Guía de la sesión sobre PySpark

Alejandro Fernández Camello y Eduardo Cano García

10 de diciembre de 2022

Índice general

Lista de códigos

1 — Introducción

1.1. Propósito

Mediante esta documentación, junto al **Jupyter Notebook**, se desarrolla la sesión para el proyecto de AOS sobre **Apache Spark**.

La estructura de la práctica consistirá en primer lugar en un pequeño apartado para localizar la instalación de **Apache Spark**, enlazarla con **PySpark** e iniciar una sesión de **PySpark**.

Después de este apartado introductorio y de los pasos previos a seguir, vendrán dos grandes apartados que analizarán las dos estructuras de datos más importantes con las que cuenta **PySpark**. Estas son el **DataFrame** y **RDD (Resilient Distributed Dataset)**.

Se mostrarán ejemplos del uso de ambas estructuras de datos y, después de algunos ejemplos, habrá pequeños ejercicios a realizar por parte de los alumnos. Se espera que estos pequeños ejercicios se puedan resolver en 1-2 minutos. Al final de cada una de las secciones habrá un ejercicio más largo que requerirá alrededor de unos 5-10 minutos. Este ejercicio requerirá usar todos los conocimientos adquiridos previamente en esa sección

para poder resolverlo satisfactoriamente.

1.2. Contexto del ejemplo

Durante esta sesión se va a trabajar utilizando el *dataset* del Titanic.

Dicho *dataset* almacena un conjunto de datos de interés acerca de la tragedia ocurrida.

Nos ha parecido un ejemplo de adecuada dificultad, a la par que ilustrativo, por lo que nos hemos decantado por ese *dataset*. Cuenta con casi 900 filas y 12 columnas. En esta documentación se ha cogido solo la parte de entrenamiento. Las columnas son las que se describen a continuación:

1. **PassengerId**. Indica el número del pasajero.
2. **Survived**. Indica si sobrevivió (valor 1) o si falleció (valor 0).
3. **Pclass**: Indica la clase:
   - 1: Primera clase
   - 2: Segunda clase
   - 3: Tercera clase
4. **Name**. Indica el nombre del pasajero.
5. **Sex**. Indica género del pasajero: masculino (male) o femenino (female).
6. **Age**. Indica la edad del pasajero.
7. **SibSp**. Indica el número de hermanos y parejas del pasajero.

8. **Parch**. Indica el número de padres/hijos del pasajero, los cuales se encontraban a bordo.
9. **Ticket**. Indica el billete del pasajero.
10. **Fare**. Indica la tarifa pagada.
11. **Cabin**. Indica la cabina en que se encontraba la persona.
12. **Embarked**. Indica el lugar de embarque. C = Cherbourg, Q = Queenstown, S = Southampton.

2 — Pasos previos

Una vez inicializado el contenedor correspondiente a la práctica, es necesario cargar en el **Jupyter Notebook** el cuaderno correspondiente a la sesión junto al *dataset* del Titanic.

En caso de duda, revisar la guía de instalación.

Los archivos con que se va a trabajar pueden ser obtenidos en el siguiente repositorio de **Github**:

`https://github.com/alexfdez1010/pyspark_aos`

O bien en la carpeta de **OneDrive** proporcionada en la entrega con el siguiente *link*:

Carpeta de OneDrive

Una vez subidos mediante el botón *upload* del **Notebook** debe quedar la interfaz gráfica similar a la Figura 2.1.

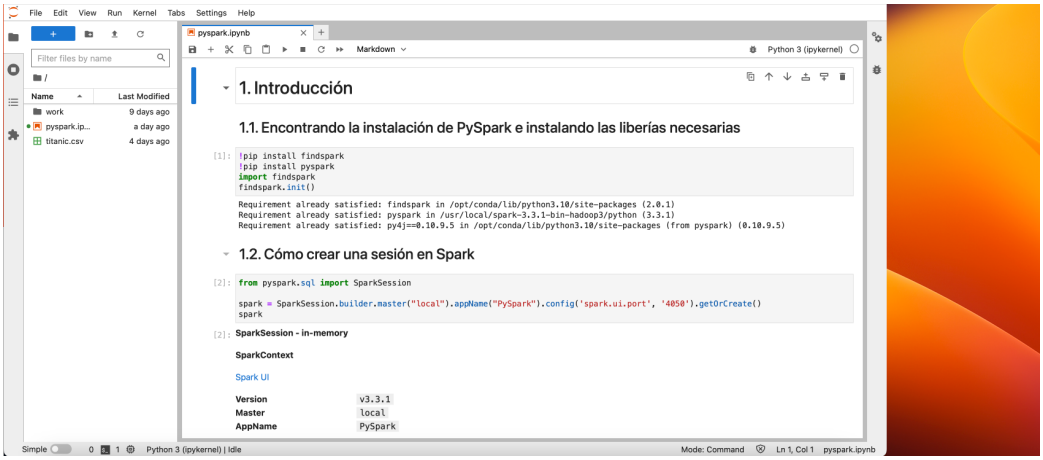

Figura 2.1: Interfaz gráfica del Notebook una vez subidos los archivos

Teniendo estos archivos subidos (**pyspark.ipynb** y **titanic.csv**) ya se puede comenzar la práctica.

3 — Ejemplos

3.1. Librerías e instalación

Antes de empezar con el programa es necesario vincular la instalación de **PySpark** con **Apache Spark**.

Para ello, se procede a instalar la librería **findspark**, la cual se encarga de encontrar la instalación de PySpark y vincularla con Apache Spark automáticamente.

Se usan en las líneas 1-2 los comandos mágicos **pip install**, precedidos de una exclamación para que el notebook lo reconozca.

En caso de que no estén estas librerías previamente instaladas, las descarga e instala automáticamente.

Se denominan comnados mágicos ya que se realizan sobre el sistema en vez de la terminal de **Python**.

En la línea 3 se importa la librería **findspark** y después se ejecuta **findspark.init()**, que busca y enlaza la instalación de **PySpark**.

```
!pip install findspark
!pip install pyspark
import findspark
findspark.init()
```

Código 1: Instalación e importaciones

3.2. Creación de la sesión

Antes de comenzar a trabajar con **Spark** es necesario crear una sesión. Una sesión permite ejecutar todos los servicios de **Spark**. Antes se usaba el **Context**, el cual solo permitía en ese momento ejecutar un servicio concreto.

A la hora de crear la sesión es necesario especificar cuál será el nodo maestro, el nombre de la aplicación y en qué puerto se ejecutará la interfaz gráfica que permitirá monitorizar los diferentes procesos.

Para acceder a la interfaz gráfica, se deberá introducir en un buscador después de ejecutar la celda 2 lo siguiente:

`http://localhost:4050`

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local").appName("PySpark")
        .config('spark.ui.port', '4050').getOrCreate()
spark
```

Código 2: Creación de una sesión en Spark

3.3. DataFrame

3.3.1. Leer un csv

El primer paso consiste en leer el archivo CSV (Comma-Separated Values) donde se encuentra el *dataset*. Esto se puede hacer de manera sencilla con la librería **spark.read** que permite leer diferentes formatos incluido CSV.

En este acaso ya que se encuentra un *header* en el CSV es necesario especificar que **header=True** para que considere que la primera fila es la cabecera.

Después de ello se imprime el esquema para ver los tipos y valores de cada columna.

```
1 df_titanic = spark.read.csv('titanic.csv',header=True)
2 df_titanic.printSchema()
```

Código 3: Lectura de un csv

3.3.2. Eliminando las columnas no necesarias

A la hora de procesar los datos solo se van a tener en cuenta un subconjunto pequeño de columnas por lo que para no gastar memoria se eliminan aquellas que no se usarán en pasos posteriores.

Las columnas eliminadas son aquellas que por decisión humana se consideran que no serán útiles para un tratamiento estadístico o de aprendizaje automático. Algunos ejemplos de columnas de este tipo son el identificador de la fila, el número de *ticket*, etc.

Para eliminar las columnas se podría usar el método **drop** o bien seleccionar solo las columnas relevantes con **select**. Se

usará el segundo método.

Se han clasificado las columnas relevantes en tres tipos.

1. NUMERIC_COLUMNS: Columnas con valores numéricos
2. ENUM_COLUMNS: Columnas que solo pueden tomar un número reducido de valores y cuyo orden dentro de los valores no es relevante.
3. LABEL_COLUMN: Columna que indica la etiqueta. Este valor es el que suele buscar predecir. En este caso es si sobrevive o no. Por lo que es una decisión binaria.

Al método **select** se le pasa una lista con todas las columnas a seleccionar. Después de ello se imprime el esquema para comprobar que realmente se han eliminado las columnas.

```
NUMERIC_COLUMNS = ["Age","SibSp","Parch","Fare"]
ENUM_COLUMNS = ["Embarked","Pclass","Sex"]
LABEL_COLUMN = ["Survived"]

df_titanic = df_titanic.select(
    NUMERIC_COLUMNS + ENUM_COLUMNS + LABEL_COLUMN
)
df_titanic.printSchema()
```

Código 4: Eliminar columnas

3.3.3. Transformando el DataFrame

Una de los métodos más usados para transformar un **DataFrame** es **transform**. Este método aplica una función al **DataFrame** completo. En este caso se utilizará para convertir las columnas perteneciente a la categoría de numéricas a *float* para facilitar su procesamiento posterior.

Se usa la función **select** para aplicar las transformaciones individuales a cada columna. Si es una columna numérica aplica el método **cast** que convierte toda la columna en un determinado tipo seleccionado por su parámetro. En este caso es *float*. Si no es numérica se deja la columna tal y como es.

A **transform** se le pasa la función a aplicar. De nuevo se imprime el esquema para comprobar que las columnas han cambiado de tipo.

```
from pyspark.sql.functions import col
def cast_to_float(input_df):
    return input_df.select([col(col_name)
    .cast("float") if col_name in NUMERIC_COLUMNS
        else col(col_name)
        for col_name in input_df.columns])

df_titanic = df_titanic.transform(cast_to_float)
df_titanic.printSchema()
```

Código 5: Transformando un Dataframe

3.3.4. Filtrando por las columnas

Filtrar las columnas por determinados valores es una de las operaciones más usuales cuando se trabaja con **DataFrames**. Se puede utilizar **where** (más orientado a programadores funcionales) o **SQL** (más orientado a programadores provenientes de bases de datos).

Filtrando por las columnas usando where

Para filtrar las columnas se puede hacer uso de la función **where**. A esta función se le pasa una sentencia booleana que indica la condición. Si la fila cumple la condición se mantiene y

en caso contrario es eliminada. En este caso se filtra para obtener solo los supervivientes que son aquellos que tienen la columna **Survived** a 1.

Después con el método **show()** se muestran las 10 primeras ocurrencias. Cuando se usa **show()** se efectúan todos los cálculos anteriores excepto el de leer el **CSV** que si lo realiza en el momento.

```
df_survived = df_titanic.where(col("Survived") == "1")
df_survived.show()
```

Código 6: Filtrar por columnas usando where

Filtrando por las columnas usando SQL

En esta caso también se filtrará por los supervivientes, pero el método será muy diferente. En primer lugar, se registra una vista temporal asociada a ese **DataFrame** para que se puede operar con **SQL** sobre él.

Una vez realizado esto ya se puede realizar la sentencia **SQL**. La sentencia **SQL** se le pasa a **spark.sql**. La sentencia es la más sencilla de filtrar por un **WHERE**. Después se muestran los resultados debiendo obtenerse los mismos que en el apartado anterior.

```
df_titanic.createOrReplaceTempView("df_titanic")
df_sql_survived = spark.sql("SELECT * FROM df_titanic WHERE survived='1'")
df_sql_survived.show()
```

Código 7: Filtrar por columnas usando SQL

3.3.5. Resumen estadístico del DataFrame

Los **DataFrames** pueden generar una tabla con un resumen estadístico del propio **DataFrame**. Este resumen incluye por

columna la media, valores máximos y mínimos, percentiles, etc.

Es muy útil para obtener rápidamente informacción sobre la tabla antes de aplicar otros procesamientos que consumen una gran ccantidad de tiempo.

```
1 df_titanic.select(["Age","SibSp","Parch","Pclass"]).summary().show()
```

Código 8: Resumen estadístico de un DataFrame

3.3.6. Agrupando columnas

Un **DataFrame** es muy similar a una tabla o vista de una base de datos por lo que es natural que existan operaciones de agrupamiento y agregación.

Las funciones de agrupamiento permite agrupar filas por una característica que tengan en común y después a las columnas no pertenecientes a la agrupación se les aplican funciones de agregación que calculan un nuevo valor para las tuplas agrupadas en base a los valores de esas columnas.

En este caso se agrupa por la clase en la que habían embarcado en el barco y se calculan de operaciones de agregación el número de personas en cada una de ellas y la media de edad.

De esta manera se puede obtener una información muy interesante de cuántas personas había en cada clase y cuál era su edad.

```
import pyspark.sql.functions as F
df_titanic_groupby_class = df_titanic.groupby("Pclass")
    .agg(
        F.count("Pclass").alias("Number of ocurrences"),
        F.mean("Age").alias("Average age"))
    .orderBy("Pclass",ascending=True)
df_titanic_groupby_class.show()
```

Código 9: Agrupar columnas de un DataFrame

3.3.7. Definiendo funciones UDF (User Defined Functions) y aplicándolas sobre una columna concreta

Además de las funciones proporcionadas por **Spark** se pueden crear las nuestras personalizadas para las tareas concretas que deseemos hacer.

Para simplificar este proceso se cuenta con el decorador **udf**. Un decorador en **Python** es una función que se aplica sobre otras funciones. En este caso convierte una función que procesa una columna de una única tupla en una función capaz de procesar todas las filas en paralelo y distribuido.

Como se puede ver se coloca el decorador encima de la función con '@' delante y solo con eso ya se habría convertido la función. Probar a quitar el decorador y ver qué pasa.

en este caso se aplica una función muy sencilla que reduce la clase en uno. Es necesario transformar los datos a entero ya que es un *string* y después se vuelve a transformar un *string* para restablecer su tipo.

Después se utiliza el método **withColumn** para aplicar la función a una única columna. Este método permite crear una

nueva columna en el **DataFrame** manteniendo las anteriores. En este caso al tener el mismo nombre una ya presente sobreescribe a la ya existente.

```
from pyspark.sql.functions import udf

@udf
def substract_one(x):
    return str(int(x)-1)

df_titanic_substracted = df_titanic
    .withColumn("Pclass",substract_one(col("Pclass")))
df_titanic_substracted.show(10)
```

Código 10: Funciones UDF sobre una columna

3.4. RDD

En el siguiente código se inicializa el contexto a partir de sparkContext, el cual representa la conexión con el cluster de Spark, y es utilizado a continuación para crear un RDD.

Tras ello, se usa el método parallelize (función de SparkContext) para crear un RDD a partir de una lista inidicada por parámetros.

Con take, se recupera el número de elementos indicado por parámetros. En este caso, los primeros 15 elementos.

```
sc = spark.sparkContext
rdd = sc.parallelize(range(1,101))
rdd
rdd.take(15)
```

Código 11: Crear un RDD y tomar una muestra

3.4.1. Filtrando

En el siguiente código se utiliza la función filter, la cual devuelve un nuevo RDD que contiene solo los elementos que cumplen el predicado pasado por parámetros.

En este caso, se recuperan los 10 primeros elementos del RDD, cuyo valor sea superior a 50.

```
1 rdd_filtered = rdd.filter(lambda x: x > 50)
2 rdd_filtered.take(10)
```

Código 12: Filtrar un RDD

3.4.2. Aplicando funciones a todos los elementos

En el siguiente código se utiliza la función map. Dicha función es una transformación de un RDD, la cual se usa para aplicar la función lambda recibida por parámetros a cada elemento del RDD o DataFrame indicado, devolviendo el nuevo RDD/DataFrame.

La función map() normalmente se usa para aplicar operaciones costosas y su salida tiene siempre el mismo número de registros que su entrada.

En este caso, cada elemento del RDD será el valor del antiguo, pero multiplicado por 100. A continuación, se devuelven los 5 primeros elementos del RDD.

```
1 rdd_mapped = rdd.map(lambda x: x*100)
2 rdd_mapped.take(5)
```

Código 13: Aplicar funciones a los elementos del RDD

3.4.3. Obteniendo una parte aleatoria del RDD

En el siguiente código se utiliza la función sample:

```
RDD.sample(withReplacement: bool, fraction: float, seed: Optional[int] = None)
```

Código 14: Función sample

Dicha función recibe un bool indicando en withReplacement si los elementos pueden ser tomados múltiples veces. También recibe un float, indicando el tamaño esperado de la muestra como fracción del tamaño del RDD.

En caso de que el primer parámetro sea falso, el valor de fraction estará entre 0 y 1 e indicará la fracción de filas a generar.

Si el primer parámetro es verdadero, el valor de fraction será igual o superior a 0.

El último parámetro (opcional) es un entero e indica la semilla para generar números aleatorios.

```
rdd_fraction = rdd.sample(withReplacement=True,fraction=0.1)
rdd_fraction.take(10)
```

Código 15: Obtener parte aleatoria de un RDD

3.4.4. Reduciendo por clave

Para el primer bloque (líneas 1-5 del código):

Se calcula el resto de dividir por 7 los elementos del RDD y se muestra el resultado para los 7 primeros elementos.

Para el segundo bloque (líneas 7-10):

Se utiliza la función del RDD reduceByKey(), la cual se emplea para combinar los valores de cada clave, a partir de la función lambda que recibe por parámetros.

Se está indicando en la función lambda que, para cada elemento se sume el nuevo valor al acumulador con el valor total guardado hasta ese momento.

En la línea 10 se indica que se devuelva cada elemento del RDD a partir de la función collect().

La función collect() se utiliza para recuperar todos los elementos de los nodos de vuelta al nodo principal.

Así, se está agrupando también por el resto de la división por 7 para los elementos del RDD.

```
1  rdd_groupby_remainder_of_seven = [(x % 7, x) for x in range(1,101)]
2  rdd_groupby_remainder_of_seven = sc.parallelize(
3      rdd_groupby_remainder_of_seven
4  )
5  rdd_groupby_remainder_of_seven.take(7)
6
7  rdd_groupby_sum = rdd_groupby_remainder_of_seven.reduceByKey(
8      lambda accu, value: accu+value
9  )
10 [element for element in rdd_groupby_sum.collect()]
```

Código 16: Reducir un RDD por clave

3.4.5. Reduciendo el RDD a un único valor

En el siguiente código se utiliza la función fold, la cual agrega los elementos de cada partición y devuelve el resultado para todas utilizando una función y un valor neutro llamado 'zeroValue'.

Es decir, que el primer parámetro verá sustituido su valor por el indicado por la función lambda que se ha pasado como segundo parámetro.

Como resultado, se obtiene la multiplicación de los valores del RDD.

```
mult_value = rdd.fold(1,
    lambda accu, value: accu*value
)
mult_value
```

Código 17: Reducir un RDD a un único valor

A continuación, se muestran los datos para el ejercicio final propuesto sobre RDD.

```
from random import choice, randint
from string import ascii_lowercase as letters
data = [
    (f"{choice(letters)}{choice(letters)}"
    , randint(1,10)) for _ in range(10000)
]
data[:10]
```

Código 18: Datos - ejercicio final RDD

4 — Soluciones a los ejercicios propuestos

4.1. DataFrame

4.1.1. Filtrando por columnas

```
df_poor = df_titanic.where(col("Fare") < 10)
df_poor.show()
```

Código 19: Solución - Filtrando por columnas con where

```
df_titanic.createOrReplaceTempView("df_titanic")
df_poor_sql = spark.sql("SELECT * FROM df_titanic WHERE Fare < 10")
df_poor.show()
```

Código 20: Solución - Filtrando por columnas con SQL

4.1.2. Resumen estadístico del DataFrame

```
df_titanic.select(["Embarked","Pclass","Sex"]).summary().show()
df_titanic.printSchema()
```

Código 21: Solución - Resumen estadístico del DataFrame

4.1.3. Agrupando columnas

```
df_titanic.select(["Embarked","Pclass","Sex"]).summary().show()
df_titanic.printSchema()
```

Código 22: Solución - Agrupando columnas

4.1.4. Definiendo funciones UDF (User Defined Functions) y aplicándolas sobre una columna concreta

```
@F.udf
def lower(x):
    return x.lower() if isinstance(x,str) else x

df_titanic_lowered = df_titanic.withColumn(
    "Embarked",
    lower(col("Embarked"))
)
df_titanic_lowered.show(10)
```

Código 23: Solución - Definiendo funciones UDF y aplicándolas sobre una columna

4.1.5. Ejercicio final

```
df_titanic_survivors = df_titanic.where(col('Survived') == '1')

df_titanic_men = df_titanic_survivors.where(col('Sex') == 'male')
df_titanic_women = df_titanic_survivors.where(col('Sex') == 'female')

df_titanic_men = df_titanic_men
        .groupby("Pclass")
        .agg(F.count("Pclass")
        .alias("Number of survivors"))
        .orderBy("Pclass",ascending=True)

df_titanic_women = df_titanic_women
        .groupby("Pclass")
        .agg(F.count("Pclass")
        .alias("Number of survivors"))
        .orderBy("Pclass",ascending=True)

df_titanic_men.show()
df_titanic_women.show()
```

Código 24: Solución - Ejercicio final DataFrame

4.2. RDD

4.2.1. Filtrando

```
rdd_even = rdd.filter(lambda x: not (x & 1))
rdd_even.take(15)
```

Código 25: Solución - Filtrando RDD

4.2.2. Aplicando funciones a todos los elementos

```
rdd_squared = rdd.map(lambda x: x*x)
rdd_squared.take(8)
```

Código 26: Solución - Aplicando funciones a todos los elementos

4.2.3. Obteniendo una parte aleatoria del RDD

```
rdd_fraction = rdd.sample(withReplacement=False,fraction=0.05)
rdd_fraction.take(5)
```

Código 27: Solución - Obteniendo una parte aleatoria del RDD

4.2.4. Reduciendo por clave

```
rdd_groupby_remainder_of_eleven = [(x % 11, x) for x in range(100,201)]
rdd_groupby_remainder_of_eleven = sc.parallelize(
    rdd_groupby_remainder_of_eleven
)
rdd_groupby_mult = rdd_groupby_remainder_of_eleven.reduceByKey(
    lambda accu, value: accu*value
)
[element for element in rdd_groupby_mult.collect()]
```

Código 28: Solución - Reduciendo por clave

4.2.5. Reduciendo el RDD a un único valor

```
sum_value = rdd.fold(0,
    lambda accu, value: accu+value
)
sum_value
```

Código 29: Solución - Reduciendo el RDD a un único valor

4.2.6. Ejercicio final

```
data = sc.parallelize(data)
data = data.filter(lambda x: x[0][0] <= x[0][1])
sum_data = data.map(lambda x: x[1]).fold(0,lambda accu, value: accu + value)
data = data.map(lambda x: (x[0], x[1] / sum_data))
data = data.reduceByKey(lambda accu, value: accu + value)

data.take(10)
```

Código 30: Solución - Ejercicio final RDD