'),")
print("y las probabilidades ('probability') del modelo para cada clase.")

# Paso 4: XD Mostrar algunas de las columnas relevantes para verificar las predicciones
print("\n--- Primeras 5 filas del DataFrame de Predicciones ---")
flightsPredictions.select("arr_delay", "arr_delay_binary", "prediction", "probability", "features").show(5, truncate=False)
print("-----")
```

```
:D Pipeline creado exitosamente.
Etapas del Pipeline: ['StringIndexer_4d60be859ea1', 'StringIndexer_46f6881a0e05', 'Binarizer_805f7fd79cfb', 'VectorAssembler_11f3527f07db', 'DecisionTreeClassifier_6c6c6fd715a6']

:D Pipeline entrenado (ajustado a los datos) con éxito.
El objeto 'pipelineModel' ahora contiene todos los transformadores ajustados y el modelo de clasificación entrenado.

:D Predicciones generadas y guardadas en 'flightsPredictions' DataFrame.
Este DataFrame ahora incluye las columnas de las características ensambladas,
la etiqueta real ('arr_delay_binary'), la predicción ('prediction'),
y las probabilidades ('probability') del modelo para cada clase.

--- Primeras 5 filas del DataFrame de Predicciones ---
+-----+-----+-----+-----+-----+
|arr_delay|arr_delay_binary|prediction|probability|features|
+-----+-----+-----+-----+-----+
|70.0     |1.0             |1.0       |[0.3,0.7]  |[10.0,1.0,1.0,235.0,0.0,1542.0,194.0]|
|-23.0    |0.0             |0.0       |[0.9140647187557427,0.08593528124425734]|[10.0,1.0,4.0,738.0,6.0,2279.0,252.0]|
|-4.0     |0.0             |0.0       |[0.9140647187557427,0.08593528124425734]|[10.0,1.0,8.0,548.0,4.0,1825.0,201.0]|
|-23.0    |0.0             |0.0       |[0.9140647187557427,0.08593528124425734]|[10.0,1.0,28.0,800.0,6.0,2282.0,251.0]|
|43.0     |1.0             |1.0       |[0.3,0.7]  |[10.0,1.0,34.0,325.0,0.0,1448.0,201.0]|
+-----+-----+-----+-----+-----+
only showing top 5 rows
```

```
3]: from pyspark.ml import PipelineModel
assert(isinstance(pipeline, Pipeline))
assert(len(pipeline.getStages()) == 5)
assert(isinstance(pipelineModel, PipelineModel))
assert("probability" in flightsPredictions.columns)
assert("prediction" in flightsPredictions.columns)
assert("rawPrediction" in flightsPredictions.columns)
```

test -> ok

Vamos a mostrar la matriz de confusión (este apartado no es evaluable). Agrupamos por la variable que tiene la clase verdadera y la que tiene la clase predicha, para ver en cuántos casos coinciden y en cuántos difieren.

```
[58]: flightsPredictions.groupBy("arr_delay_binary", "prediction").count().show()

+-----+-----+-----+
|arr_delay_binary|prediction|count|
+-----+-----+-----+
|1.0             |1.0       |752|
|0.0             |1.0       |181|
|1.0             |0.0       |23497|
|0.0             |0.0       |136318|
+-----+-----+-----+
```

Hasta el resultado final

[]:

Conclusiones de la Practica

Esta actividad la considere un reto debido que mis limitaciones en esta tecnología son amplias pero con ayuda de las clases, las guías y algunos ejercicios previos pude llevarlo hasta concluirla, punto que considero importantes, con los diferente ejercicios pude comprender las capacidades de PySpark para el análisis y modelado de datos para este caso un problema planteado de un aeropuerto X.

Se inició con la la preparación y limpieza de datos, cargando un archivo CSV, inspeccionando su esquema y limpiando valores nulos para asegurar la calidad de la información. También realizamos conversiones de tipos de datos cruciales para preparar las columnas para operaciones numéricas.

Posteriormente, exploramos el conjunto de datos para realizar un análisis exploratorio básico. Esto incluyó la identificación y el conteo de aeropuertos de origen distintos, así como el cálculo de rutas únicas (combinaciones de origen y destino). Estas operaciones de selección, distinción y conteo se me hizo fundamental para obtener **insights** iniciales sobre la distribución y las relaciones en los datos.

Y como tarea final, se generó un modelo de clasificación para predecir retrasos de vuelos utilizando un **DecisionTreeClassifier**. Para ello, se transformó el retraso continuo en una variable binaria `arr_delay_binary` con un **Binarizer** y preparamos las variables **predictoras** (como mes, aerolínea, tiempos) mediante **StringIndexer** para categorizar y **VectorAssembler** para consolidarlas en un formato **features** que el modelo pudiera procesar. Se encapsuló todas estas etapas en un **Pipeline de Spark**, se entrenó y se aplicó para generar predicciones, con el fin de comprender un flujo de trabajo completo de Machine Learning con PySpark.

En mi opinión de novato **Spark** es muy completo y versátil y con gran ayuda de Python que es ahí donde tengo fortaleza por mi experiencia como desarrollador de software, es sin duda una gran tecnología para aprender y comprender el mundo del Big Data.