

**BIG DATA TOOLS AND TECHNIQUES**

**STUDENT ID: @00713171**

**MSC DATA SCIENCE**

**School of Computing, Science & Engineering**

**University of Salford, Manchester.**

**AIRIOHUODION CLEMENT**

**APRIL 02, 2024**

## **TASK ONE**

## **ABSTRACT**

The goal of this job is to get a thorough grasp of large data systems through the exploration of particular subjects that highlight the interaction between theory and real-world applications. It entails employing well-liked big data tools and methodologies in addition to having competence with common data analysis tasks including loading, cleaning, analyzing, and producing reports. It is essential to be proficient with Python, SQL, or Linux terminal commands in order to pass this test.

## **INTRODUCTION**

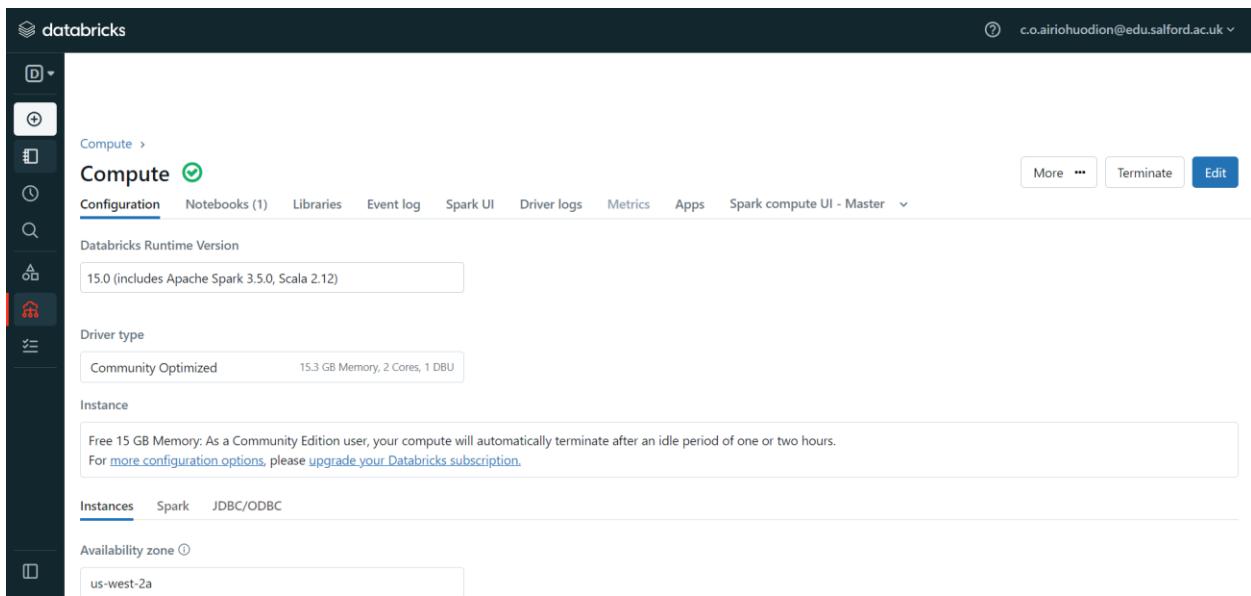
In my capacity as an AI engineer and data scientist given the opportunity to investigate more about clinical trials, I've carefully examined the dataset that is given me. Several important topics are intended to be addressed by this investigation, which skillfully uses visuals to bolster findings.

First, I'll look at the total number of studies in the dataset, making sure to explicitly account for different research. The types of research that are here will next be discussed, with each type listed with its corresponding frequency in order of least to most frequent. I'll also include the top 5 conditions found in the dataset along with their corresponding frequencies.

I'll also look for sponsors, paying special attention to non-pharmaceutical businesses. I'll count the number of clinical studies that the top 10 sponsors outside the pharmaceutical industry have supported. Etc.

# SETUP

## Fire up the Databricks workspace

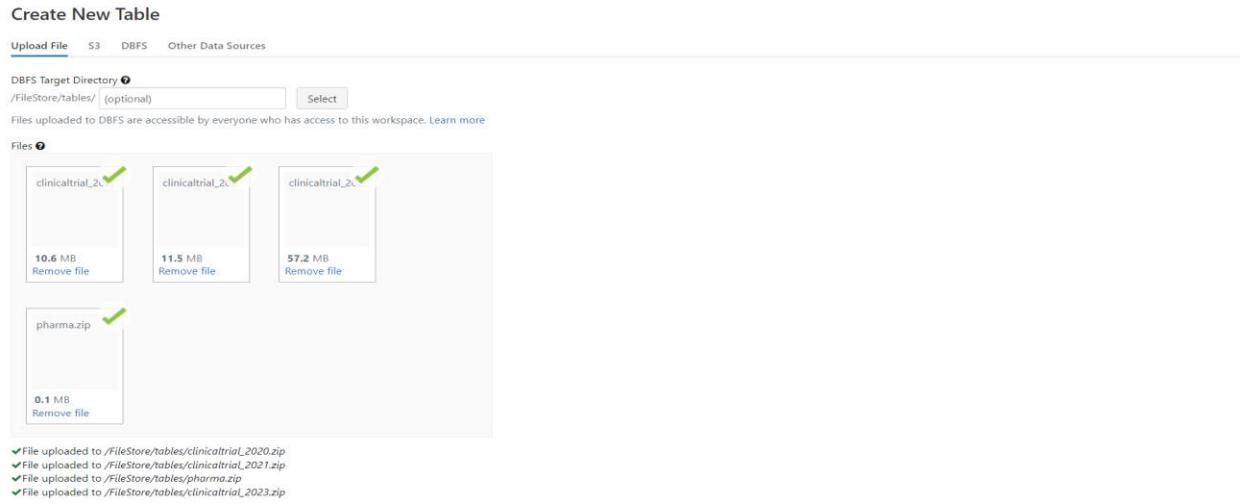


**Figure 1.1.0** Setting up the platform required to kick start the task given

After logging into my Databricks Community Edition account, I created a cluster to start my analysis to have access to a machine to use then I click on create Compute and type a new name for the cluster in this case **Clement\_Airiohuodion\_rdd** for my RDD solution. I picked the most recent run time 15.0, after the green tick meaning successfully started up.

## IMPORTING ZIP FILES

(including descriptions, justifications and screenshots of all code).



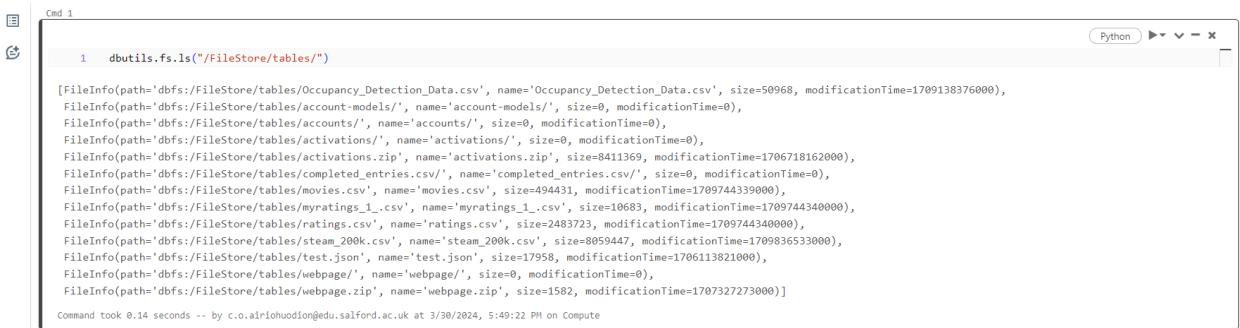
The screenshot shows the 'Create New Table' interface with the 'Upload File' tab selected. A 'DBFS Target Directory' field contains '/fileStore/tables/'. Below it, a note says 'Files uploaded to DBFS are accessible by everyone who has access to this workspace. Learn more'. A 'Select' button is next to the target directory field. A 'Files' section lists three CSV files: 'clinicaltrial\_2020.csv' (10.6 MB), 'clinicaltrial\_2021.csv' (11.5 MB), and 'clinicaltrial\_2023.csv' (57.2 MB), each with a 'Remove file' link. Below them is a 'pharma.zip' file (0.1 MB) with a green checkmark and a 'Remove file' link. At the bottom, a log shows four successful file uploads:

```
✓ File uploaded to /FileStore/tables/clinicaltrial_2020.zip
✓ File uploaded to /FileStore/tables/clinicaltrial_2021.zip
✓ File uploaded to /FileStore/tables/pharma.zip
✓ File uploaded to /FileStore/tables/clinicaltrial_2023.zip
```

**Figure 1.1.1** The screen short above shows uploaded clinical.csv of year 2020, 2021, 2023 files and a pharma file.

## FILE UNZIPPING PROCESS

I created a python notebook to have access to runnable cells under the active cluster. Given clinical\_trial 2020, 2021,2023 and pharma file we have been instructed to answer some questions. Further steps were required to unzip the 4 files after they were uploaded to the FileStore/tables directory, as shown in the Figures below.



A Jupyter Notebook cell titled 'Cmd 1' containing Python code. The code is:

```
dbutils.fs.ls("/FileStore/tables/")
```

The output shows a list of FileInfo objects for various files and folders in the '/FileStore/tables/' directory. The list includes:

- [FileInfo(path='dbfs:/FileStore/tables/Occupancy\_Detection\_Data.csv', name='Occupancy\_Detection\_Data.csv', size=50968, modificationTime=1709138376000),
- FileInfo(path='dbfs:/FileStore/tables/account-models/', name='account-models/', size=0, modificationTime=0),
- FileInfo(path='dbfs:/FileStore/tables/accounts/', name='accounts/', size=0, modificationTime=0),
- FileInfo(path='dbfs:/FileStore/tables/activations/', name='activations/', size=0, modificationTime=0),
- FileInfo(path='dbfs:/FileStore/tables/activations.zip', name='activations.zip', size=8411369, modificationTime=1706718162000),
- FileInfo(path='dbfs:/FileStore/tables/completed\_entries.csv', name='completed\_entries.csv', size=0, modificationTime=0),
- FileInfo(path='dbfs:/FileStore/tables/movies.csv', name='movies.csv', size=494431, modificationTime=1709744339000),
- FileInfo(path='dbfs:/FileStore/tables/myratings\_1\_.csv', name='myratings\_1\_.csv', size=10683, modificationTime=1709744340000),
- FileInfo(path='dbfs:/FileStore/tables/ratings.csv', name='ratings.csv', size=2483723, modificationTime=1709744340000),
- FileInfo(path='dbfs:/FileStore/tables/steam\_200k.csv', name='steam\_200k.csv', size=8059447, modificationTime=1709836533000),
- FileInfo(path='dbfs:/FileStore/tables/test.json', name='test.json', size=17958, modificationTime=1706113821000),
- FileInfo(path='dbfs:/FileStore/tables/webpage/', name='webpage/', size=0, modificationTime=0),
- FileInfo(path='dbfs:/FileStore/tables/webpage.zip', name='webpage.zip', size=1582, modificationTime=1707327273000)]

Command took 0.14 seconds -> by c.o.aliohuddion@edu.salford.ac.uk at 3/30/2024, 5:49:22 PM on Compute

**Figure 1.1.2** Using the dbutils.fs.ls() command, generally listing the files and folders in the “tables” directory inside the “FileStore” directory.

```

Cmd 2
Python > v - x

1 # Function to check if file exists in the file system
2 def check_zip_file(fileroot):
3     try:
4         db = dbutils.fs.ls(f"/FileStore/tables/{fileroot}.zip")
5         return True
6     except Exception:
7         return False
8
9 fileroot_clinical = "clinictrial_2023"
10 fileroot_pharma = "pharma"
11
12 fileroot_clinical_zip_file_exists = check_zip_file(fileroot_clinical)
13 fileroot_pharma_zip_file_exists = check_zip_file(fileroot_pharma)
14
15 print("fileroot_clinical Zip File Exists:", fileroot_clinical_zip_file_exists)
16 print("fileroot_pharma Zip File Exists:", fileroot_pharma_zip_file_exists)

fileroot_clinical Zip File Exists: True
fileroot_pharma Zip File Exists: True

Command took 0.13 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 6:13:06 PM on Compute

```

**Figure 1.1.3** Creating and Using a function to check specific files (clinictrial\_2023 and pharma file) respectively.

```

Cmd 3
Python > v - x

1 def copy_into_localfile(fileroot):
2     try:
3         dbutils.fs.cp(f"/FileStore/tables/{fileroot}.zip", "file:/tmp/")
4         dbutils.fs.ls(f"/FileStore/tables/{fileroot}.zip")
5     except Exception:
6         pass
7
8 if fileroot_clinical_zip_file_exists:
9     copy_into_localfile(fileroot_clinical)
10 if fileroot_pharma_zip_file_exists:
11     copy_into_localfile(fileroot_pharma)

Command took 1.36 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 6:18:11 PM on Compute

```

**Figure 1.1.4** This defined a function called **copy\_into\_localfile()** in the supplied code, with **dbutils.fs.cp** it copies the two ZIP files from the **FileStore** directory to the **local file system** directory **/temp/**. It then displays the contents of the transferred ZIP file if the process is successful. Based on the existence of each file which was previously established, the function is called twice: once for the fileroot\_clinical ZIP file and again for the fileroot\_pharm ZIP file.

```

Cmd 4
Python > v - x

1 import os
2
3 os.environ['fileroot_clinical'] = fileroot_clinical
4 os.environ['fileroot_pharma'] = fileroot_pharma

Command took 0.10 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 6:19:55 PM on Compute

```

**Figure 1.1.5** Knowing that the local filesystem or shell command line does not automatically know about the “**fileroot**” variable, we set the environment variables “**fileroot\_clinical**” and “**fileroot\_pharma**” to the variables. We set this value as an environment variable so that it maybe accessed in shell commands.

```
Cmd 5
1 %sh
2
3 ls /tmp/$fileroot_clinical.zip
4 ls /tmp/$fileroot_pharma.zip|
```

/tmp/clinicaltrial\_2023.zip  
/tmp/pharma.zip

Command took 0.10 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 6:21:17 PM on Compute

**Figure 1.1.6** This “ls” shell command above checks the /tmp/ to confirm that **clinical\_trail.zip** and **pharma.zip** was moved successfully.

```
Cmd 6
1 %sh
2
3 unzip -d /tmp /tmp/$fileroot_clinical.zip
4 unzip -d /tmp /tmp/$fileroot_pharma.zip|
```

Archive: /tmp/clinicaltrial\_2023.zip  
inflating: /tmp/clinicaltrial\_2023.csv  
Archive: /tmp/pharma.zip  
inflating: /tmp/pharma.csv

Command took 2.42 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 6:23:34 PM on Compute

**Figure 1.1.7** This code above Unzips the two .zip files

The Files designated by the variables \$fileroot\_clinical and \$fileroot\_pharma is unzipped into the /tmp/ directory, and the extracted files and their corresponding archives are then shown.

```
Cmd 7
1 %sh
2
3 ls /tmp/$fileroot_clinical.csv
4 ls /tmp/$fileroot_pharma.csv|
```

/tmp/clinicaltrial\_2023.csv  
/tmp/pharma.csv

Command took 0.04 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 6:25:52 PM on Compute

**Figure 1.1.8** This “ls” shell command above checks the /tmp/ directory to confirm that **clinical\_trail.csv** and **pharma.csv** now exist to ensure successful unzipping.

```
Cmd 8
1 def mv_file_into_dbfs(fileroot):
2     try:
3         dbutils.fs.mv(f"file:/tmp/{fileroot}.csv", f"/FileStore/tables/{fileroot}.csv")
4         print(f"File '{fileroot}.csv' moved to DBFS successfully.")
5     except Exception:
6         pass
7
8 mv_file_into_dbfs(fileroot_clinical)
9 mv_file_into_dbfs(fileroot_pharma)|
```

File 'clinicaltrial\_2023.csv' moved to DBFS successfully.  
File 'pharma.csv' moved to DBFS successfully.

Command took 17.62 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 6:28:39 PM on Compute

**Figure 1.1.9** Both files from the local filesystem, namely from the /tmp directory, are moved to the (DBFS) File System located under the /FileStore/tables directory using the

given method `mv_file_into_dbfs()`. The function is run twice to transfer the files that correspond to the `fileroot_pharma` and `fileroot_clinical` variables after it has been defined. The result attests to the two files successful transfer to DBFS.

```
1 def remove_f_dir(fileroot):
2     try:
3         dbutils.fs.rm(f"/FileStore/tables/{fileroot}.zip")
4     except Exception:
5         pass
6
7 remove_f_dir(fileroot_clinical)
8 remove_f_dir(fileroot_pharma)
```

**Figure 1.2.0** The function `remove_f_dir()` tries to remove the zip files from the DBFS that have been unzipped earlier. It tries to delete the file indicated by `FileStore/tables/{fileroot}.zip` using the `dbutils.fs.rm()` function.

**Figure 1.2.1** The code reads the CSV file from the /FileStore/tables/ directory, revealing the first few lines. The fileroot\_clinical variables value determines the filename.

## DATA CLEANING AND PREPARATION

### RDD IMPLEMENTATION



```
Cmd 11
1 #creating 1st RDD for fileroot parameter
2 def create_rdd(fileroot):
3     myrddc = sc.textfile("/FileStore/tables/" + fileroot + ".csv")
4     return myrddc
Python ▶ v - x
```

Command took 0.09 seconds -- by c.o.airiohudson@edu.salford.ac.uk at 3/30/2024, 6:40:04 PM on Compute

**Figure 1.2.2** At this point the CSV files in use in the DBFS is used to construct an RDD(Resilient distributed Dataset) using the “**create\_rdd()**” function. It accepts the given parameter called fileroot, which indicates the target CSV files root name. The function uses **sc.textFilfe()** method with a filename that is obtained from the fileroot parameter to read the contents of the CSV file that is located in the file store table directory. The function then returns the final RDD. Using CSV files, this method makes it easier to create RDDs for further data processing tasks.



```
Cmd 12
1 #split record in RDD based on corresponding delimiter character
2 delimiter_selector = {
3     "clinicaltrial_2023": "\t",
4     "clinicaltrial_2021": "|",
5     "clinicaltrial_2020": "|",
6     "pharma": ","
7 }
8
9 # creating 2nd RDD
10 def clean_rdd(myrddc, fileroot):
11     Cleanup_myrddc = myrddc.map(lambda x: [i.replace(',', '').replace('\"', '') for i in x.split(delimiter_selector[fileroot])])
12     return Cleanup_myrddc
Python ▶ v - x
```

Command took 0.09 seconds -- by c.o.airiohudson@edu.salford.ac.uk at 4/12/2024, 8:01:35 AM on sgst

**Figure 1.2.3** The code defines a dictionary called **delimiter\_selector**, linking file origins to appropriate delimiter characters. The function called **clean\_rdd()** accept two parameters a file root (fileroot) and an RDD(**myrddc**) and converts the RDD to a file root, using the **map()** function to separate records based on the delimiter character, simplifying RDD cleaning.

```

Cmd 13
Python ▶ v - x

1 # Creating 3rd RDDs
2 ClinicalT_RDD = create_rdd(fileroot_clinical)
3 Pharma_RDD = create_rdd(fileroot_pharma)
4
5 # Clean up RDDs
6 ClinicalT_RDD_clean = clean_rdd(ClinicalT_RDD, fileroot_clinical)
7 Pharma_RDD_clean = clean_rdd(Pharma_RDD, fileroot_pharma)
8
9 # Take first 5 elements from cleaned ClinicalT RDD
10 ClinicalT_RDD_clean.take(5)

> (1) Spark Jobs
[[{"id": "NCT0303471", "Study Title": "Effectiveness of a Problem-solving Intervention for Common Adolescent Mental Health Problems in India", "Acronym": "PRIDE", "Status": "COMPLETED", "Conditions": "Mental Health Issue (E.g. Depression Psychosis Personality Disorder Substance Abuse)", "Interventions": "BEHAVIORAL: PRIDE 'Step 1' problem-solving intervention|BEHAVIORAL: Enhanced usual care", "Sponsor": "Sangath", "Collaborators": "", "Enrolment": "", "Funder Type": "", "Type": "Study Design", "Start": "", "Completion": ""}], Command took 7.73 seconds -- by c.oairiohudson@edu.salford.ac.uk at 4/12/2024, 8:01:36 AM on agst

```

**Figure 1.2.4** The `create_rdd()` function creates two RDDs, `ClinicalT_RDD` and `Pharma_RDD`, representing raw data from pharma files and clinical trials. The `clean_rdd()` method cleans the raw RDDs, dividing records within each RDD. The `take(5)` action extracts the first items from the cleaned `ClinicalT_RDD`, ensuring efficient data management

## DATAFRAME IMPLEMENTATION

In this section I have used Spark DataFrames to process the clinical trial and pharma files. Various cleaning process was carried out using unique codes, for example the completion column in the clinical trial 2023 had a double quote ‘ “ ’ and lots of commas that needed special code to clean it off. After the cleaning process has been thoroughly done, questions are given are then answered.

```

Cmd 1
Python ▶ v - x

1 fileroot_clinical = "clinicaltrial_2023"
2 fileroot_pharma = "pharma"

```

**Figure 1.2.5** My file roots are `file_pharma` and `fileroot_clinical`. The names of the CSV files containing pharmaceutical and clinical trial 2020, 2021 and 2023 data respectively, are held in these variables

```

Code 2 Python

1 #Creating DataFrame
2 from pyspark.sql.types import * #StructType, StructField
3 from pyspark.sql.functions import *
4
5 delimiter_selector = {
6     "clinicaltrial_2023": "\t",
7     "clinicaltrial_2021": "|",
8     "clinicaltrial_2020": "|",
9     "pharma": ","
10 }
11
12 def create_clinical_dataframe(fileroot):
13     if fileroot == "clinicaltrial_2023":
14         rdd = sc.textfile(f"/FileStore/tables/{fileroot}.csv").map(lambda row: row.split(delimiter_selector[fileroot]))
15         head = rdd.first()
16         rdd = rdd.map(lambda row: row + [" " for i in range(len(head) - len(row)) if len(row) < len(head) else row])
17         df = rdd.toDF()
18         first = df.first()
19         for col in range(0, len(list(first))):
20             df = df.withColumnRenamed(f"_{(col + 1)}", list(first)[col].replace('"', "").replace("'", ""))
21             df = df.withColumn('Index', monotonically_increasing_id())
22             df = df.withColumn('Completion', regexp_replace("Completion", "\"", "")).withColumn('Completion', regexp_replace("Completion", "'", ""))
23             return df.filter(~df.index.isin([0])).drop('Index')
24         else:
25             return spark.read.csv(f"/FileStore/tables/{fileroot}.csv", sep=delimiter_selector[fileroot], header=True)
26
27 ClinicalT_dataframe = create_clinical_dataframe(fileroot_clinical)
28 pharma_dataframe = create_clinical_dataframe(fileroot_pharma)
29 ClinicalT_dataframe.show(20)
30

```

**Figure 1.2.6** The code imports Modules to define schema and functions for DataFrame operations. It introduces a **delimiter\_selector** dictionary and maps file root names to their delimiters. The function requires a fileroot as input to produce a DataFrame corresponding to the file root. The code reads the Clinical\_trial 2023 file, divides each row by its delimiter, generates an RDD, and transforms it into a DataFrame. It removes special characters, changes column names, removes the DataFrame header row, adds an index column, replaces double quotes and commas in the completion column, and returns the cleaned DataFrame.

The clinical\_trial CSV files 2020 and 2021 are read using the spark read.csv function, and a DataFrame is created for the **clinicaltrial** and **pharma** data using the **Create\_clinical\_dataframe** method, showing 20 rows.

**Figure 1.2.7** This above is the sample of the cleaned clinical\_trial 2023

## SQL PySpark IMPLEMENTATION



```
1 %python
2
3 fileroot_clinical = "clinicaltrial_2023"
4 fileroot_pharma = "pharma"
5
```

Command took 0.08 seconds -- by c.o.ai.onuodion@edu.salford.ac.uk at 4/12/2024, 8:12:01 AM on sgst

**Figure 1.2.8** My file roots are file\_pharma and fileroot\_clinical. The names of the CSV files containing pharmaceutical and clinical trial 2020, 2021 and 2023 data respectively, are held in these variables



```
1 %python
2
3 #Creating DataFrame
4 from pyspark.sql.types import * #StructType, StructField
5 from pyspark.sql.functions import *
6
7 delimiter_selector = {
8     "clinicaltrial_2023": "\t",
9     "clinicaltrial_2021": "|",
10    "clinicaltrial_2020": "|",
11    "pharma": ","
12 }
13
14 def create_clinical_dataframe(fileroot):
15     if fileroot == "clinicaltrial_2023":
16         rdd = sc.textFile(f"/FileStore/tables/{fileroot}.csv").map(lambda row: row.split(delimiter_selector[fileroot]))
17         head = rdd.first()
18         rdd = rdd.map(lambda row: row + [" " for i in range(len(head) - len(row)) if len(row) < len(head) else row])
19         df = rdd.toDF()
20         first = df.first()
21         for col in range(0, len(list(first))):
22             df = df.withColumnRenamed(f"_{col + 1}", list(first)[col].replace('"', "").replace("'", ""))
23             df = df.withColumn("index", monotonically_increasing_id())
24             df = df.withColumn('Completion', regexp_replace("Completion", "\"", "")).withColumn('Completion', regexp_replace("Completion", "'", ""))
25             return df.filter(~df.index.isin([0])).drop('index')
26     else:
27         return spark.read.csv(f"/FileStore/tables/{fileroot}.csv", sep=delimiter_selector[fileroot], header=True)
28
29 Clinical_dataframe = create_clinical_dataframe(fileroot_clinical)
30 pharma_dataframe = create_clinical_dataframe(fileroot_pharma)
31 Clinical_dataframe.show(20)
```

**Figure 1.2.9** The code imports Modules to define schema and functions for DataFrame operations. It introduces a **delimiter\_selector** dictionary and maps file root names to their delimiters. The function requires a fileroot as input to produce a DataFrame corresponding to the file root. The code reads the Clinical\_trial 2023 file, divides each row by its delimiter, generates an RDD, and transforms it into a DataFrame. It removes special characters, changes column names, removes the DataFrame header row, adds an index column, replaces double quotes and commas in the completion column, and returns the cleaned DataFrame.

The clinical\_trial CSV files 2020 and 2021 are read using the spark read.csv function, and a DataFrame is created for the **clinicaltrial** and **pharma** data using the **Create\_clinical\_dataframe** method, showing 20 rows.

```

Cmd 3
1 %python
2
3 pharma_dataframe.createOrReplaceTempView(fileroot_pharma)
4 ClinicalT_dataframe.createOrReplaceTempView(fileroot_clinical)

Command took 0.22 seconds -- by c.o.miriohuadion@edu.salford.ac.uk at 4/12/2024, 8:12:01 AM on sgst

```

**Figure 1.3.0** This code creates two temporary table (view) Pharma\_dataframe and ClinicalT\_dataframe. I used **createOrReplaceTempView** method.

Cmd 4

```

1 SELECT * FROM clinicaltrial_2023

```

(1) Spark Jobs

Id	Study Title	Acronym	Status	Conditions
1 "NCT03630471	Effectiveness of a Problem-solving Intervention for Common Adolescent Mental Health Problems in India	PRIDE	COMPLETED	Mental Health Issue (E.G., Depression, Psycho
2 "NCT05992571	Oral Ketone Monoester Supplementation and Resting-state Brain Connectivity		RECRUITING	Cerebrovascular Function/Cognition
3 "NCT00237471	Impact of Tight Glycaemic Control in Acute Myocardial Infarction		TERMINATED	Myocardial Infarct Hyperglycemia
4 "NCT03820271	New Prognostic Predictive Models of Mortality of Decompensated Cirrhotic Patients Waiting for Liver Transplantation	SUPERMELD	RECRUITING	Decompensated Cirrhosis Liver Transplantatio
5 "NCT06229171	InTake Care: Development and Validation of an Innovative "Personalized Digital Health Solution for Medication Adherence Support in Cardiovascular Prevention	InTakeCare	NOT_YET_RECRUITING	Hypertension Treatment Adherence and Comp
6 "NCT02945371	Tailored Inhibitory Control Training to Reverse EA-linked Deficits in Mid-life	REV	COMPLETED	Smoking Alcohol Drinking Prescription Drug A
7 "NCT01055171	Neuromodulation of Trauma Memories in PTSD & Alcohol Dependence		COMPLETED	Alcohol Dependence PTSD

4,839 rows | Truncated data | 3.67 seconds runtime

Refreshed 9 minutes ago

Command took 3.67 seconds -- by c.o.miriohuadion@edu.salford.ac.uk at 4/12/2024, 8:12:01 AM on sgst

**Figure 1.3.1** The complete dataset is returned by this query, giving users a thorough overview of the clinical trial data for 2023. A single clinical trial is represented by each row in the result, with columns denoting different characteristics such as trial ID, conditions, sponsors.

Cmd 5

```

1 SELECT * FROM pharma

```

(1) Spark Jobs

Company	Parent_Company	Penalty_Amount	Subtraction_From_Penalty	Penalty_Amount_Adjusted
1 Abbott Laboratories	Abbott Laboratories	\$5,475,000	\$0	\$5,475,000
2 Abbott Laboratories Inc.	AbbVie	\$1,500,000,000	\$0	\$1,500,000,000
3 Abbott Laboratories Inc.	AbbVie	\$126,500,000	\$0	\$126,500,000
4 Abbott Laboratories Puerto Rico, Inc.	Abbott Laboratories	\$49,045	\$0	\$49,045
5 Acclarient Inc.	Johnson & Johnson	\$18,000,000	\$0	\$18,000,000

968 rows | 1.67 seconds runtime

Refreshed 10 minutes ago

Command took 1.67 seconds -- by c.o.miriohuadion@edu.salford.ac.uk at 4/12/2024, 8:12:01 AM on sgst

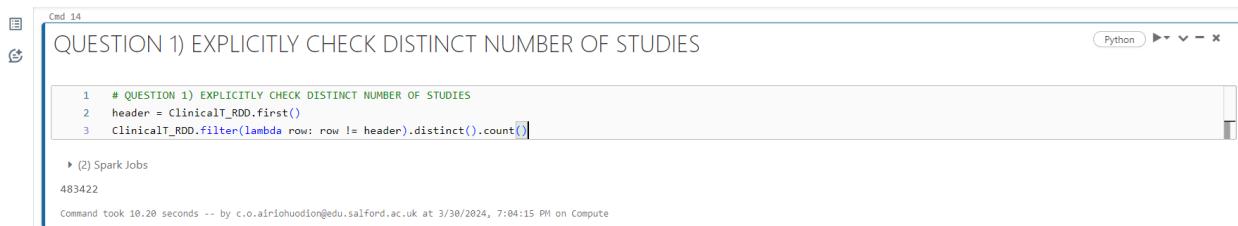
**Figure 1.3.2** Every field and record in the pharma table will be retrieved by this query

## PROBLEM ANSWERS

### SOLUTION 1

ASSUMPTIONS: The aim is to explicitly validate the unique number of studies in the dataset, assuming the csv file is cleaned, consistent and free of error.

#### RDD Implementation



```
QUESTION 1) EXPLICITLY CHECK DISTINCT NUMBER OF STUDIES
```

```
1 # QUESTION 1) EXPLICITLY CHECK DISTINCT NUMBER OF STUDIES
2 header = ClinicalT_RDD.first()
3 ClinicalT_RDD.filter(lambda row: row != header).distinct().count()
```

(2) Spark Jobs  
483422

Command took 10.20 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 7:04:15 PM on Compute

**Figure 1.3.3** This code retrieves the first row of the data from ClinicalT\_RDD and assigns it to the variable header. It filters the RDD ClinicalT\_RDD by comparing each row to the header row using the lambda function. The distinct() function is applied to the filtered RDD to count distinct rows by removing the header row and deleting duplicates, then counting the remaining rows.

#### DATAFRAME Implementation



```
QUESTION 1) EXPLICITLY CHECK DISTINCT NUMBER OF STUDIES
```

```
1 # QUEST|1
2
3 ClinicalT_dataframe.distinct().count()
```

(3) Spark Jobs  
483422

Command took 17.27 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/9/2024, 7:21:54 PM on cl

**Figure 1.3.4** From the question given I counted the unique entries in the ClinicalT\_dataframe using pyspark. By taking into account all columns, this count indicates the total number of unique rows in the DataFrame. Using ‘**Dinstinct**’ to get unique entries and then ‘**count**’ them.

## SQL Implementation



The screenshot shows a command-line interface for a database system. The command entered is:

```
1. SELECT DISTINCT count(*) FROM clinicaltrial_2023
```

The results are displayed in a table:

count(1)
1 483422

Below the table, it says "1 row | 12.38 seconds runtime". In the bottom right corner, it says "Refreshed 10 minutes ago".

**Figure 1.3.5** The number of unique records in the clinicaltrial\_2023 table will be counted by this query, which will return the result in a distinct\_count column.

## RESULT DISCUSSION:

The collection contains 483,422 different studies, according to the analysis, each study is counted once.

## SOLUTION 2

**ASSUMPTIONS:** The “Type” column in the dataset provides trustworthy labels for each study type, and the analysis is predicated on the dataset having accurate and consistent information on clinical trials.

## RDD Implementation



The screenshot shows a command-line interface for a Python-based distributed computing environment. The code entered is:

```
2. QUESTION 2) filter, map, reduce, and sort a cleaned RDD then collect
```

```
1 # QUESTION 2
2 typ_idx = ClinicalT_RDD_clean.first().index('Type')
3 ClinicalT_RDD_clean.filter(lambda x: len(x) > typ_idx + 1) \
4     .map(lambda x: (x[typ_idx], 1)) \
5     .filter(lambda row: row[0] != 'Type') \
6     .reduceByKey(lambda a, b: a + b) \
7     .filter(lambda x: x[0] != '') \
8     .sortBy(lambda x: x[1], ascending=False) \
9     .collect()
```

The output is a list of tuples:

```
[('INTERVENTIONAL', 371382),
 ('OBSERVATIONAL', 118221),
 ('EXPANDED_ACCESS', 928)]
```

Below the list, it says "Command took 10.37 seconds -- by c.o.airiuhuodion@edu.salford.ac.uk at 4/1/2024, 12:54:35 AM on clust".

**Figure 1.3.6** Using a lambda function to check for inequality with the header row, it filters out the header row from the RDD ClinicalT\_RDD, which is represented by the variable header. After deleting duplicates, it uses the distinct function to procedure unique rows. Lastly, it counts the number of distinct rows in the dataset, which is a direct correlation to the number of distinct studies.

## DATAFRAME Implementation

The screenshot shows a Jupyter Notebook cell titled "Cmd 4" with the following content:

```
2. list all the types, freq., desc

1 #QUESTION 2
2
3 ClinicalT_dataframe.groupBy('Type').count().orderBy('count', ascending=False).show(3)
```

Output:

```
+-----+-----+
|      Type| count|
+-----+-----+
| INTERVENTIONAL| 371382|
| OBSERVATIONAL| 110221|
| EXPANDED_ACCESS| 928|
+-----+-----+
only showing top 3 rows
```

Command took 11.36 seconds -- by c.o.eiriohudson@edu.salford.ac.uk at 4/9/2024, 7:21:54 PM on cl

**Figure 1.3.7** The pyspark above groups the entire row in the DataFrame ClinicalT\_dataframe by the “Type” column using **.groupBy** then after it uses “**.count**” function to count the number of rows in each group, to further achieve the result I used the **.orderBy**” to place the data in descending order by count, then finally show three (3), results.

## SQL Implementation

The screenshot shows a Jupyter Notebook cell titled "Cmd 7" with the following content:

```
2. list all the types, freq., desc

1 SELECT clinicaltrial_2023.Type, count(*) as count FROM clinicaltrial_2023
2 GROUP BY clinicaltrial_2023.Type
3 ORDER BY count DESC
4 LIMIT 3
```

Output:

Type	count
INTERVENTIONAL	371382
OBSERVATIONAL	110221
EXPANDED_ACCESS	928

3 rows | 16.10 seconds runtime

Refreshed 11 minutes ago

Command took 16.10 seconds -- by c.o.eiriohudson@edu.salford.ac.uk at 4/12/2024, 8:12:01 AM on sgst

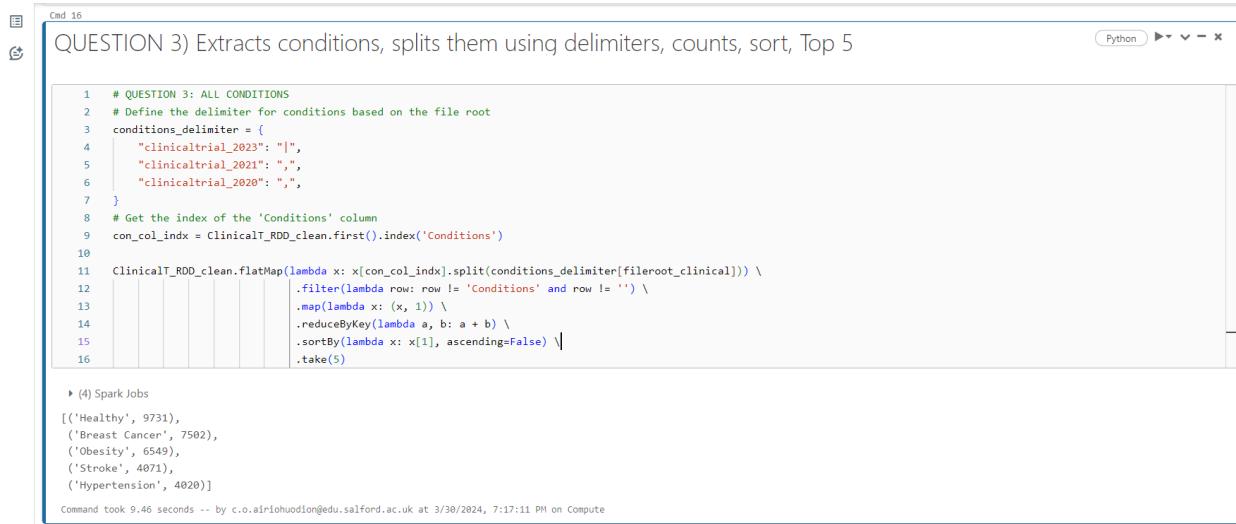
**Figure 1.3.8** The clinicaltrial 2023 tables Type column is selected by the SQL query, which then groups the records according to the Type columns values. Using the COUNT(\*) function, it counts the number of records within each group and aliases the result as count. Following that, the findings are sorted by count in descending order, and only the top three categories of clinical trials with the highest counts are included.

Discussion of Result: The dataset consists of observational, expanded access, and interventional studies, providing crucial information on the composition and areas of interest in clinical research.

## SOLUTION 3

ASSUMPTION: The “Conditions” column contains a list of case-sensitive conditions, divided by a delimiter depending on the particular clinical csv dataset, and is assumed to be accurately reported without any formatting issues.

### RDD Implementation



The screenshot shows a Jupyter Notebook cell titled "QUESTION 3) Extracts conditions, splits them using delimiters, counts, sort, Top 5". The code uses flatMap, filter, map, reduceByKey, sortBy, and take(5) methods to process an RDD. It includes a dictionary for file roots and a list of tuples for the top 5 conditions.

```
1 # QUESTION 3: ALL CONDITIONS
2 # Define the delimiter for conditions based on the file root
3 conditions_delimiter = {
4     "clinicaltrial_2023": "|",
5     "clinicaltrial_2021": ",",
6     "clinicaltrial_2020": ","
7 }
8 # Get the index of the 'Conditions' column
9 con_col_idx = ClinicalT_RDD_clean.first().index('Conditions')
10
11 ClinicalT_RDD_clean.flatMap(lambda x: x[con_col_idx].split(conditions_delimiter[fileroot_clinical])) \
12     .filter(lambda row: row != 'Conditions' and row != '') \
13     .map(lambda x: (x, 1)) \
14     .reduceByKey(lambda a, b: a + b) \
15     .sortBy(lambda x: x[1], ascending=False) \
16     .take(5)

▶ (4) Spark Jobs
[('Healthy', 9731),
 ('Breast Cancer', 7502),
 ('Obesity', 6549),
 ('Stroke', 4071),
 ('Hypertension', 4020)]
```

Command took 9.46 seconds -- by c.o.airiohudion@edu.salford.ac.uk at 3/30/2024, 7:17:11 PM on Compute

**Figure 1.3.9** A dictionary called `conditions_delimiter` is created to assign distinct delimiters to file roots, indicating file root based division for the “Conditions” column. The index is obtained from the cleaned RDD `ClinicalT_RDD_clean`. RDD processing for Counting Conditions uses `flatMap()`, `filter()`, `Map()`, `reduceByKey()`, `sortBy`, and `Take(5)` methods. `flatMap` split the ‘Conditions’ column for each row, `filter` removes empty rows, `Map` maps a tuple to each condition, and takes the five most prevalent conditions.

## DATAFRAME Implementation



```

Cmd 5
3. Top 5 cond. with their freq.

1 # QUESTION 3
2
3 conditions_delimiter = [
4 "clinicaltrial_2023": "\\",
5 "clinicaltrial_2021": ",",
6 "clinicaltrial_2020": ","
7 ]
8
9 ClinicalTrial_dataframe.withColumn('Conditions', explode(split(col("Conditions"), conditions_delimiter[fileRoot_clinical]))).groupBy('Conditions').count().orderBy('count', ascending=False).filter("Conditions != ''").show(5, truncate=False)

(2) Spark Jobs
+-----+-----+
|Conditions |count|
+-----+-----+
|Healthy |9731 |
|Breast Cancer|7502 |
|Obesity |6549 |
|Stroke |4071 |
|Hypertension |4020 |
+-----+-----+
only showing top 5 rows

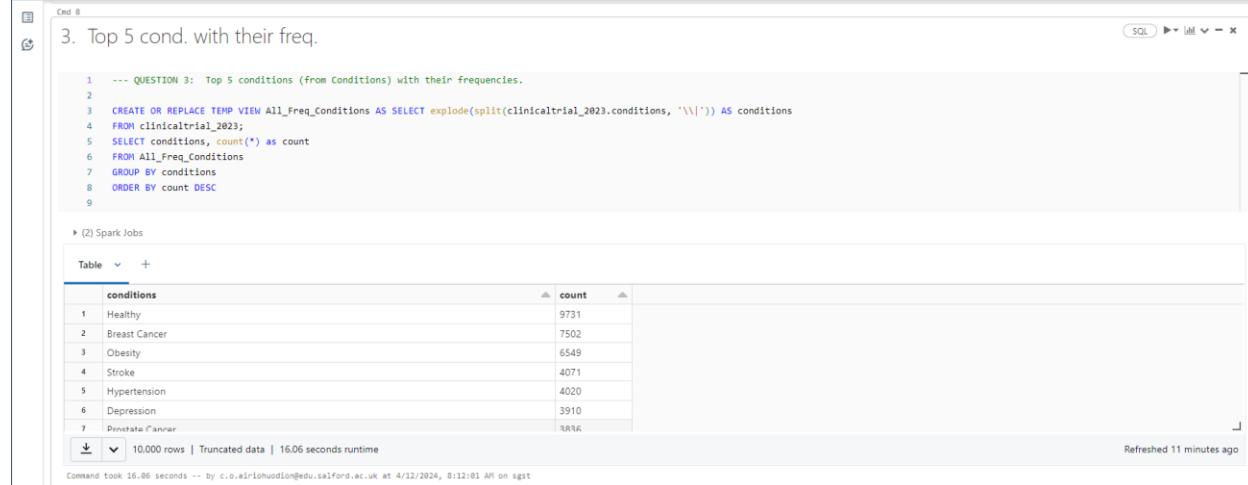
Command took 12.84 seconds -- by c.o.airohuddion@edu.salford.ac.uk at 4/9/2024, 7:21:54 PM on cl

```

**Figure 1.4.0** The ‘Conditions’ column is divided according to the delimiter that is defined for the specified file root (which is kept in **conditions\_delimiter**). It then splits the resultant array into several rows, each of which has a single condition. This guarantees that any condition in a row that includes multiple conditions separated by the designed delimiter will have its own row.

Then I counted the instances of each condition and organized the DataFrame based on the ‘conditions’ column, after which I ordered the result in descending using ‘.orderby’ in order to achieve desired result I **filtered** out unfilled conditions, and showed the **top 5** rows without truncating the data.

## SQL Implementation



```

Cmd 8
3. Top 5 cond. with their freq.

1 --- QUESTION 3: Top 5 conditions (from Conditions) with their frequencies.
2
3 CREATE OR REPLACE TEMP VIEW All_Freq_Conditions AS SELECT explode(split(clinicaltrial_2023.conditions, '\\|')) AS conditions
4 FROM clinicaltrial_2023;
5 SELECT conditions, count(*) AS count
6 FROM All_Freq_Conditions
7 GROUP BY conditions
8 ORDER BY count DESC
9

(2) Spark Jobs
Table + 
conditions count
1 Healthy 9731
2 Breast Cancer 7502
3 Obesity 6549
4 Stroke 4071
5 Hypertension 4020
6 Depression 3910
7 Prostate Cancer 3896
↓ 10.000 rows | Truncated data | 16.06 seconds runtime
Refreshed 11 minutes ago

Command took 16.06 seconds -- by c.o.airohuddion@edu.salford.ac.uk at 4/12/2024, 8:12:01 AM on sgst

```

**Figure 1.4.1** By exploding the conditions column, this SQL query produces a temporary view called all\_conditions. It then counts the frequency of each conditions and arranges

the result in decreasing order of count. Lastly, it restricts the outcome to the top five circumstances.

**Discussion of Result:** The clinical trial dataset reveals the frequency of common conditions like obesity, stroke, breast cancer, and hypertension, providing valuable insights for further analysis and interpretation.

## SOLUTION 4

**ASSUMPTION:** The ‘Parent Company’ column in the **Pharma\_RDD\_clean** dataset includes all potential pharmaceutical businesses, while the “Sponsor” column in the **ClinicalT\_RDD\_clean** dataset represents clinical trial sponsors.

### RDD Implementation



```
QUESTION 4) Extract, parent pharma companies, compute top 10 non-pharma

1 # Get the index of the 'Sponsor' column in ClinicalT_RDD_clean
2 ct_sponsor_col_index = ClinicalT_RDD_clean.first().index("sponsor")
3
4 # Extract the parent pharmaceutical companies from Pharma_RDD_clean
5 parent_pharm_comp = Pharma_RDD_clean.map(lambda x: x[1].replace("", ""))
6
7 ClinicalT_RDD_clean.map(lambda x: x[ct_sponsor_col_index]) \
8     .filter(lambda row: row != 'Sponsor') \
9     .subtract(parent_pharm_comp) \
10    .map(lambda x: (x, 1)) \
11    .reduceByKey(lambda x, y: x + y) \
12    .sortBy(lambda x: x[1], ascending=False) \
13    .take(10)

▶ (4) Spark Jobs
[('National Cancer Institute (NCI)', 3410),
 ('Assiut University', 3335),
 ('Cairo University', 3023),
 ('Assistance Publique - Hôpitaux de Paris', 2951),
 ('Mayo Clinic', 2766),
 ('M.D. Anderson Cancer Center', 2702),
 ('Novartis Pharmaceuticals', 2393),
 ('National Institute of Allergy and Infectious Diseases (NIAID)', 2340),
 ('Massachusetts General Hospital', 2263),
 ('National Taiwan University Hospital', 2181)]
```

Command took 11.20 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 7:33:46 PM on Compute

**Figure 1.4.2** The RDD’s ‘sponsor’ column index is obtained using the **index()** function. The algorithm extracts parent pharmaceutical companies from the **Pharma\_RDD\_clean** RDD, filters out rows with equal values, subtracts them, and maps each surviving “Sponsor” to a 1-valued tuple, then it counts occurrences by key, and finally sorts results in descending order by count. The ten (10) most prevalent conditions are found by **.take(10)**.

## DATAFRAME Implementation

```

Cmd 6
4. 10 most com. spon. in C not in P

1 # QUESTION 4
2 pharma_list = create_clinical_dataframe(fileroot_pharma).select("Parent_Company").rdd.flatMap(lambda x: x).collect()
3 ct_sponsor_dataframe = ClinicalT_dataframe.select("Sponsor")
4
5 non_pharma_sponsors = ct_sponsor_dataframe.groupBy("Sponsor").count().orderBy("count", ascending=False).filter(~ct_sponsor_dataframe.Sponsor.isin(pharma_list)).show(10, truncate= False)

(4) Spark Jobs
+-----+-----+
|Sponsor|count|
+-----+-----+
|National Cancer Institute (NCI)|3410 |
|Assiut University|3335 |
|Cairo University|3023 |
|Assistance Publique - Hôpitaux de Paris|2951 |
|Mayo Clinic|2766 |
|M.D. Anderson Cancer Center|2702 |
|Novartis Pharmaceuticals|2393 |
|National Institute of Allergy and Infectious Diseases (NIAID)|2340 |
|Massachusetts General Hospital|2263 |
|National Taiwan University Hospital|2181 |
+-----+-----+
only showing top 10 rows

Command took 25.45 seconds -- by c.o.airlohuodion@edu.salford.ac.uk at 4/9/2024, 7:21:54 PM on cl

```

**Figure 1.4.3** The first line of the pyspark code in this solution takes the ‘Parent\_Company’ column of the pharmaceutical DataFrame and generates a list of distinct parent firms. The second line of the code picks the ‘Sponsor’ column from the clinical trial DataFrame (ClinicalT\_dataframe).

The last line classifies the clinical trial DataFrame according to ‘Sponsor’ using ‘.groupby’, counts the instances of each sponsor using .count(), sorts the findings in descending order based on count, removes sponsors that are already on the pharma list, and shows the top 10 non-pharma sponsors without cropping the output.

## SQL Implementation

```

Cmd 9
4. 10 most com. spon. in C not in P

1 CREATE OR REPLACE TEMP VIEW Sponsor_Notpharma_Comp AS SELECT Sponsor FROM clinicaltrial_2023 WHERE Sponsor NOT IN (SELECT Parent_Company FROM pharma);
2
3 SELECT Sponsor, count(*) as count
4 GROUP BY Sponsor
5 ORDER BY count DESC
6 LIMIT 10

(3) Spark Jobs
Table + 
Sponsor count
1 National Cancer Institute (NCI) 3410
2 Assiut University 3335
3 Cairo University 3023
4 Assistance Publique - Hôpitaux de Paris 2951
5 Mayo Clinic 2766
6 M.D. Anderson Cancer Center 2702
7 Novartis Pharmaceuticals 2393
↓ 10 rows | 15.83 seconds runtime
Refreshed 11 minutes ago

Command took 15.83 seconds -- by c.o.airlohuodion@edu.salford.ac.uk at 4/12/2024, 8:12:01 AM on sgst

```

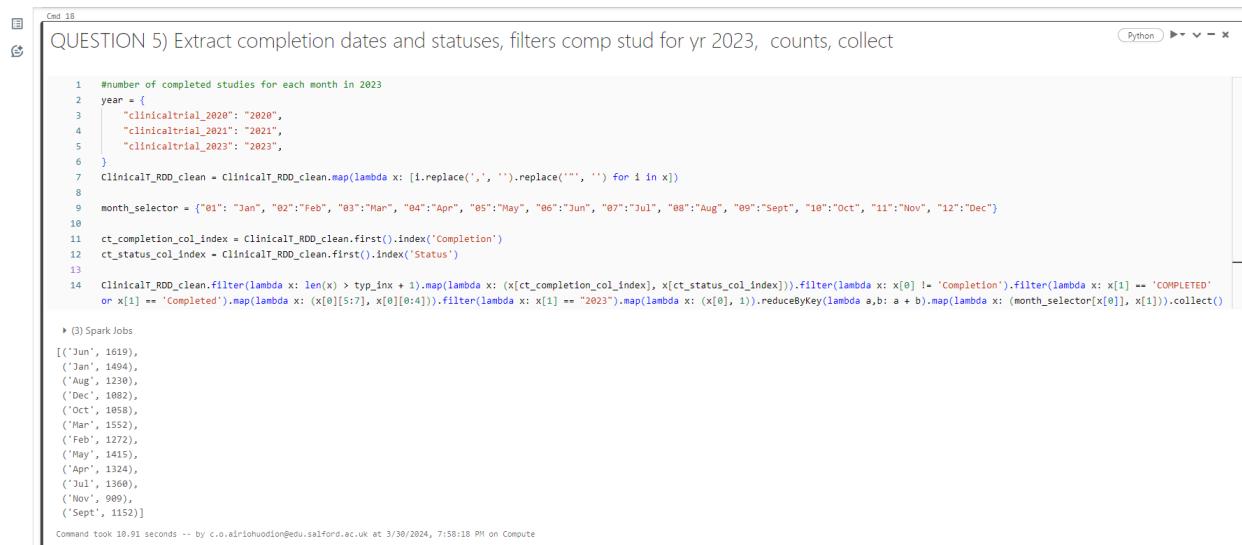
**Figure 1.4.4** In order to filter sponsors from **clinicaltrial\_2023** based in their absence in the pharma database, it constructs a temporary view called **Sponsor\_Notpharma\_Comp**. The output is then **limited** to the top 10 sponsors after **counting** the number of times each sponsor appears in the temporary display and sorting the results by count in descending order.

**Discussion of Result:** The outcome reveals the top ten non-pharmaceutical sponsors and their financing of clinical trials, showcasing a diverse range of institutions and groups supporting clinical studies. The National Cancer institute is the most frequent sponsor, alongside academic institutions and medical centers.

## SOLUTION 5

**ASSUMPTION:** The dataset includes “Status” columns for current study status and “Completion” columns for completion date, with completion date in ‘YYYY-M-DD’ format.

### RDD Implementation



```

Cmd 18
QUESTION 5) Extract completion dates and statuses, filters comp stud for yr 2023, counts, collect
Python

1 #number of completed studies for each month in 2023
2 year = {
3     "clinicaltrial_2020": "2020",
4     "clinicaltrial_2021": "2021",
5     "clinicaltrial_2023": "2023",
6 }
7 ClinicalTrial_RDD_clean = ClinicalTrial_RDD_clean.map(lambda x: [i.replace(',', '') for i in x])
8
9 month_selector = {"01": "Jan", "02": "Feb", "03": "Mar", "04": "Apr", "05": "May", "06": "Jun", "07": "Jul", "08": "Aug", "09": "Sept", "10": "Oct", "11": "Nov", "12": "Dec"}
10
11 ct_completion_col_index = ClinicalTrial_RDD_clean.first().index('Completion')
12 ct_status_col_index = ClinicalTrial_RDD_clean.first().index('Status')
13
14 ClinicalTrial_RDD_clean.filter(lambda x: len(x) > typ_inx + 1).map(lambda x: (x[ct_completion_col_index], x[ct_status_col_index])).filter(lambda x: x[0] != 'Completion').filter(lambda x: x[1] == 'COMPLETED' or x[1] == 'Completed').map(lambda x: (x[0][5:], x[0][0:4])).filter(lambda x: x[1] == "2023").map(lambda x: (x[0], 1)).reduceByKey(lambda a,b: a + b).map(lambda x: (month_selector[x[0]], x[1])).collect()
15
16 (3) Spark Jobs
17
18 [('Jun', 1619),
19 ('Jan', 1404),
20 ('Aug', 1230),
21 ('Dec', 1082),
22 ('Oct', 1058),
23 ('Mar', 1552),
24 ('Feb', 1272),
25 ('May', 1415),
26 ('Apr', 1324),
27 ('Jul', 1360),
28 ('Nov', 909),
29 ('Sept', 1152)]

```

Command took 10.91 seconds -- by c.o.airlohuodion@edu.salford.ac.uk at 3/30/2024, 7:58:18 PM on Compute

**Figure 1.4.5** This code calculates the total number of studies completed each month in 2023 by mapping file names to corresponding years using a dictionary called year, eliminating commas and double quotes, and using the **month\_selector** dictionary to translate month names into numerical representations. It finds the ‘Completion’ and ‘Status’ column indexes in the RDD, filters out rows with missing data, and extracts the month and year from the completion date. It applies a 2023 filter, mapping each month to a tuple with a value of 1, decreasing by key, and using the **month\_selector** to map each month to its associated name.

## DATAFRAME Implementation

```
QUESTION 5) Extract completion dates and statuses, filter comp stud for yr 2023, counts, show
```

```
1  from pyspark.sql.functions import split, regexp_replace, expr
2
3  # Apply transformations to the ClinicalT_dataframe
4  for col in ClinicalT_dataframe.columns:
5      ClinicalT_dataframe = ClinicalT_dataframe.withColumnRenamed(col, col.strip(",").strip('"'))
6
7  # Extracting month and year from Completion column
8 completed_cd = ClinicalT_dataframe.withColumn("Year", split('Completion', "-")[0]) \
9     .withColumn('Month', split('Completion', "-")[1])
10    .withColumn('Month', regexp_replace("Month", ",", ""))
11    .withColumn('Month', regexp_replace("Month", "'", ""))
12    .filter(ClinicalT_dataframe.Status.isin(["COMPLETED"]))
13    .select("Month", "Year", "Status")
14
15 # Filter and aggregate to get counts for each month in 2023
16 month_counts = completed_cd.filter(completed_cd.Year.isin(["2023"])) \
17     .groupBy("Month").count().orderBy("Month", ascending=True)
18
19 # Apply the mapping using expr to convert month numbers to month names
20 month_df = month_counts.withColumn("month",
21     expr("""
22         CASE
23             WHEN month = '01' THEN 'Jan'
24             WHEN month = '02' THEN 'Feb'
25             WHEN month = '03' THEN 'Mar'
26             WHEN month = '04' THEN 'Apr'
27             WHEN month = '05' THEN 'May'
28             WHEN month = '06' THEN 'June'
29             WHEN month = '07' THEN 'July'
30             WHEN month = '08' THEN 'Aug'
31             WHEN month = '09' THEN 'Sep'
32             WHEN month = '10' THEN 'Oct'
33             WHEN month = '11' THEN 'Nov'
34             WHEN month = '12' THEN 'Dec'
35             ELSE 'Null'
36         END AS month
37     """))
38
39 # Select the month and count columns
40 result_df = month_df.select("month", "count")
41 result_df.show()
```

**Figure 1.4.6** The code imports split and regexp\_replace, renaming column names and removing leading and following commas and double quotes. The Completion column is split into Years and Month columns, and the Dataframe is filtered to include records with Status equal to ‘COMPLETED’. The completed\_cd DataFrame is created, grouped by Month and aggregated for each month of 2023, sorting results in ascending order.

```
(2) Spark Jobs
  ▶ ClinicalT_dataframe: pyspark.sql.dataframe.DataFrame = [Id: string, Study Title: string ... 12 more fields]
  ▶ completed_cd: pyspark.sql.dataframe.DataFrame = [Month: string, Year: string ... 1 more field]
  ▶ month_counts: pyspark.sql.dataframe.DataFrame = [Month: string, count: long]
  ▶ month_df: pyspark.sql.dataframe.DataFrame = [month: string, count: long]
  ▶ result_df: pyspark.sql.dataframe.DataFrame = [month: string, count: long]

+-----+
|month|count|
+-----+
| Jan| 1494|
| Feb| 1272|
| Mar| 1552|
| Apr| 1324|
| May| 1415|
| June| 1619|
| July| 1360|
| Aug| 1230|
| Sep| 1152|
| Oct| 1058|
| Nov| 909|
| Dec| 1082|
+-----+

Command took 13.31 seconds -- by c.o.airi@ucl.ac.uk at 5/2/2024, 12:45:39 AM on Compute
```

**Figure 1.4.7** The result above is the outcome of the DataFrame code using show() method to display the number of clinical trials completed each month in 2023 .

## SQL Implementation



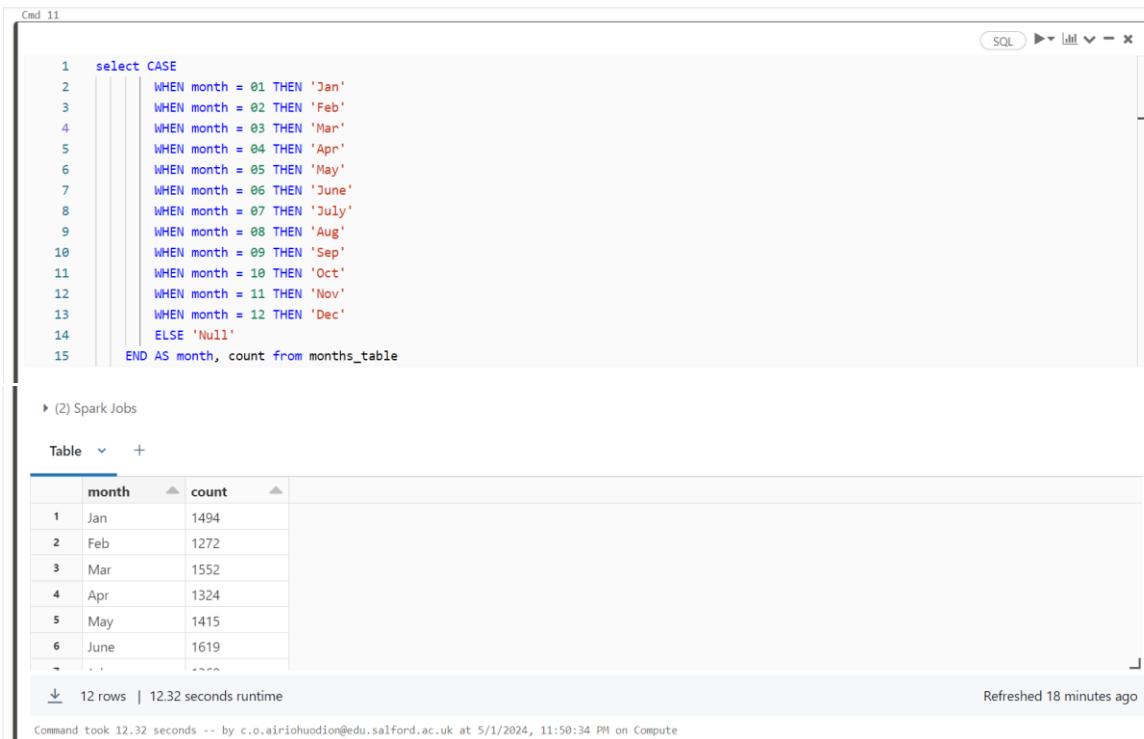
```
QUESTION 5) Extract completion dates and statuses, filter comp stud for yr 2023, counts, show

1 --QUESTION 5: Number of completed studies for each month in 2023
2
3 CREATE OR REPLACE TEMP VIEW Months_Table AS SELECT split(clinicaltrial_2023.Completion, '-' )[1] AS month, COUNT(*) AS count
4 FROM clinicaltrial_2023
5 WHERE clinicaltrial_2023.Status = 'COMPLETED'
6 AND split(clinicaltrial_2023.Completion, '-' )[0] = '2023'
7 GROUP BY month
8 ORDER BY month

OK

Command took 0.81 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/12/2024, 8:12:01 AM on sgst
```

**Figure 1.4.8** I created a temporary view named **Months\_Table** that shows the number of studies finished in each month of 2023. This is accomplished by **filtering** for clinical trials that were completed in 2023 and **dividing** the completion date of trials into **month-by-year** components. The outcome is arranged and **classified by month**.



```
Cmd 11
SQL ▶ | | | - x

1 select CASE
2   WHEN month = 01 THEN 'Jan'
3   WHEN month = 02 THEN 'Feb'
4   WHEN month = 03 THEN 'Mar'
5   WHEN month = 04 THEN 'Apr'
6   WHEN month = 05 THEN 'May'
7   WHEN month = 06 THEN 'June'
8   WHEN month = 07 THEN 'July'
9   WHEN month = 08 THEN 'Aug'
10  WHEN month = 09 THEN 'Sep'
11  WHEN month = 10 THEN 'Oct'
12  WHEN month = 11 THEN 'Nov'
13  WHEN month = 12 THEN 'Dec'
14  ELSE 'Null'
15 END AS month, count FROM months_table

(2) Spark Jobs

Table +
```

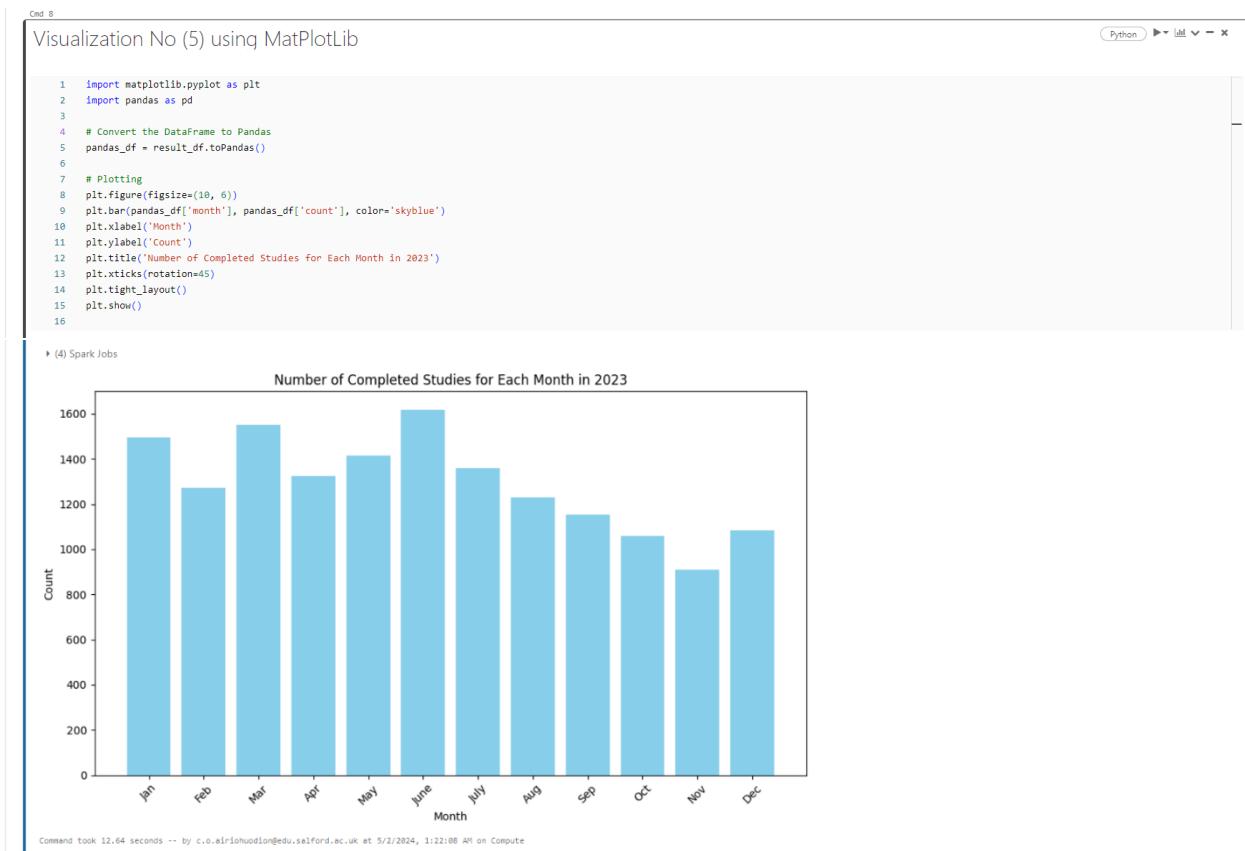
	month	count
1	Jan	1494
2	Feb	1272
3	Mar	1552
4	Apr	1324
5	May	1415
6	June	1619
..		....

↓ 12 rows | 12.32 seconds runtime      Refreshed 18 minutes ago

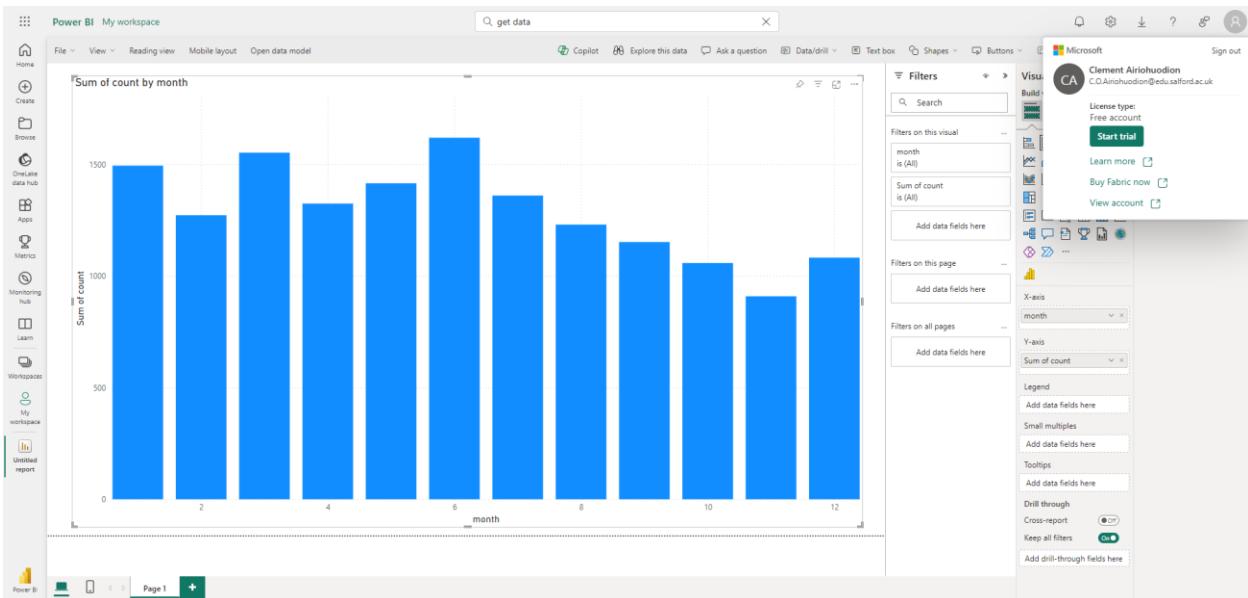
```
Command took 12.32 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 5/1/2024, 11:50:34 PM on Compute
```

**Figure 1.4.9** The data is then retrieved from the **Months\_Table** view using the **SELECT** query to access the total counts of studies for each month in 2023 and then replace each month with its respective names.

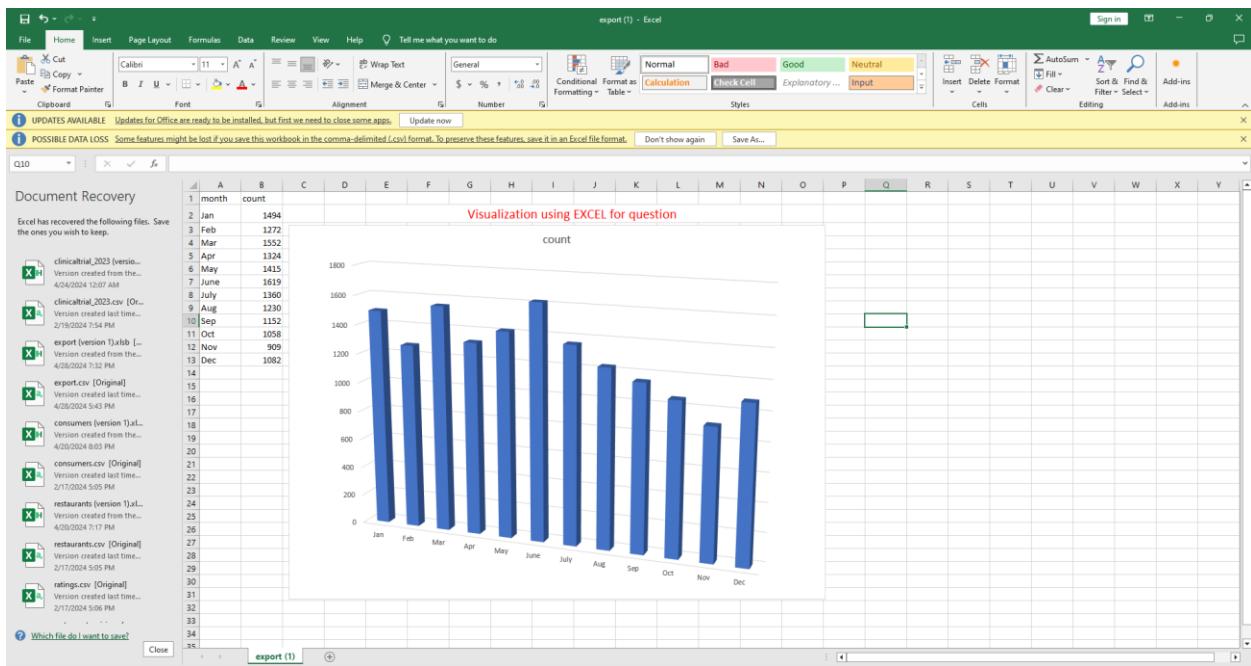
Discussion of Result: In 2023, research completion rates vary by month, with June having the highest number (1619) and January and August following closely behind (1230), indicating potential trends affecting year-round completion rates.



**Figure 1.5.0** Visualization using Matplotlib for question (5) in Dataframe NoteBook



**Figure 1.5.1** Visualization using PowerBi (Web) for question (5)



**Figure 1.5.2 Visualization using EXCEL Sheet for question (5)**

**Further Analyses of the data were done in my Dataframe Notebook, motivated by the questions as follows:**

1. How many studies have interventions involving drug treatments, and what is their distribution across different conditions

The screenshot shows a Jupyter Notebook cell with the following content:

```
Cmd 9
Python ▶ v - ×

1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import explode, split, col, count
3
4 # Assuming you have already loaded your clinical trial data into a PySpark DataFrame called `ClinicalT_dataframe`
5
6 # Split interventions and explode them to separate rows
7 intervention_df = ClinicalT_dataframe.withColumn('Intervention', explode(split(col('Interventions'), '\|')))
8
9 # Filter studies with interventions involving drug treatments
10 drug_intervention_df = intervention_df.filter(intervention_df.Intervention.contains('DRUG'))
11
12 # Count the number of studies with drug interventions and their distribution across different conditions
13 drug_intervention_count = drug_intervention_df.groupBy('Conditions').agg(count('Id').alias('NumStudies')).orderBy('NumStudies', ascending=False)
14
15 drug_intervention_count.show(truncate=False)
16
```

(2) Spark Jobs

- intervention\_df: pyspark.sql.dataframe.DataFrame = [Id: string, Study Title: string ... 13 more fields]
- drug\_intervention\_df: pyspark.sql.dataframe.DataFrame = [Id: string, Study Title: string ... 13 more fields]
- drug\_intervention\_count: pyspark.sql.dataframe.DataFrame = [Conditions: string, NumStudies: long]

Conditions	NumStudies
Healthy	12159
Breast Cancer	4691
Asthma	3053
HIV Infections	3029
Healthy Volunteers	2927
Multiple Myeloma	2611
Diabetes Mellitus", " Type 2	2371
Type 2 Diabetes Mellitus	2237

Command took 12.69 seconds -- by c.o.airiohudion@edu.salford.ac.uk at 4/30/2024, 9:18:55 PM on jfgfjjfjg

**Figure 1.5.2** This code counts the number of studies and their distribution across various conditions after first splitting the interventions and then filtering for studies utilizing medication treatments.

2. How many studies have been terminated, and what are their respective titles.

The screenshot shows a Jupyter Notebook cell with the following code:

```
1 from pyspark.sql import SparkSession
2
3 # Assuming you have already loaded your clinical trial data into a PySpark DataFrame called `ClinicalT_dataframe`
4
5 # Filter studies that have been terminated
6 terminated_studies_df = ClinicalT_dataframe.filter(ClinicalT_dataframe.Status == 'TERMINATED')
7
8 # Count the number of terminated studies
9 num_terminated_studies = terminated_studies_df.count()
10
11 # Show the titles of terminated studies
12 terminated_studies_df.select('Study Title').show(num_terminated_studies, truncate=False)
13
```

Below the code, the output is displayed:

▶ (4) Spark Jobs

terminated\_studies\_df: pyspark.sql.dataframe.DataFrame = [Id: string, Study Title: string ... 12 more fields]

Study Title
Impact of Tight Glycaemic Control in Acute Myocardial Infarction
Preoperative Use of Darifenacin (Enablex) to Alleviate Postoperative Ureteral Stent Pain
Study of Evobrutinib in Participants With RMS
Sunitinib and Capecitabine for First Line Colon Cancer

Command took 26.73 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 5/1/2024, 7:28:21 PM on Task 1.0

**Figure 1.5.3** The code filters the DataFrame to include only the terminated studies, counts the number of terminated studies, and displays their respective titles.

3. Analyze the distribution of study durations(Completion- Start) using quartiles and determine the interquartile range (IQR) for each type of study.

Cmd 11

Python ▶️ 🔍 ⌂

```
1 from pyspark.sql.functions import col, datediff, expr
2
3 # Assuming you have already loaded your DataFrame as ClinicalT_dataframe
4
5 # Step 1: Calculate the duration of each study
6 study_durations = ClinicalT_dataframe.withColumn('Duration', datediff(col('Completion'), col('Start')))
7
8 # Step 2: Group the studies by the Type column
9 grouped_by_type = study_durations.groupBy('Type')
10
11 # Step 3: Calculate quartiles for the duration of each type of study
12 quartiles = grouped_by_type.agg(expr('percentile(Duration, array(0.25, 0.5, 0.75))').alias('quartiles'))
13
14 # Step 4: Compute the interquartile range (IQR) for each type of study
15 iqr = quartiles.withColumn('Q1', col('quartiles')[0]) \
16     .withColumn('Q2', col('quartiles')[1]) \
17     .withColumn('Q3', col('quartiles')[2]) \
18     .withColumn('IQR', col('Q3') - col('Q1')) \
19     .select('Type', 'Q1', 'Q2', 'Q3', 'IQR')
20
21 # Show the results
22 iqr.show()
23
```

▶ (2) Spark Jobs

- ▶ [study\_durations: pyspark.sql.dataframe.DataFrame = [Id: string, Study Title: string ... 13 more fields]]
- ▶ [quartiles: pyspark.sql.dataframe.DataFrame = [Type: string, quartiles: array]]
- ▶ [iqr: pyspark.sql.dataframe.DataFrame = [Type: string, Q1: double ... 3 more fields]]

Type	Q1	Q2	Q3	IQR
INTERVENTIONAL	365.0	759.0	1386.0	1021.0
OBSERVATIONAL	366.0	799.0	1583.0	1217.0
EXPANDED_ACCESS	632.0	1293.5	1994.0	1362.0
	NULL	NULL	NULL	NULL
	NULL	NULL	NULL	NULL

Command took 17.43 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 5/1/2024, 7:58:18 PM on Task 1.0

**Figure 1.5.3** It calculates the length of each study, groups them based on type, determines quartiles for each type, and calculates the interquartile range (IQR) for each

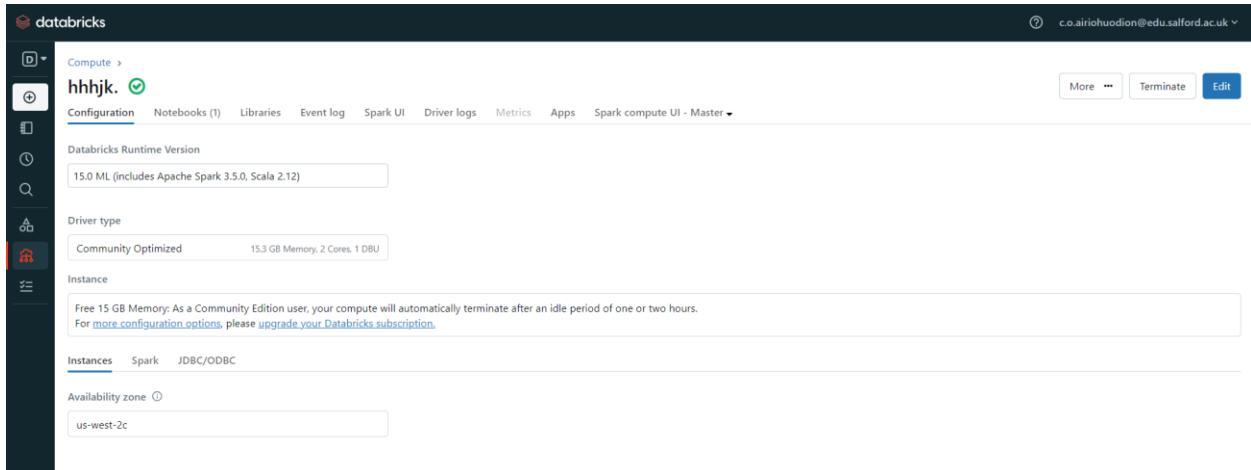
## **TASK TWO**

## **INTRODUCTION**

The dataset used in the analysis was taken from the online game distribution platform Steam. This dataset includes details on the games that various members have purchased and played, as well as the related gameplay durations.

Within the Databricks environment, PySpark Dataframe features were used to preprocess and analyze the Data. Numerous investigations were carried out, such as identifying the most popular games to play, buy and have the longest playthrough periods. The dataset was also used to carry out machine learning activities, such as dividing the data into training and testing sets and applying the Alternating Least Squares (ALS) method to train a recommendation system.

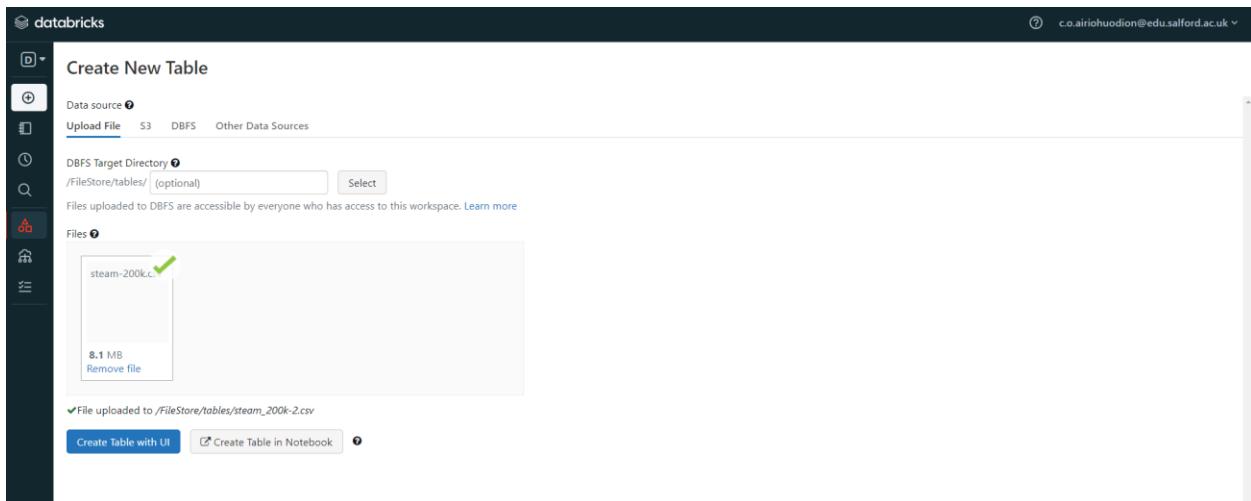
## COMPUTE SETUP AND FILE IMPORT



**Figure 2.1.0 Compute Setup**

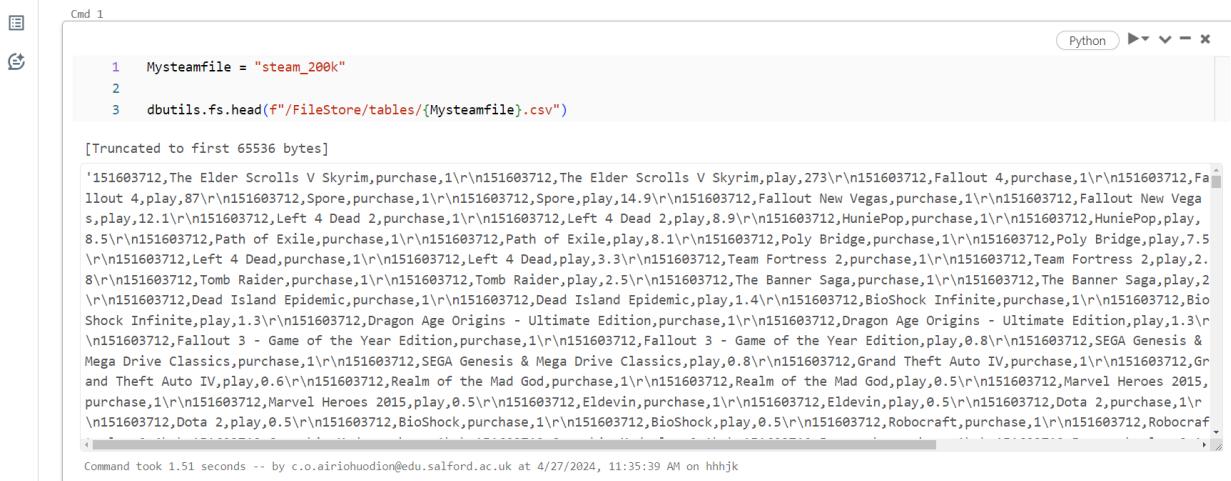
A cluster with 15GB space having Apache Spark 3.5.0 and Scala 2.12 was created and used for the implementation of the task on DataBricks Community Cloud Platform to create clusters containing two attached cores.

## UPLOADING DATASET



**Figure 2.1.1 Uploaded steam-200k.csv dataset using the (GUI)**

## Loading The Data



The screenshot shows a Jupyter Notebook cell with the following content:

```
Cmd 1
Python ▶ v - ×

1 Mysteamfile = "steam_200k"
2
3 dbutils.fs.head(f"/FileStore/tables/{Mysteamfile}.csv")

[Truncated to first 65536 bytes]

'151603712,The Elder Scrolls V Skyrim,purchase,1\r\n151603712,The Elder Scrolls V Skyrim,play,273\r\n151603712,Fallout 4,purchase,1\r\n151603712,Fallout 4,play,12.9\r\n151603712,Fallout New Vegas,purchase,1\r\n151603712,Fallout New Vegas,play,12.1\r\n151603712,Left 4 Dead 2,purchase,1\r\n151603712,Left 4 Dead 2,play,8.9\r\n151603712,HuniePop,purchase,1\r\n151603712,HuniePop,play,8.5\r\n151603712,Path of Exile,purchase,1\r\n151603712,Path of Exile,play,8.1\r\n151603712,Poly Bridge,purchase,1\r\n151603712,Poly Bridge,play,7.5\r\n151603712,Left 4 Dead,purchase,1\r\n151603712,Left 4 Dead,play,3.3\r\n151603712,Team Fortress 2,purchase,1\r\n151603712,Team Fortress 2,play,2.8\r\n151603712,Tomb Raider,purchase,1\r\n151603712,Tomb Raider,play,2.5\r\n151603712,The Banner Saga,purchase,1\r\n151603712,The Banner Saga,play,2\r\n151603712,Dead Island Epidemic,purchase,1\r\n151603712,Dead Island Epidemic,play,1.4\r\n151603712,BioShock Infinite,purchase,1\r\n151603712,BioShock Infinite,play,1.3\r\n151603712,Dragon Age Origins - Ultimate Edition,purchase,1\r\n151603712,Dragon Age Origins - Ultimate Edition,play,1.3\r\n151603712,Fallout 3 - Game of the Year Edition,purchase,1\r\n151603712,Fallout 3 - Game of the Year Edition,play,0.8\r\n151603712,SEGA Genesis & Mega Drive Classics,purchase,1\r\n151603712,SEGA Genesis & Mega Drive Classics,play,0.8\r\n151603712,Grand Theft Auto IV,purchase,1\r\n151603712,Grand Theft Auto IV,play,0.6\r\n151603712,Realm of the Mad God,purchase,1\r\n151603712,Realm of the Mad God,play,0.5\r\n151603712,Marvel Heroes 2015,purchase,1\r\n151603712,Marvel Heroes 2015,play,0.5\r\n151603712,Eldevin,purchase,1\r\n151603712,Eldevin,play,0.5\r\n151603712,Dota 2,purchase,1\r\n151603712,Dota 2,play,0.5\r\n151603712,BioShock,purchase,1\r\n151603712,BioShock,play,0.5\r\n151603712,Robocraft,purchase,1\r\n151603712,Robocraft,play,0.5

Command took 1.51 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhjk
```

**Figure 2.1.2** Retrieving and displaying the first lines of the CSV file named “steam\_200.csv”.

This file has been stored in the Databricks FileStore, and has been given the variable name “Mysteamfile”.

## Loading Data and Exploratory Analysis

### Creating DataFrame

```
Cmd. 3 Python ▶▼ - ×
1 #CREATING A DATAFRAME
2 from pyspark.sql.functions import *
3 from pyspark.sql.types import StructField, StructType
4
5
6 def create_dataframe(fileroot):
7     df = spark.read.options(delimiter = ",").csv(f"/FileStore/tables/{fileroot}.csv")
8     return df
9
10 steam_dataf = create_dataframe("Mysteamfile")
11 steam_dataf.show(5)

▶ (2) Spark Jobs
▶ steam_dataf: pyspark.sql.dataframe.DataFrame = [c0: string, c1: string ... 2 more fields]
+-----+-----+-----+
| _c0 | _c1 | _c2|_c3|
+-----+-----+-----+
|151603712|The Elder Scrolls...|purchase| 1|
|151603712|The Elder Scrolls...| play|273|
|151603712|          Fallout 4|purchase| 1|
|151603712|          Fallout 4| play| 87|
|151603712|           Spore|purchase| 1|
+-----+-----+-----+
only showing top 5 rows

Command took 18.26 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk
```

**Figure 2.1.3** This section of code takes the CSV file containing Steam game data and uses it to generate a DataFrame called `steam_dataf`. It provides a method named `create_dataframe`, with a comma (,) as the delimiter, that uses Spark's DataFrame API to read the CSV file. The function accepts an argument called `fileroot`, which is the file root name. The `show()` function is used by the code to display the first 10 rows after it has been created. The procedure enables preliminary investigation of the content and structure of the dataset.

Creating a DataFrame and inferring a schema from `steam200k.csv`

Replace the generic column names from the last data frame and datatype

```
Cmd 4 Python ▶ ▷ ▷ - x
1
2 def alter_schema_name(df):
3 # Rename columns
4     df = df.withColumnRenamed("_c0", "User_ID") \
5         .withColumnRenamed("_c1", "Game") \
6         .withColumnRenamed("_c2", "Member_Behaviour") \
7         .withColumnRenamed("_c3", "PlayTime_Length")
8     df = df.withColumn("User_ID", df["User_ID"].cast("int")).withColumn("PlayTime_Length", df["PlayTime_Length"].cast("float"))
9     return df
10 # Show the updated DataFrame
11 steam_dataf = alter_schema_name(steam_dataf)
12 steam_dataf.show(5)
▶ (1) Spark Jobs
▶ [steam_dataf: pyspark.sql.dataframe.DataFrame = [User_ID: integer, Game: string ... 2 more fields]
+-----+-----+-----+
| User_ID | Game | Member_Behaviour | PlayTime_Length |
+-----+-----+-----+
| 151603712 | The Elder Scrolls... | purchase | 1.0 |
| 151603712 | The Elder Scrolls... | play | 273.0 |
| 151603712 | Fallout 4 | purchase | 1.0 |
| 151603712 | Fallout 4 | play | 87.0 |
| 151603712 | Spore | purchase | 1.0 |
+-----+-----+-----+
only showing top 5 rows
Command took 1.73 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk
```

**Figure 2.1.4**

In order to replace the generic column names from the last data frame, I've written this function to alter the previous schema names, by changing the schema of the **steam\_dataf**, which contains the gaming data. It changes all four (4) column **c0,c1,c2,c3** to **User\_ID,Game,Member\_Behaviour,PlayTime\_Length** names including the data types of 'User\_ID' to 'int' and PlayTime\_Length to float from the previous string datatype. Then I show first 10 rows of the updated DataFrame.

Summary statistics

```
Cmd 5
```

```
1 # Summary statistics
2 steam_dataf.describe().show()
```

▶ (2) Spark Jobs

	User_ID	Game	Member_Behaviour	PlayTime_Length
count	200000	200000	200000	200000
mean	1.0365586594664E8	140.0	NULL	17.87438400420385
stddev	7.208073512913968E7	0.0	NULL	138.05695165082415
min	5250	007 Legends	play	0.1
max	309903146	theHunter Primal	purchase	11754.0

Command took 10.98 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.1.4** The summary statistics for the DataFrame counting Steam data include count, mean, standard deviation, minimum, and maximum values for numerical columns like User\_ID and Playtime\_Length.

Number of distinct records

```
Cmd 6
```

```
1 #Analysis; Number of distinct records
2
3 numb_dist_record = steam_dataf.distinct().count()
4 print("Number of distinct records:", numb_dist_record)
```

▶ (3) Spark Jobs

Number of distinct records: 199293

Command took 5.83 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.1.5** This code has calculated the unique records in the DataFrame **steam\_dataf** by removing **duplicates** and **counting** the remaining unique records, providing insight into the datasets variety and facilitating preliminary data **exploration**.

### Count of Unique Games

```
Cmd 7
```

```
1 #Analysis; Unique Games
2 |
3 number_games = steam_dataf.select("Game").distinct().count()
4 print("Number of unique games:", number_games)
```

▶ (3) Spark Jobs  
Number of unique games: 5155  
Command took 2.50 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.1.6** This code analyzes the ‘Game’ column in the **steam\_dataf** DataFrame, and removes duplicates, calculates the number of unique games using **distinct()** and **count()** functions, providing gaming total number of unique games.

### Count of Unique users

```
Cmd 8
```

```
1 # Analysis; Unique users
2 |
3 number_users = steam_dataf.select("User_ID").distinct().count()
4 print("Number of unique users:", number_users)
```

▶ (3) Spark Jobs  
Number of unique users: 12393  
Command took 2.41 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.1.7** The code analyzes the DataFrame **steam\_dataf** to determine the number of **distinct** users by selecting the ‘User\_ID’ column, removing duplicates, and calculating the total unique user IDs, aiding in understanding gaming user engagement.

### Number of Member\_Behaviour records with the details

Cmd 9

```
1 # Number of Member_Behaviour records with the details
2 Member_Behaviour = steam_dataf.groupBy('Member_Behaviour').count()
3 Member_Behaviour.show()

▶ (2) Spark Jobs
▶ Member_Behaviour: pyspark.sql.dataframe.DataFrame = [Member_Behaviour: string, count: long]
+-----+-----+
|Member_Behaviour| count|
+-----+-----+
|      purchase|129511|
|         play| 70489|
+-----+-----+
```

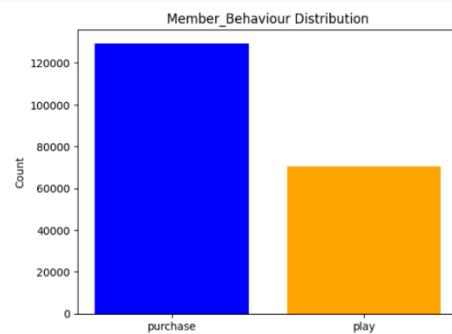
Command took 2.89 seconds -- by c.o.airohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.1.8** The process involves calculating member behavior instances, grouping Steam data based on that behavior, and displaying the distribution of behaviors, including “play” and “purchase” numbers.

### Visualization of Member Behavior Distribution

Cmd 10

```
1 import matplotlib.pyplot as plt
2
3 # Get Member_Behaviour counts
4 steam_counts = steam_dataf.groupBy('Member_Behaviour').count().toPandas()
5
6 # Define colors for the bars
7 colors = ['Blue', 'Orange']
8
9 # Create a bar chart of the counts
10 plt.bar(steam_counts['Member_Behaviour'], steam_counts['count'], color=colors)
11
12 # Set the chart title and axis labels
13 plt.title('Member_Behaviour Distribution')
14 plt.xlabel('Member_Behaviour')
15 plt.ylabel('Count')
16
17 # Show the chart
18 plt.show()
```



Command took 4.01 seconds -- by c.o.airohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.1.9** This code creates a bar chart using the Steam dataset, counting member behavior using Spark SQLs groupBy function, and converting it to a Pandas DataFrame. The chart displays the distribution of member activity types, with “purchase and play” represented by distinct colors.

Total number of purchased games which has been played

Cmd 11

```
1 #Analysis; Total number of purchased games which has been played
2
3 steam_dataaf.select("Game", "Member_Behaviour", "PlayTime_Length").filter(steam_dataaf["Member_Behaviour"] == "play").count()
```

▶ (2) Spark Jobs

70489

Command took 1.90 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.2.0** The code analyzes the total number of **purchased games** using the DataFrame **steam\_dataframe**, filtering it to include rows with ‘**play**’ behavior in the ‘**Member\_Behavior**’ column. The **count()** method determines the number of games bought and played.

### Most Often played games (Highest played game)

```
Cmd 12
```

```
Python ▶▶ ▶▶ - x
```

```
1 #Analysis; Most Often played games (Highest played game)
2
3 # Filter the dataframe for "play" behavior, group by "Game", count occurrences, and order by count
4 most_played_games = steam_dataframe.filter(steam_dataframe["Member_Behaviour"] == "play") \
5     .groupBy("Game") \
6     .count() \
7     .orderBy('count', ascending=False) \
8     .limit(10) # Limit the result to top 10 games
9 most_played_games.show(truncate=False)
```

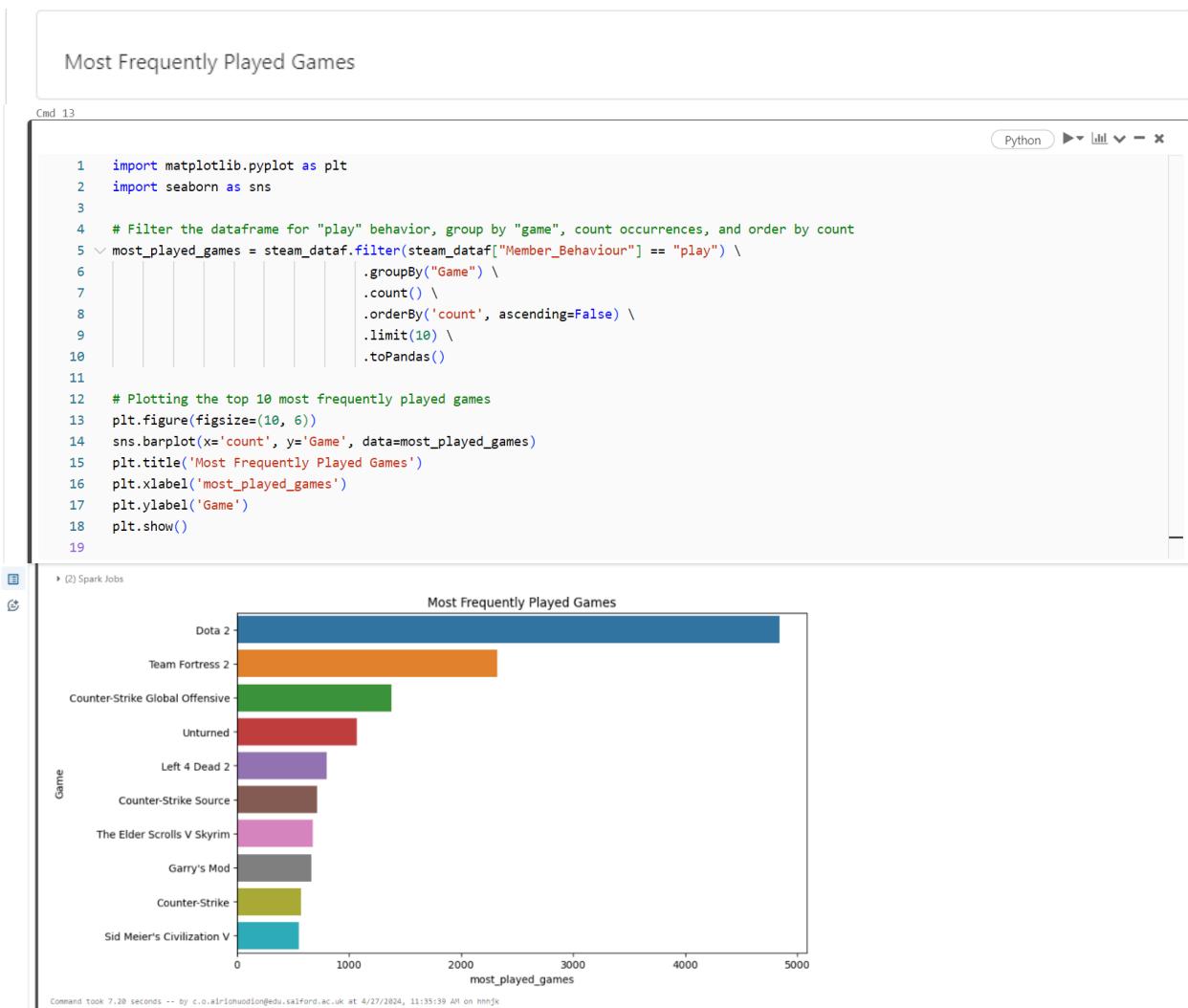
► (2) Spark Jobs

```
► most_played_games: pyspark.sql.dataframe.DataFrame = [Game: string, count: long]
```

Game	count
Dota 2	4841
Team Fortress 2	2323
Counter-Strike Global Offensive	1377
Unturned	1069
Left 4 Dead 2	801
Counter-Strike Source	715
The Elder Scrolls V Skyrim	677
Garry's Mod	666
Counter-Strike	568
Sid Meier's Civilization V	554

```
Command took 2.84 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk
```

**Figure 2.2.1** This code analyzes the top played games using the DataFrame **steam\_dataframe**. It filters rows with play behavior, aggregates information, **counts** instances, **sorts** them in **descending order**, and displays the results. The **limit()** function restricts results to the top 10 most played games, identifying the games with the highest player involvement.



**Figure 2.2.2** It employs Seaborn to generate a bar plot with x-axis and y-axis, providing a visual representation of the most played games.

Calculate the count of purchases per game

Cmd 14

Python ► ▾ ▷ ✎

```
1 # Calculate the count of purchases per game
2 most_purchased_games = steam_dataf.filter(steam_dataf["Member_Behaviour"] == "purchase") \
3                                         .groupBy("Game") \
4                                         .count() \
5                                         .orderBy('count', ascending=False) \
6                                         .limit(10) # Limit the result to top 10 games
7
8 most_purchased_games.show(truncate=False)
```

▶ (2) Spark Jobs

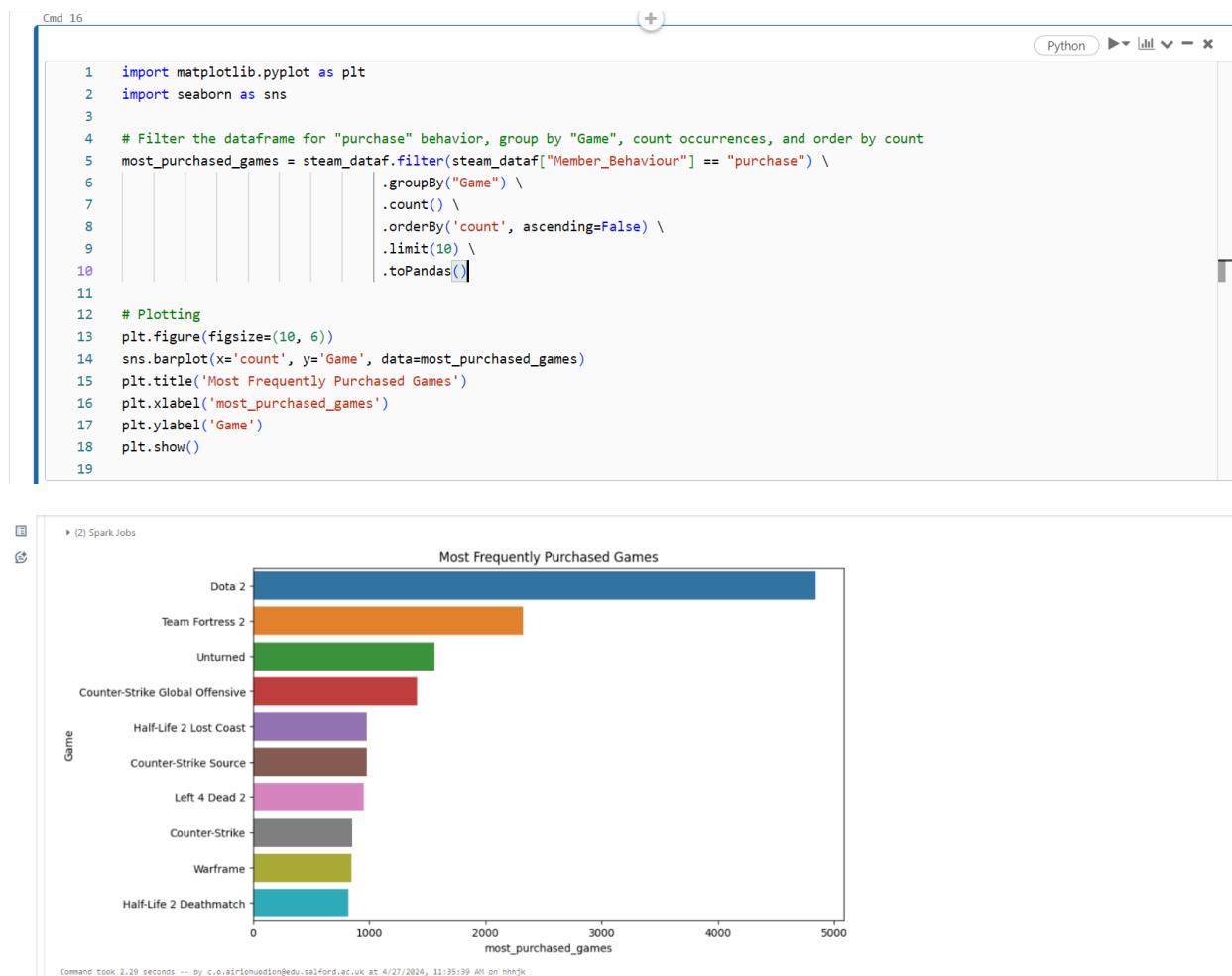
▶ most\_purchased\_games: pyspark.sql.dataframe.DataFrame = [Game: string, count: long]

Game	count
Dota 2	4841
Team Fortress 2	2323
Unturned	1563
Counter-Strike Global Offensive	1412
Half-Life 2 Lost Coast	981
Counter-Strike Source	978
Left 4 Dead 2	951
Counter-Strike	856
Warframe	847
Half-Life 2 Deathmatch	823

Command took 2.41 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.2.3** This code analysis game **purchases** using the DataFrame **steam\_dataframe**, focusing on **popular titles**. It filters entries based on purchasing behavior, aggregates information, counts instances, sorts them in decreasing order, and displays results, remove truncate using the **show()** method and **limit()** function to allow only first 10 rows . This study provides data on game demand and popularity.

### Most Frequently Purchased Games



**Figure 2.2.4** It employs Seaborn to generate a bar plot with x-axis and y-axis, providing a visual representation of the most purchased games.

### Games with the highest number of PlayTime\_Length in hours

```
Cmd 17
```

```
1 #Analysis; Games with the highest number of PlayTime_Length in hours
2
3 # Calculate the total play time length per game
4 total_playtime_per_game = steam_dataf.filter(steam_dataf["Member_Behaviour"] == "play") \
5     .groupBy("Game") \
6     .agg(round(sum(steam_dataf["PlayTime_Length"]), 2).alias("Total_PlayTime_IN_HOUR")) \
7     .orderBy('Total_PlayTime_IN_HOUR', ascending=False) \
8     .limit(10)
9 total_playtime_per_game.show(truncate=False)
```

▶ (2) Spark Jobs

▶ total\_playtime\_per\_game: pyspark.sql.dataframe.DataFrame = [Game: string, Total\_PlayTime\_IN\_HOUR: double]

Game	Total_PlayTime_IN_HOUR
Dota 2	981684.6
Counter-Strike Global Offensive	322771.6
Team Fortress 2	173673.3
Counter-Strike	134261.1
Sid Meier's Civilization V	99821.3
Counter-Strike Source	96075.5
The Elder Scrolls V Skyrim	70889.3
Garry's Mod	49725.3
Call of Duty Modern Warfare 2 - Multiplayer	42009.9
Left 4 Dead 2	33596.7

Command took 2.71 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhjk

**Figure 2.2.5** This code fetches games with extended playtime to determine the largest cumulative playtime. It filters the DataFrame to show play behavior, **groups** games by 'Game' column, and adds 'PlayTime\_Length' numbers to calculate overall duration of play in **descending** order. This research focuses on the top 10 games with the longest overall playtime, providing insights into their popularity and user involvement levels.

The game that has the highest average play time per user in hours

```
Cmd 18
1 #The game that has the highest average play time per user in hours
2 avg_playtime_per_game = steam_dataframe.filter(steam_dataframe["Member_Behaviour"] == "play") \
3 | groupBy("Game") \
4 | .agg(round(avg("PlayTime_Length"), 2).alias("Avg_PlayTimeIN_HOUR")) \
5 | .orderBy("Avg_PlayTimeIN_HOUR", ascending=False) \
6 | .limit(10)
7 avg_playtime_per_game.show(truncate=False)

▶ (2) Spark Jobs
▶ avg_playtime_per_game: pyspark.sql.dataframe.DataFrame = [Game: string, Avg_PlayTimeIN_HOUR: double]
+-----+-----+
|Game |Avg_PlayTimeIN_HOUR|
+-----+-----+
|Eastside Hockey Manager |1295.0 |
|Baldur's Gate II Enhanced Edition|475.26 |
|FIFA Manager 09 |411.0 |
|Perpetuum |400.97 |
|Football Manager 2014 |391.98 |
|Football Manager 2012 |390.45 |
|Football Manager 2010 |375.05 |
|Football Manager 2011 |365.7 |
|Freaking Meatbags |331.0 |
|Out of the Park Baseball 16 |330.4 |
+-----+-----+

Command took 2.98 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk
```

**Figure 2.2.6** The code filters the DataFrame `steam_dataframe` to show **average** user game play behavior. It then **groups** game records and uses the '**avg**' aggregation function to calculate average playtime length for each game, aliased as '**Avg PlayTimeIN\_HOUR**'. The data is **sorted** by average playtime, presenting the **top 10** games with the highest average playtime per user, providing insights into user habits.

How many users have played more than one game

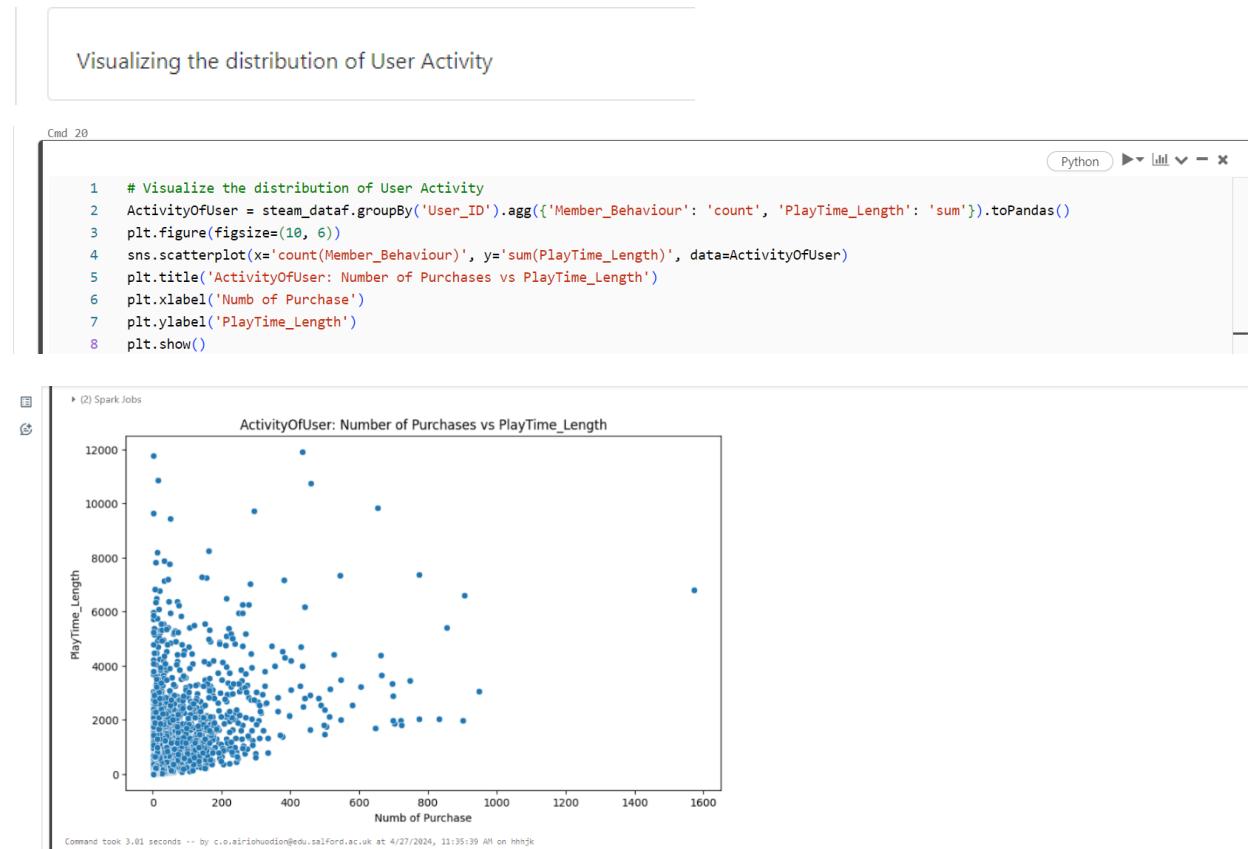
```
Cmd 19
1 #How many users have played more than one game
2 steam_dataframe.filter(steam_dataframe["Member_Behaviour"] == "play") \
3 | groupBy("User_ID") \
4 | .agg(countDistinct("Game").alias("Games Played")) \
5 | .filter(col("Games Played") > 1).count()

▶ (4) Spark Jobs
4791

Command took 3.48 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk
```

**Figure 2.2.7** This code calculates the number of people who have played many games

using the DataFrame **steam\_dataframe**. It filters the data to show play behavior, categorizes it using **User\_ID**, and uses the **countDistinct** aggregation function to determine unique games played. The report filters aggregated data to include users who have played multiple games and uses the count function to determine their level of involvement with the site.



**Figure 2.2.8** This code visualizes user activity in the Steam dataset by grouping data by “User\_ID”, adding total “PlayTime\_Behaviour” for each user, and generating a scatter plot with Seaborn, revealing gameplay duration and purchase quantity.

```

Cmd 21
1 # Average play time for each user in hours
2 avg_playtime_per_user = steam_dataframe.filter(steam_dataframe["Member_Behaviour"] == "play") \
3                                         .groupBy("User_ID") \
4                                         .agg(avg("PlayTime_Length").alias("Avg Play Time")) \
5                                         .limit(10)
6 avg_playtime_per_user.show(truncate=False)
7

(2) Spark Jobs
avg_playtime_per_user: pyspark.sql.dataframe.DataFrame = [User_ID: integer, Avg Play Time: double]
+-----+
|User_ID |Avg Play Time |
+-----+
|16167221 |40.679999999403954 |
|166705920|697.0   |
|244878837|4.0    |
|99992274 |6.300000190734863 |
|174415183|8.53333330154419 |
|156156544|5.300000190734863 |
|152861732|72.65000008543332 |
|171911285|32.0   |
|128412180|1.2000000476837158 |
|74557142 |0.1000000149011612|
+-----+

```

Command took 2.75 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhjk

**Figure 2.2.9** This code calculates the average play time of users based on their gaming activity using the DataFrame **steam\_dataframe**. The DataFrame is filtered to show play behavior, categorized by **User\_ID**, and averaged using the ‘**avg**’ aggregation function on the ‘**PlayTime\_Length**’ column. The ‘**Avg Play Time**’ is an alias for the average play time. The limit function restricts to top 10 users with highest average play time, while the show function displays the data, providing insight into gaming habits.

## Data Pre-Processing and Preparation

```

Cmd 23
Data Pre-Processing and Preparation
Python ▶▼◀ - ×

1 #Preparing data for Machine Learning (ML) training
2 # Generating the GameID
3 game_id_dataframe = steam_dataframe.select("Game").distinct().withColumn('GameID', monotonically_increasing_id()).withColumnRenamed("Game", "Game_")
4 steam_dataframe_id = game_id_dataframe.join(steam_dataframe, game_id_dataframe["Game_"] == steam_dataframe["Game"]).drop("Game_")

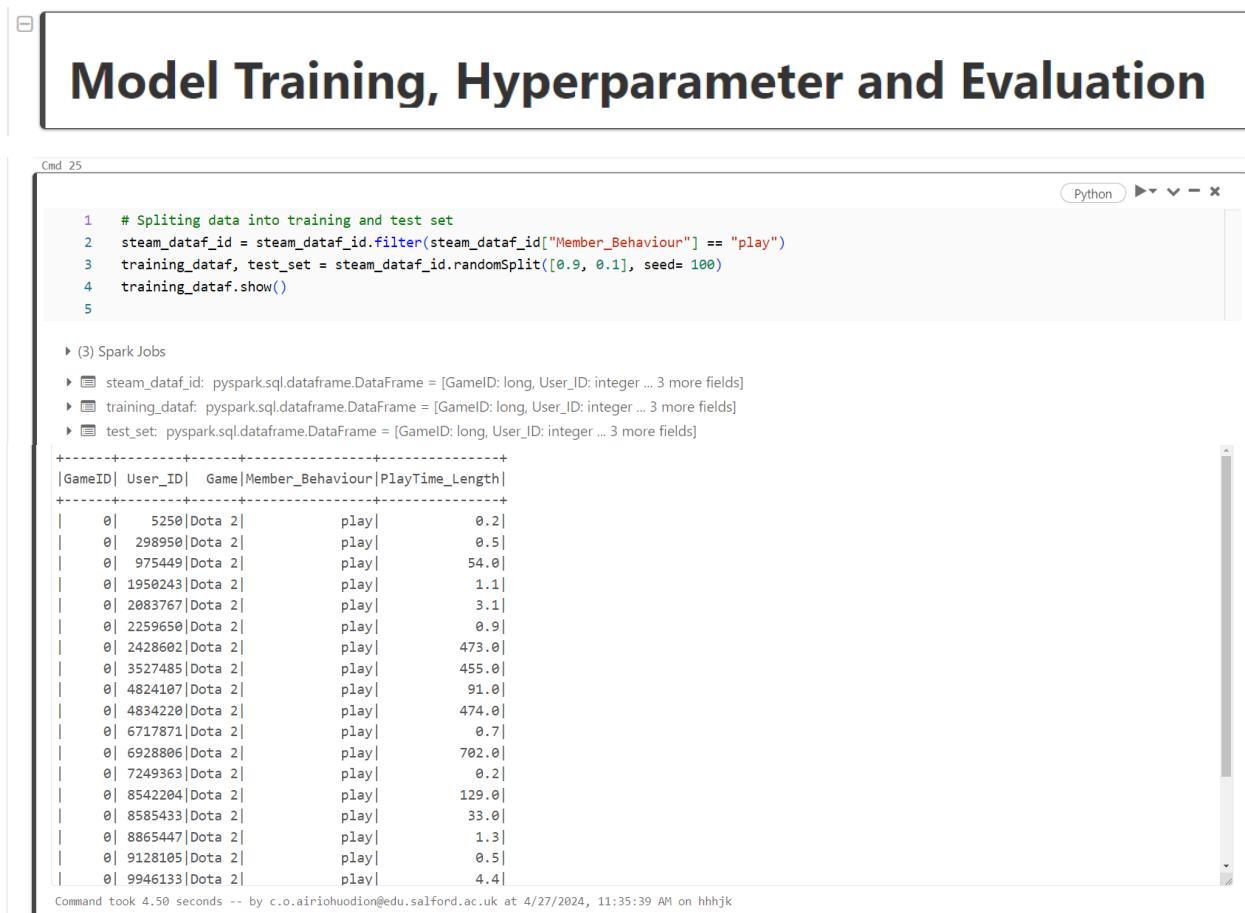
game_id_dataframe: pyspark.sql.dataframe.DataFrame = [Game_: string, GameID: long]
steam_dataframe_id: pyspark.sql.dataframe.DataFrame = [GameID: long, User_ID: integer ... 3 more fields]

Command took 0.49 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhjk

```

**Figure 2.3.0** By selecting distinct games, assigning GameIDs using ‘**monotonically\_increase\_id()**’, joining with the original dataframe based on the ‘**Game**’

column, and removing unnecessary columns, the code creates a unique identifier (**GameID**) for each game in the **steam\_dataframe** dataset in preparation for machine learning (ML) training. The end result is the **steam\_dataframe\_id** DataFrame.



The screenshot shows a Jupyter Notebook cell with the following code:

```
1 # Splitting data into training and test set
2 steam_dataaf_id = steam_dataaf_id.filter(steam_dataaf_id["Member_Behaviour"] == "play")
3 training_dataaf, test_set = steam_dataaf_id.randomSplit([0.9, 0.1], seed= 100)
4 training_dataaf.show()
```

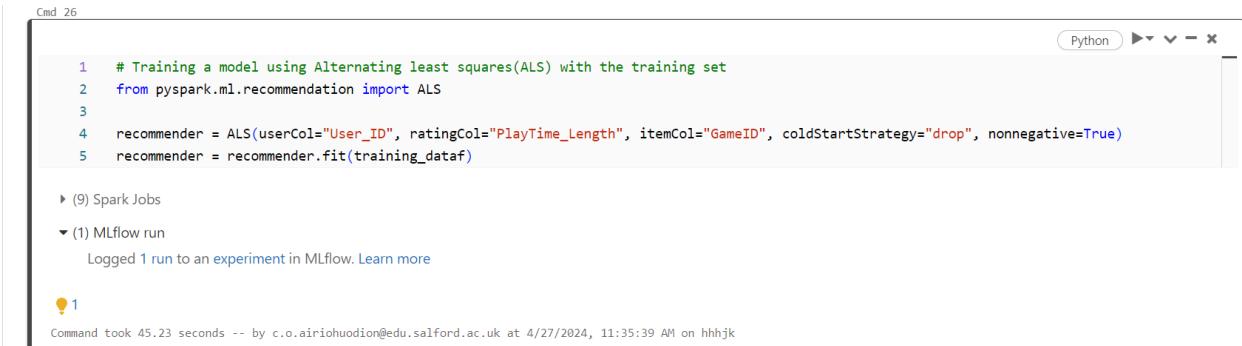
Below the code, the output shows the first few rows of the `training_dataaf` DataFrame:

GameID	User_ID	Game	Member_Behaviour	PlayTime_Length
0  5250 Dota 2  play  0.2				
0  2988950 Dota 2  play  0.5				
0  975449 Dota 2  play  54.0				
0  1950243 Dota 2  play  1.1				
0  2083767 Dota 2  play  3.1				
0  2259650 Dota 2  play  0.9				
0  2428602 Dota 2  play  473.0				
0  3527485 Dota 2  play  455.0				
0  4824107 Dota 2  play  91.0				
0  4834220 Dota 2  play  474.0				
0  6717871 Dota 2  play  0.7				
0  6928806 Dota 2  play  702.0				
0  7249363 Dota 2  play  0.2				
0  8542204 Dota 2  play  129.0				
0  8585433 Dota 2  play  33.0				
0  8865447 Dota 2  play  1.3				
0  9128105 Dota 2  play  0.5				
0  9946133 Dota 2  play  4.4				

Command took 4.50 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhjk

**Figure 2.3.1** This section of code divides the data into test and training sets for the purpose of machine learning. First, only rows with ‘play’ selected in the ‘**Member\_Behaviour**’ column of the **steam\_dataframe\_id** DataFrame are included. Next, it splits the filtered DataFrame into the **trainning\_set** (which contains 90% of the data) and the **test\_set** (10% of the data) using the **randomSplit** function. Using the **show()** method, it finally shows the first few rows of the **trainning\_set** DataFrame.

Training a model using Alternating least squares(ALS) with the training set



```
Cmd 26
```

```
1 # Training a model using Alternating least squares(ALS) with the training set
2 from pyspark.ml.recommendation import ALS
3
4 recommender = ALS(userCol="User_ID", ratingCol="PlayTime_Length", itemCol="GameID", coldStartStrategy="drop", nonnegative=True)
5 recommender = recommender.fit(training_data)
```

▶ (9) Spark Jobs  
▼ (1) MLflow run  
Logged 1 run to an experiment in MLflow. Learn more

💡 1

Command took 45.23 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.3.2** This code uses the training set data to train an Alternating Least Squares (**ALS**) recommendation model. The **ALS** class from the **pyspark.ml.recommendation** module. The parameters that have been provided are;

**userCol:** The User\_ID's column name.

**ratingCol:** The name of the play time or ratings column.

**itemCol:** The item or game ID column name.

**coldStartStrategy:** this strategy is used to tackle cold-start problems, it is configured to 'drop' any rows that contain **NaN** values which stands for 'Not a Number' it is a place holder value that is meant to stand in for unclear or missing data.

**nonnegative:** indicates if non-negativity requirements should be applied to the factors. The training set (**trainning\_set**) is used to train the model using the '**fit()**' function after it has been define. The variable called **recommender** contains the learned model.

Prediction Generated by Recommender Model

Cmd 27

Python ► ▾ - x

```
1 #Testing the recommendation system by Prediction Generated by Recommender Model
2
3 predicts = recommender.transform(test_set)
4 predicts.show(300)
```

▶ (5) Spark Jobs

▶ pyspark.sql.dataframe.DataFrame = [GameID: integer, User\_ID: integer ... 4 more fields]

GameID	User_ID	Member_Behaviour	PlayTime_Length	prediction	
0	1612666	Data 2	play	7.4	7.885475
0	7431946	Data 2	play	17.8	63.999695
0	11161178	Data 2	play	256.0	1.8151823
0	20207081	Data 2	play	4845.0	45.045086
0	21061921	Data 2	play	41.0	153.13756
0	23672423	Data 2	play	0.5	44.625328
0	32498610	Data 2	play	1.7	457.64517
0	33865373	Data 2	play	7.9	526.73724
0	45974860	Data 2	play	4.5	1581.3497
0	49724738	Data 2	play	10.5	4.5551653

Command took 10.80 seconds -- by c.oairiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

**Figure 2.3.3** By forecasting the length of the game and comparing it to the actual play time, this code checks the recommendation system. It uses the `transform()` function to apply the trained recommendation model (`recommender`) to the test set (`test_set`), producing predictions based on the model. The `preds` DataFrame contains the forecasts.

Next, the first 300 rows of the ‘`preds`’ DataFrame are displayed using the `show()` function, displaying the actual play time for each game as well as the forecast play time.

### Evaluating the model by computing the RMSE

```
Cmd 28
```

```
1 # Evaluate the model by computing the RMSE
2 from pyspark.ml.evaluation import RegressionEvaluator
3
4 evaluator = RegressionEvaluator(metricName="rmse", labelCol="PlayTime_Length", predictionCol="prediction")
5 rmse = evaluator.evaluate(predicts)
6 print("Test data's Root Mean Squared Error (RMSE) =", rmse)
```

▶ (7) Spark Jobs

```
Test data's Root Mean Squared Error (RMSE) = 102.17969922533806
```

```
Command took 21.24 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/29/2024, 6:10:00 PM on dhfahkfa
```

**Figure 2.3.4** The recommender model's performance is evaluated by calculating the Root Mean Squared Error (RMSE) on test data to measure its accuracy in predicting user playtime duration.

### Testing the recommendation system on a single user

```
Cmd 29
```

```
1 # Testing the recommendation system on a single user
2
3 single_user = steam_dataframe.filter(steam_dataframe["User_ID"] == 32125590)
4 single_recommendation = recommender.transform(single_user)
5 single_recommendation.show()
```

▶ (5) Spark Jobs

```
▶ [single_user: pyspark.sql.dataframe.DataFrame = [GameID: long, User_ID: integer ... 3 more fields]
▶ [single_recommendation: pyspark.sql.dataframe.DataFrame = [GameID: integer, User_ID: integer ... 4 more fields]
```

GameID	User_ID	Game	Member_Behaviour	PlayTime_Length	prediction
0	32125590	Dota 2	play	2.1	2.1902604
357	32125590	Real Warfare	play	0.1	0.10405182
511	32125590	Saints Row The Third	play	16.0	15.892562
560	32125590	Company of Heroes 2	play	18.7	18.573105
886	32125590	Men of War Vietnam	play	0.5	0.7354553
1971	32125590	Deathmatch Classic	play	0.1	1.0321039
3054	32125590	Men of War Red Tide	play	0.9	0.0019004201
3243	32125590	Theatre of War	play	0.2	0.19466317
3605	32125590	Counter-Strike Co...	play	5.8	5.792397
3689	32125590	Counter-Strike	play	33.0	24.63085

```
Command took 7.87 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk
```

**Figure 2.3.5** The 'show()' method is used to display the recommendations for the single user after the recommendation system is tested on a single user with the User ID {32125590}. The 'steam\_dataframe\_id' DataFrame is filtered to select only the rows where the User ID is '2'.

Hyperparameter Tuning with ALS Model

```

Cmd 31 Python ▶ v - x

1  from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
2  from pyspark.ml.recommendation import ALS
3  from pyspark.ml.evaluation import RegressionEvaluator
4
5  # ALS model definition
6  als = ALS(userCol="User_ID", itemCol="GameID", ratingCol="PlayTime_Length", coldStartStrategy="drop")
7
8  # Definition of parameter grid for hyperparameter tuning
9  param_grid = ParamGridBuilder() \
10    .addGrid(als.rank, [5, 10, 15]) \
11    .addGrid(als.regParam, [0.01, 0.1, 0.5]) \
12    .build()
13
14  # Define evaluator
15  evaluator = RegressionEvaluator(metricName="rmse", labelCol="PlayTime_Length", predictionCol="prediction")
16
17  # Define cross-validator
18  cross_validator = CrossValidator(estimator=als,
19    estimatorParamMaps=param_grid,
20    evaluator=evaluator,
21    numFolds=5) # Use 5 folds for cross-validation
22
23  # Perform cross-validation and hyperparameter tuning
24  cv_model = cross_validator.fit(training_data)
25
26  # Get the best model from cross-validation
27  best_model = cv_model.bestModel
28
29  # Make predictions on the test data using the best model
30  predictions = best_model.transform(test_set)
31
32  # Evaluate the performance of the best model
33  rmse = evaluator.evaluate(predictions)

34  print("Root Mean Squared Error (RMSE) on test data after hyperparameter tuning:", rmse)
35  print("Best Rank:", best_model.rank)
36  print("Best Regularization Parameter:", best_model._java_obj.parent().getRegParam())

▶ (5) Spark Jobs
▼ (10) MLflow runs
  Logged 10 runs to an experiment in MLflow. Learn more
  ➔ predictions: pyspark.sql.dataframe.DataFrame = [GameID: integer, User_ID: integer ... 4 more fields]
  Root Mean Squared Error (RMSE) on test data after hyperparameter tuning: 245.31185931617435
  Best Rank: 15
  Best Regularization Parameter: 0.5
  Command took 23.92 minutes -- by c.o.airiohuodion@edu.salford.ac.uk at 4/29/2024, 6:43:39 PM on dhfahkfa

```

**Figure 2.3.6** Cross-validation is used to adjust the ALS model by experimenting with various hyperparameter combinations. Based on how well the model performs as determined by the test data's root mean squared error (RMSE) measure, the optimal model is chosen. The RMSE score is provided together with the chosen hyperparameters (rank and regularization parameter) for the best model.

Average predicted play time for all games using Recom System

```

Cmd 31 Python ▶ v - x

1  #Testing the recommendation system by finding the average predicted play time for all games?
2  avg_play_time = predicts.select(avg("prediction")).collect()[0][0]
3  print("Average Predicted Play Time:", avg_play_time)

▶ (6) Spark Jobs
Average Predicted Play Time: 50.81952269711382
Command took 8.56 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:39:22 AM on hhhjk

```

**Figure 2.3.7** This average projected play time for each game suggested in the 'preds' DataFrame is computed by this code to test the recommendation system. The 'avg()' function is used to calculate the average of the predicted play times.

function is used to compute the average, gathers the result, and prints out the average anticipated play time after selecting the **prediction** column from the '**preds**' DataFrame.

## CONCLUSION

Machine learning methodologies have revolutionized various fields by evaluating complex data, identifying patterns, and generating precise predictions. Techniques like evaluation, hyperparameter tuning, model training, and data preprocessing are now understood. As techniques improve, they can tackle real-world problems and drive innovation.