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

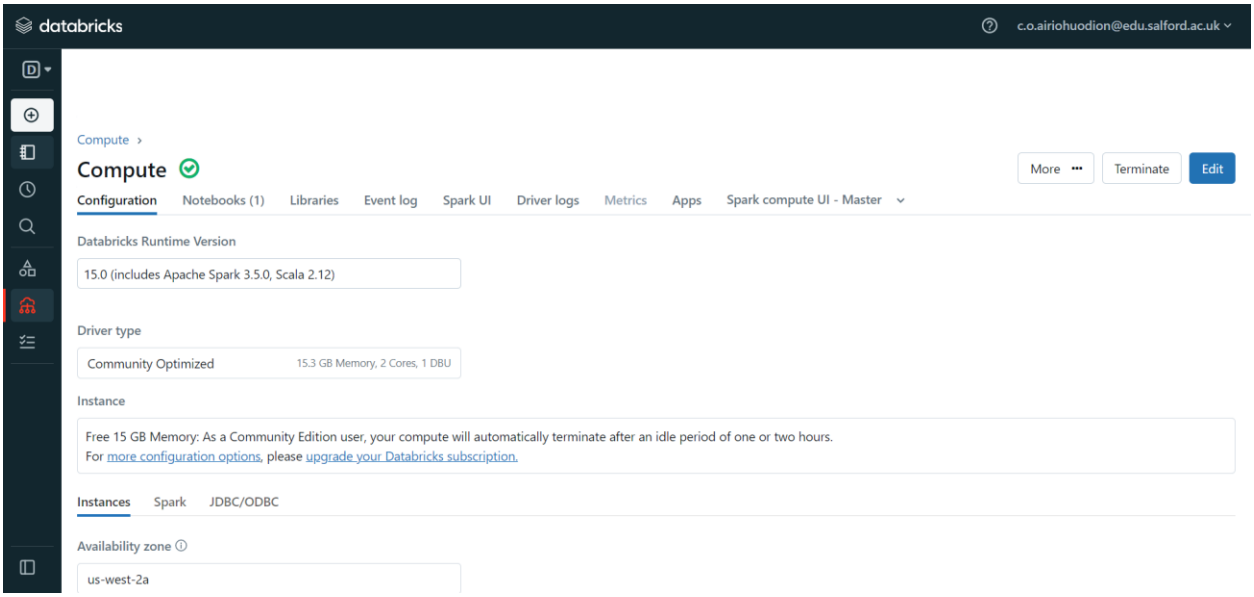


Figure 1.1.0 Setting up the platform required to kick start the task given

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).

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.

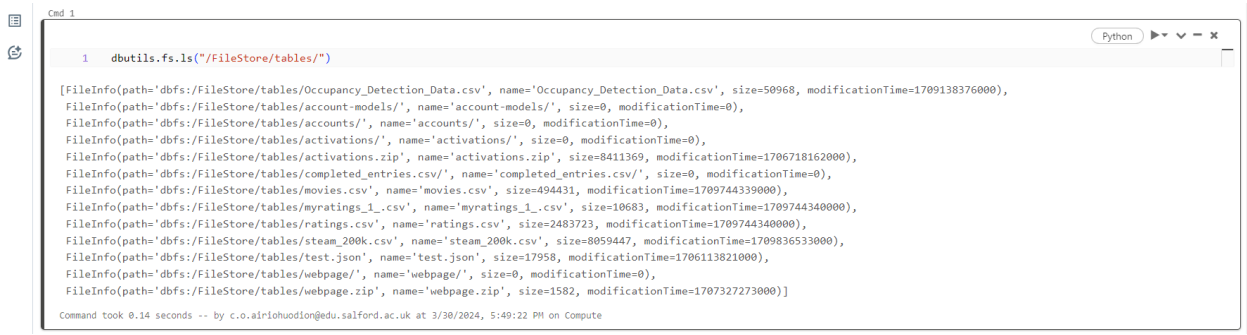


Figure 1.1.2 Using the dbutils.fs.ls() command, generally listing the files and folders in the "tables" directory inside the "FileStore" directory.

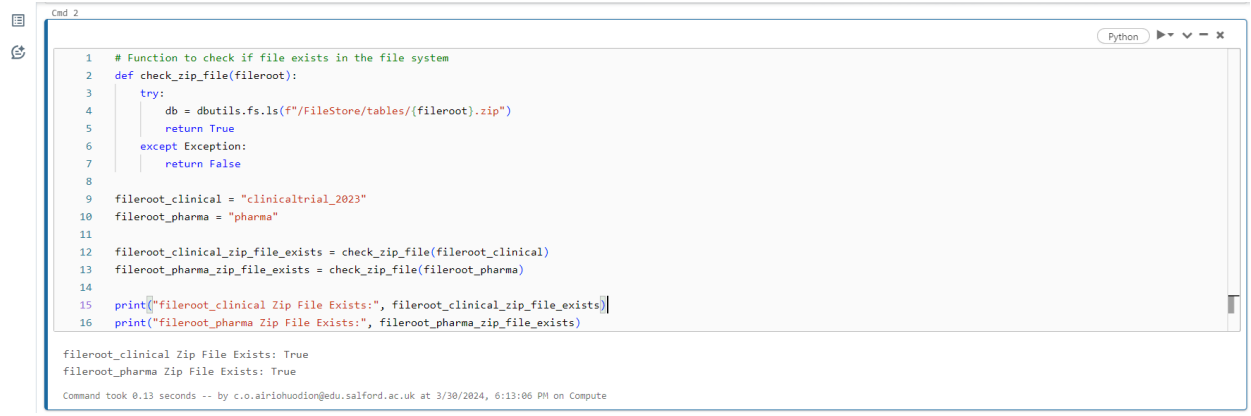


Figure 1.1.3 Creating and Using a function to check specific files (clinicantrial_2023 and pharma file) respectively.

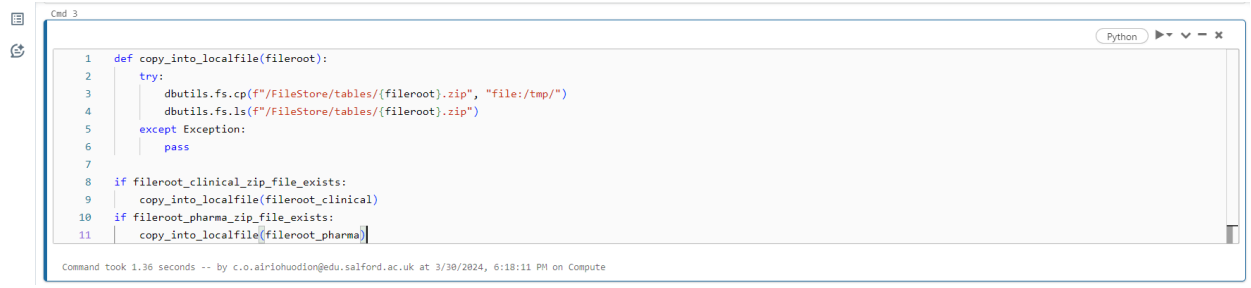


Figure 1.1.4 This defined a function called **copy_into_localfile()** in the supplied code, with **dbutils.fs.cp** it copies the two ZIP files from the **FileStore** directory to the **local file system** directory **/temp/.** It then displays the contents of the transferred ZIP file if the process is 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.

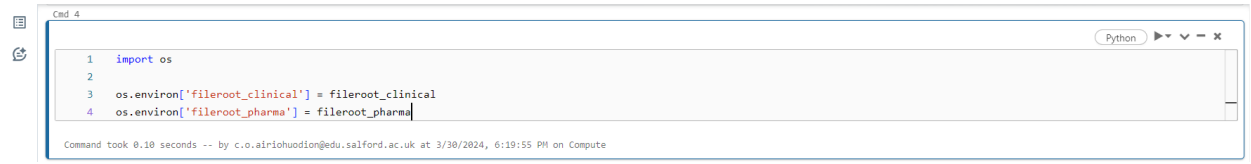


Figure 1.1.5 Knowing that the local filesystem or shell command line does not automatically know about the "**fileroot**" variable, we set the environment variables "**fileroot_clinical**" and "**fileroot_pharma**" to the variables. We set this value as an environment variable so that it maybe accessed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

Figure 1.1.6 This "ls" shell command above checks the **/tmp/** to confirm that **clinical_trail.zip** and **pharma.zip** was moved successfully.

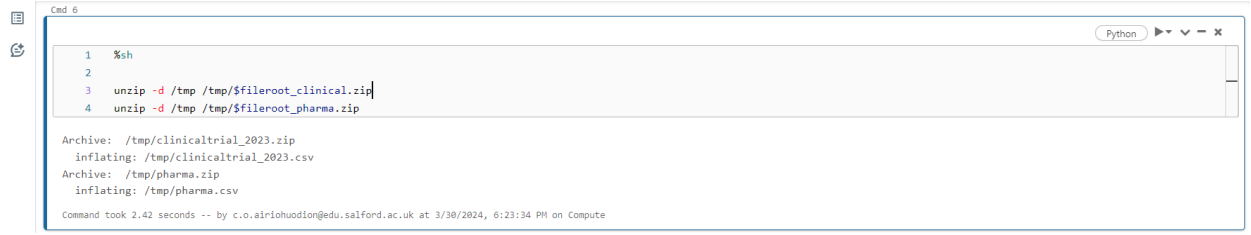


Figure 1.1.7 This code above Unzips the two .zip files

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.

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.

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.

```
Cmd 9
Python

def remove_f_dir(fileroot):
    try:
        dbutils.fs.rm(f"/FileStore/tables/{fileroot}.zip")
    except Exception:
        pass

remove_f_dir(fileroot_clinical)
remove_f_dir(fileroot_pharma)

Command took 0.30 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 12:41:04 PM on Compute
```

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.

```
Cmd 10
Python

dbutils.fs.head("/FileStore/tables/" + fileroot_clinical + ".csv")

[Truncated to first 65536 bytes]
'"Id\tStudy Title\tAcronym\tStatus\tConditions\tInterventions\tSponsor\tCollaborators\tEnrollment\tFunder Type\tType\tStudy Design\tStart\tCompletio
n",,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,,,,,\r\n"NCT03630471\tEffectiveness of a Problem-solving Intervention for Common Adolescent Mental Health Problems in India\tPRIDE\tCOMPLETED\tMental Health I
ssue (E.G.", Depression, Psychosis, Personality Disorder," Substance Abuse)\tBEHAVIORAL: PRIDE \'Step 1\' problem-solving intervention|BEHAVIORAL: Enhanced usual care\tSangath\tHarvar
d Medical School (HMS and HSDM)|London School of Hygiene and Tropical Medicine\t250.0\tOTHER\tINTERVENTIONAL\tAllocation: RANDOMIZED|Intervention Model: PARALLEL|Masking: DOUBLE (INVE
STIGATOR"," OUTCOMES_ASSESSOR)|Primary Purpose: TREATMENT\t2018-08-20\t2019-02-2
8",,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,\r\n"NCT05992571\tOral Ketone Monoester Supplementation and Resting-state Brain Connectivity\t\tRECRUITING\tCerebrovascular Function|Cognition\tOTHER: Placebo|
DIETARY_SUPPLEMENT: β-OHB\tMcMaster University\tAlzheimer\'s Society of Brant", Haldimand Norfolk," Hamilton Halton\t30.0\tOTHER\tINTERVENTIONAL\tAllocation: RANDOMIZED|Intervention M
odel: CROSSOVER|Masking: TRIPLE (PARTICIPANT", INVESTIGATOR," OUTCOMES_ASSESSOR)|Primary Purpose: BASIC_SCIENCE\t2023-10-25\t2024-0
8",,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,,,,,,\r\n"NCT00237471\tImpact of Tight Glycaemic Control in Acute Myocardial Infarction\t\tTERMINATED\tMyocardial Infarct|Hyperglycemia\tDRUG: Insulin (tight blood

Command took 0.37 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 3/30/2024, 6:36:16 PM on Compute
```

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

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.

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.

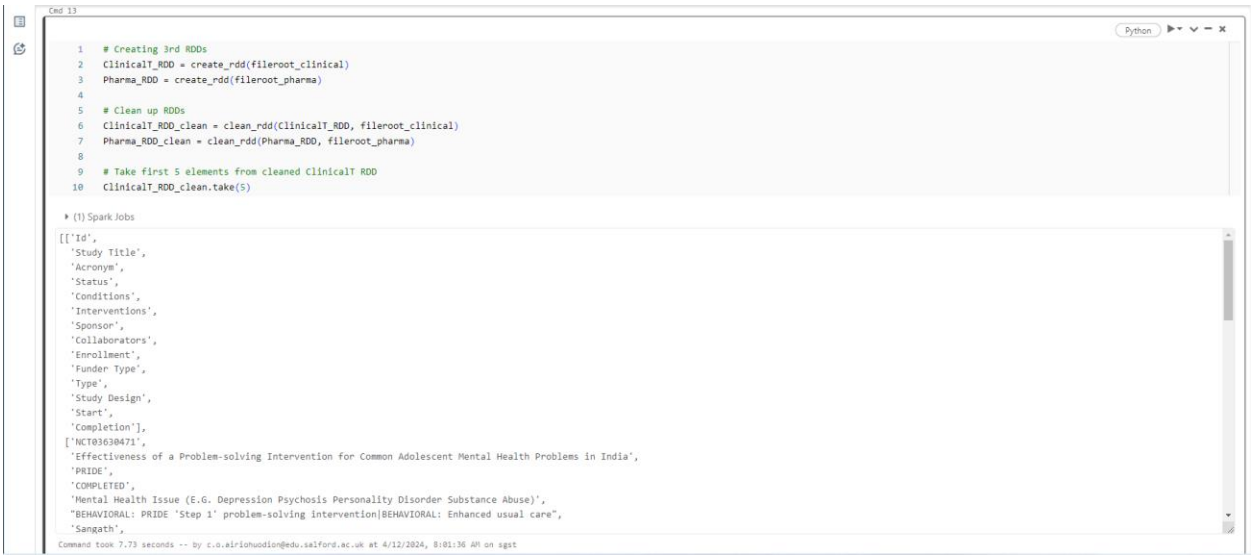


Figure 1.2.4 The create_rdd() function creates two RDDs, ClinicalT_RDD and Pharma_RDD, representing raw data from pharma files and clinical trials. The clean_rdd() method cleans the raw RDDs, dividing records within each RDD. The take(5) action extracts the first items from the cleaned ClinicalT_RDD,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

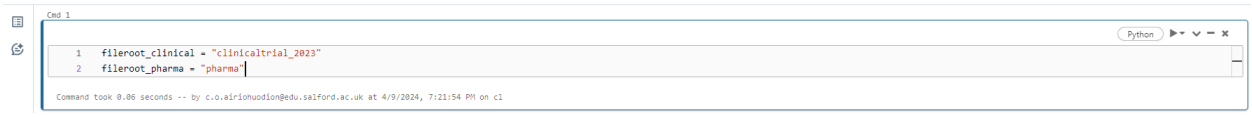


Figure 1.2.5 My file roots are file_pharma and fileroot_clinical. The names of the CSV files containing pharmaceutical and clinical trial 2020, 2021 and 2023 data respectively, are held in these variables

```
Cmd 2

#Creating DataFrame
from pyspark.sql.types import * #StructType, StructField
from pyspark.sql.functions import *

delimiter_selector = {
    "clinicaltrial_2023": "\t",
    "clinicaltrial_2021": "|",
    "clinicaltrial_2020": "|",
    "pharma": ","
}

def create_clinical_dataframe(fileroot):
    if fileroot == "clinicaltrial_2023":
        rdd = sc.textFile(f"/FileStore/tables/{fileroot}.csv").map(lambda row: row.split(delimiter_selector[fileroot]))
        head = rdd.first()
        rdd = rdd.map(lambda row: row + [" " for i in range(len(head) - len(row))] if len(row) < len(head) else row )
        df = rdd.toDF()
        first = df.first()
        for col in range(0, len(list(first))):
            df = df.withColumnRenamed(f"_{col + 1}", list(first)[col].replace(",", "").replace('"', ""))
        df = df.withColumn('index', monotonically_increasing_id())
        df = df.withColumn('Completion', regexp_replace("Completion", ",", "")).withColumn('Completion', regexp_replace("Completion", '"', ""))
        return df.filter(~df.index.isin([0])).drop('index')
    else:
        return spark.read.csv(f"/FileStore/tables/{fileroot}.csv", sep=delimiter_selector[fileroot], header=True)

ClinicalT_dataframe = create_clinical_dataframe(fileroot_clinical)
pharma_dataframe = create_clinical_dataframe(fileroot_pharma)
ClinicalT_dataframe.show(20)
```

Figure 1.2.6 The code imports Modules to define schema and functions for DataFrame operations. It introduces a **delimeter_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.

```
(5) Spark Jobs
ClinicalT_dataframe:  pyspark.sql.dataframe.DataFrame = [Id: string, Study Title: string ... 12 more fields]
pharma_dataframe:  pyspark.sql.dataframe.DataFrame = [Company: string, Parent_Company: string ... 32 more fields]
+-----------+-------------------+---------+-----------------+------------------+------------------+------------------+------------------+----------+-----------+----------
---+-------------------+---------+----------+
|         Id|        Study Title|  Acronym|           Status|        Conditions|     Interventions|           Sponsor|     Collaborators|Enrollment|Funder Type|         T
ype|       Study Design|    Start|Completion|
+-----------+-------------------+---------+-----------------+------------------+------------------+------------------+------------------+----------+-----------+----------
---+-------------------+---------+----------+
|"NCT03630471|Effectiveness of ...|    PRIDE|        COMPLETED|Mental Health Iss...|BEHAVIORAL: PRIDE...|           Sangath|Harvard Medical S...|     250.0|      OTHER|INTERVENTIO
NAL|Allocation: RANDO...|2018-08-20|2019-02-28|
|"NCT05992571|Oral Ketone Monoe...|         |       RECRUITING|Cerebrovascular F...|OTHER: Placebo|DI...| McMaster University|Alzheimer's Socie...|      30.0|      OTHER|INTERVENTIO
NAL|Allocation: RANDO...|2023-10-25|   2024-08|
|"NCT00237471|Impact of Tight G...|         |       TERMINATED|Myocardial Infarc...|DRUG: Insulin (ti...|    Melbourne Health|National Health a...|      40.0|      OTHER|INTERVENTIO
NAL|Allocation: RANDO...|   2005-10|   2006-05|
|"NCT03820271|New Prognostic Pr...| SUPERMELD|       RECRUITING|Decompensated Cir...|    OTHER: SuperMELD|Assistance Publiq...|                    |     500.0|      OTHER|INTERVENTIO
NAL|Allocation: NA|In...|2020-10-01|2023-10-01|
|"NCT06229171|InTake Care: Deve...|InTakeCare|NOT_YET_RECRUITING|Hypertension|Trea...|OTHER: adherence ...|Istituto Auxologi...|Istituti Clinici ...|     206.0|      OTHER|INTERVENTIO
NAL|Allocation: RANDO...|2024-10-01|2026-04-01|
|"NCT02945371|Tailored Inhibito...|      REV|        COMPLETED|Smoking|Alcohol D...|BEHAVIORAL: Perso...|University of Oregon|                    |     103.0|      OTHER|INTERVENTIO
NAL|Allocation: RANDO...|   2014-09|   2016-05|
|"NCT01055171|Neuromodulation o...|         |        COMPLETED|Alcohol Dependenc...|DRUG: Propranolol...|Medical Universit...|National Institut...|      44.0|      OTHER|INTERVENTIO
NAL|Allocation: RANDO...|   2010-01|   2012-08|
|"NCT01125371|Computerized Brie...|         |        COMPLETED|Alcohol; Harmful ...|BEHAVIORAL: Compu...|Johns Hopkins Uni...|National Institut...|     439.0|      OTHER|INTERVENTIO
Command took 24.87 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/10/2024, 8:52:55 PM on bzjzx
```

Figure 1.2.7 This above is the sample of the cleaned clinical_trial 2023

SQL PySpark IMPLEMENTATION

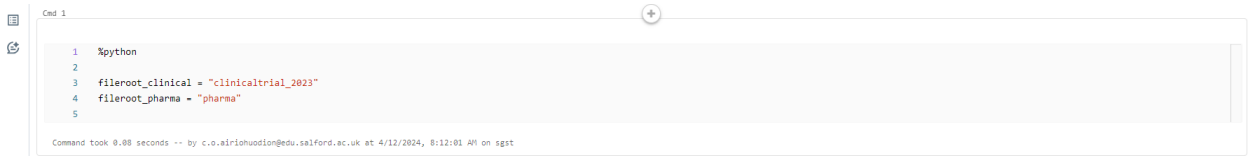


Figure 1.2.8 My file roots are file_pharma and fileroot_clinical. The names of the CSV files containing pharmaceutical and clinical trial 2020, 2021 and 2023 data respectively, are held in these variables

Figure 1.2.9 The code imports Modules to define schema and functions for DataFrame operations. It introduces a **delimeter_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.

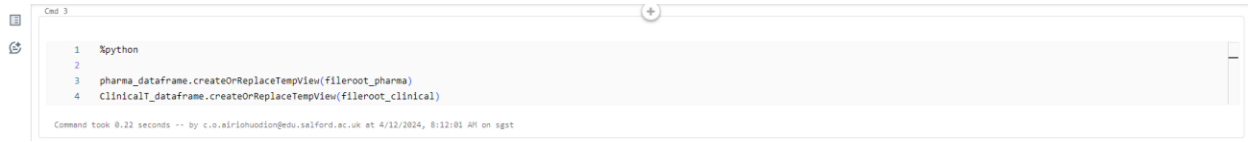


Figure 1.3.0 This code creates two temporary table (view) Pharma_dataframe and ClinicalT_dataframe. I used **createOrReplaceTempView** method.

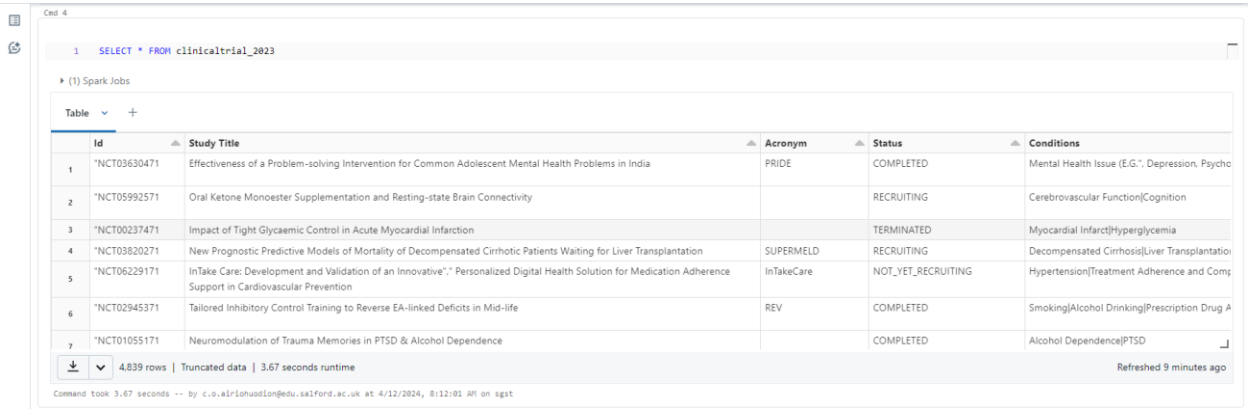


Figure 1.3.1 The complete dataset is returned by this query, giving users a thorough overview of the clinical trial data for 2023. A single clinical trial is represented by each row in the result, with columns denoting different characteristics such as trial ID, conditions, sponsors.

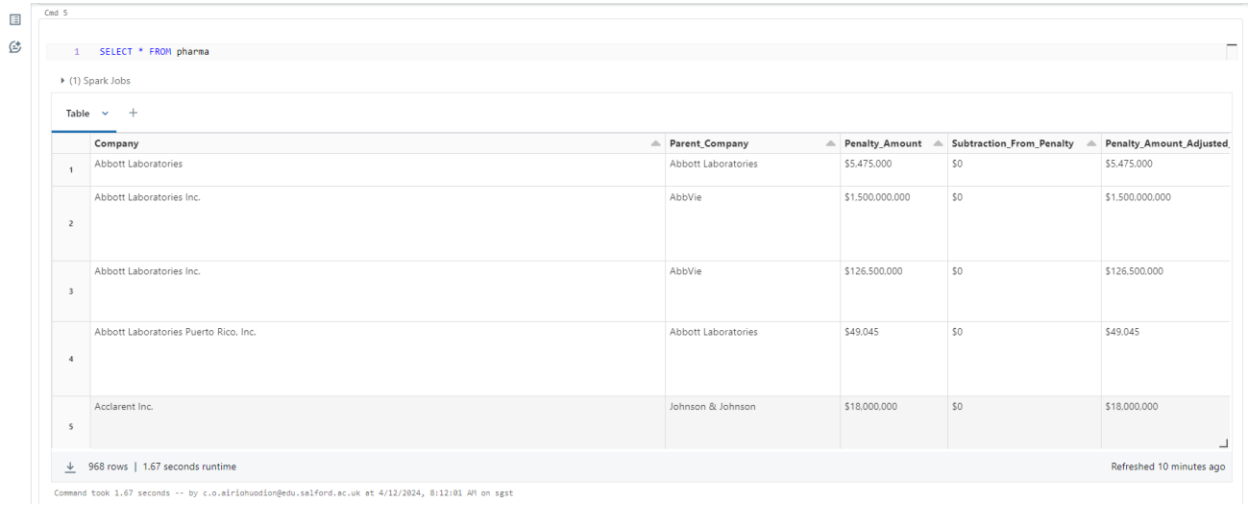


Figure 1.3.2 Every field and record in the pharma table will be retrieved by this query

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

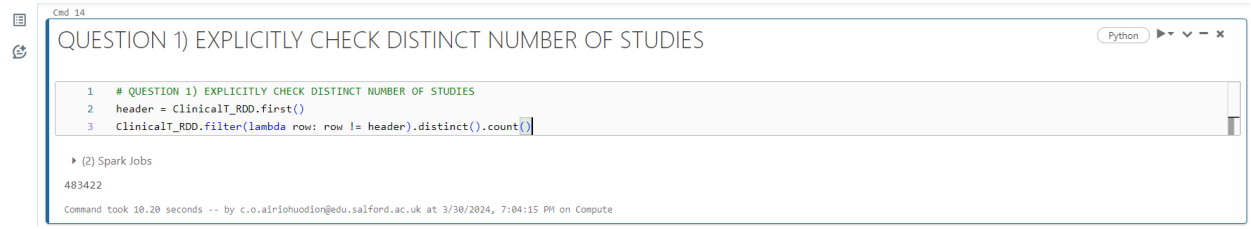


Figure 1.3.3 This code retrieves the first row of the data from ClinicalT_RDD and assigns it to the variable header. It filters the RDD ClinicalT_RDD by comparing each row to the header row using the lambda function. The distinct() function is applied to the filtered RDD to count distinct rows by removing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

DATAFRAME Implementation

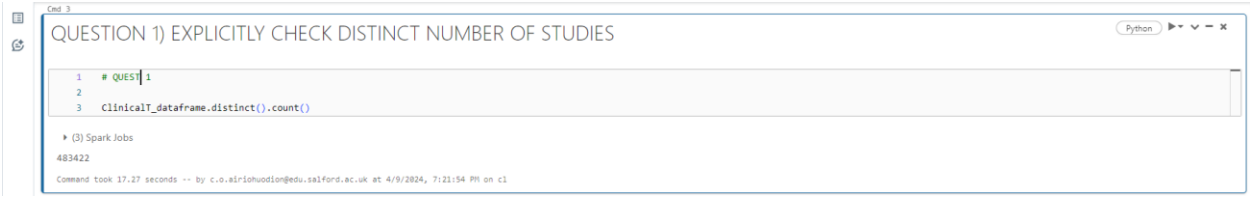


Figure 1.3.4 From the question given I counted the unique entries in the ClinicalT_dataframe using pyspark. By taking into account all columns, this count indicates the total number of unique rows in the DataFrame. Using '**Dinstinct'** to get unique entries and then '**count'** them.

SQL Implementation

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

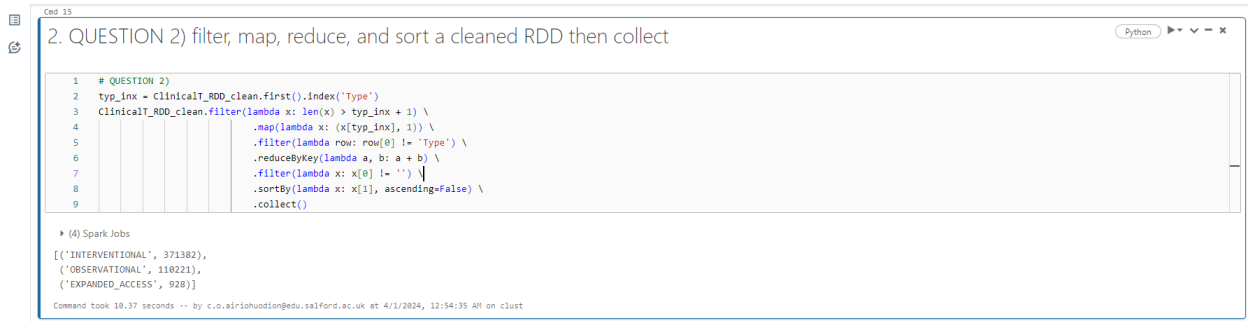


Figure 1.3.6 Using a lambda function to check for inequality with the header row, it filters out the header row from the RDD ClinicalT_RDD, which is represented by the variable header. After deleting duplicates, it uses the distinct function to procedure unique rows. Lastly, it counts the number of distinct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

DATAFRAME Implementation

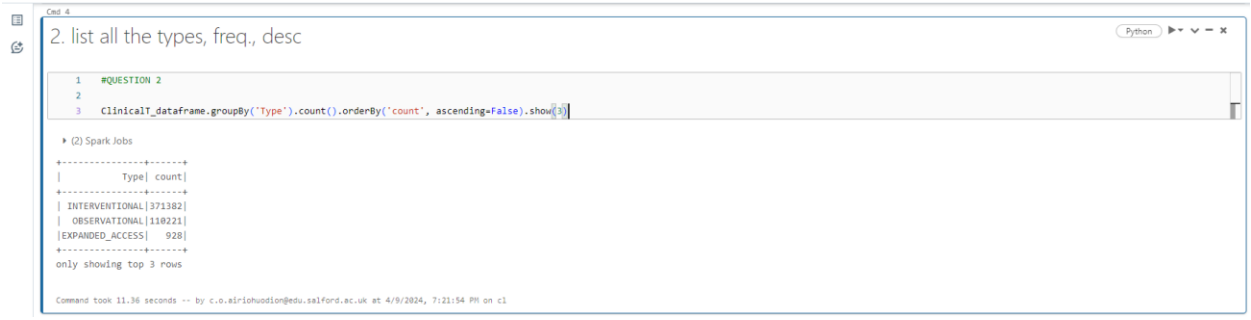


Figure 1.3.7 The pyspark above groups the entire row in the DataFrame ClinicalT_dataframe by the "Type" column using **.groupBy** then after it uses "**.count"** function to count the number of rows in each group, to further achieve the result I used the **.orderBy"** to place the data in descending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

SQL Implementation

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 delimeter depending on the particular clinical csv dataset, and is assumed to be accurately reported without any formatting issues.

RDD Implementation

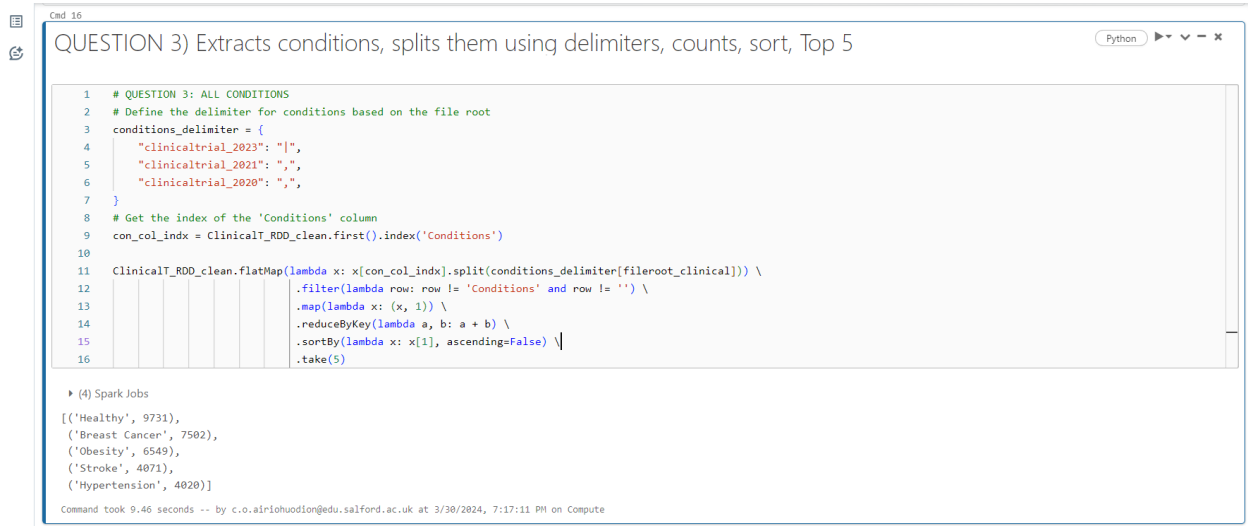


Figure 1.3.9 A dictionary called conditions_delimeter is created to assign distinct delimeters to file roots, indicating file root based division for the “Conditions” column. The index is obtained from the cleaned RDD ClinicalT_RDD_clean. RDD processing for Counting Conditions uses flatMap(), filter(), 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

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 other to achieve desired result I **filtered** out unfilled conditions, and showed the **top 5** rows without truncating the data.

SQL Implementation

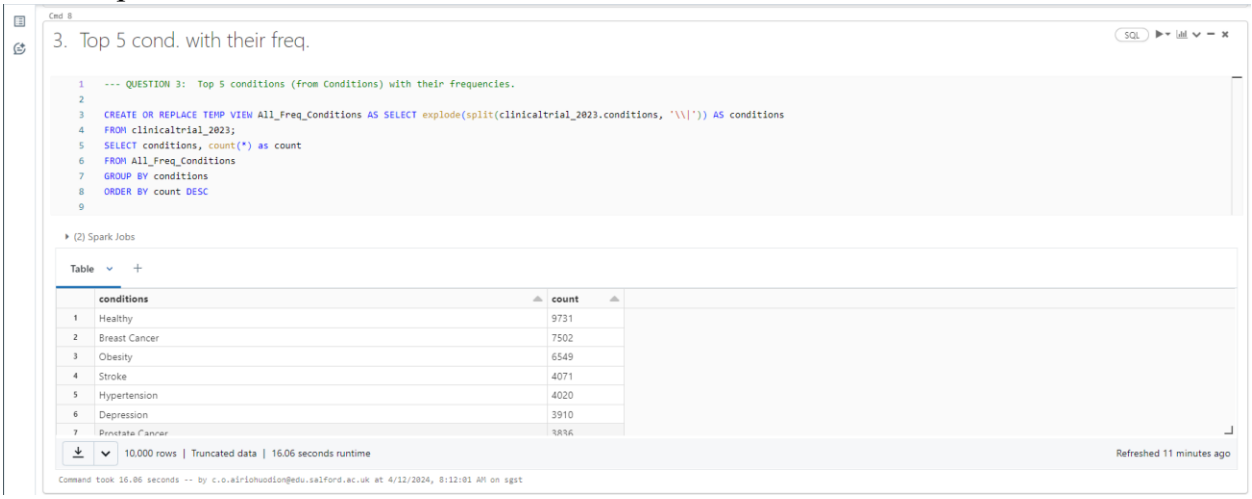


Figure 1.4.1 By exploding the conditions column, this SQL query produces a temporary view called all_conditions. It then counts the frequency of each conditions and arranges

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

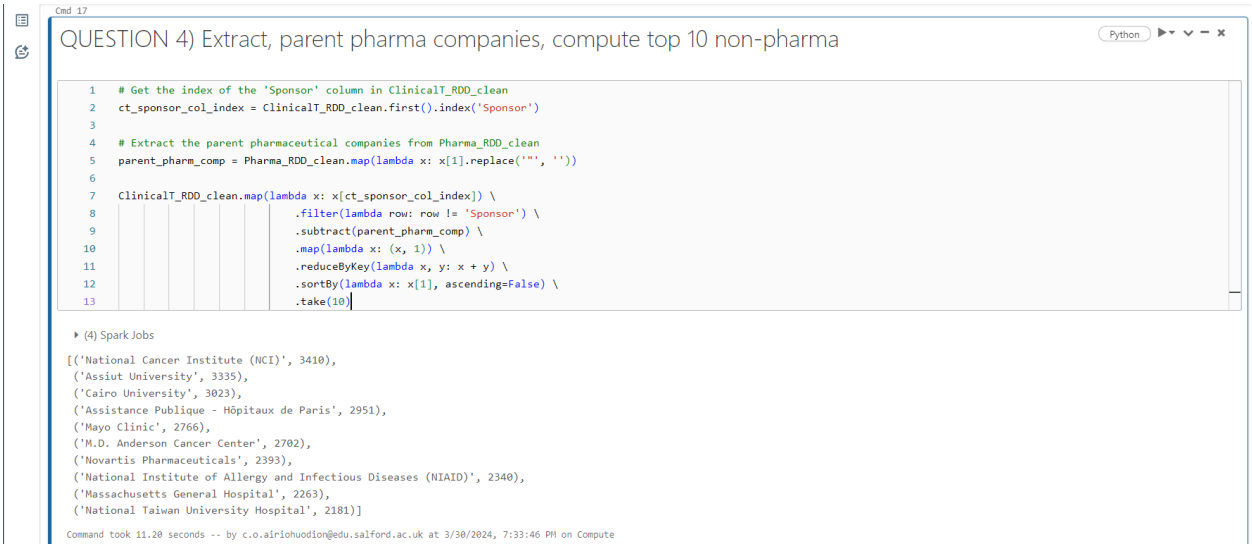


Figure 1.4.2 The RDD's 'sponsor' column index is obtained using the **index() function**. The algorithm extracts parent pharmaceutical companies from the **Pharma_RDD_clean** RDD, filters out rows with equal values, subtracts them, and **maps** each surviving "**Sponsor**" to a 1-valued tuple, then it 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

Figure 1.4.3 The first ling 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

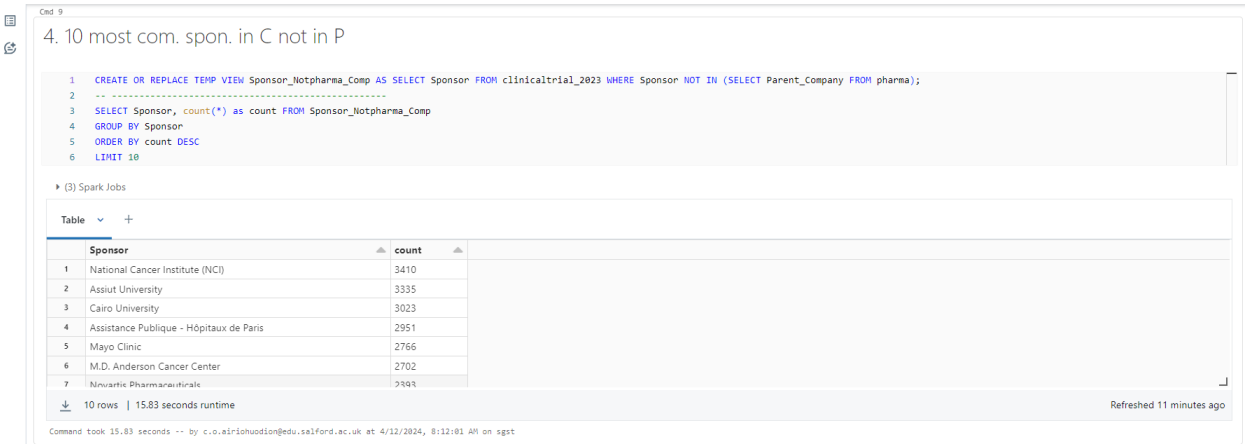


Figure 1.4.4 In order to filter sponsors from **clinicaltrial_2023** based in their absence in the pharma database, it constructs a temporary view called **Sponsor_Notpharma_Comp**. The output is then **limited** to the top 10 sponsors after **counting** the number of times each sponsor appears in the 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

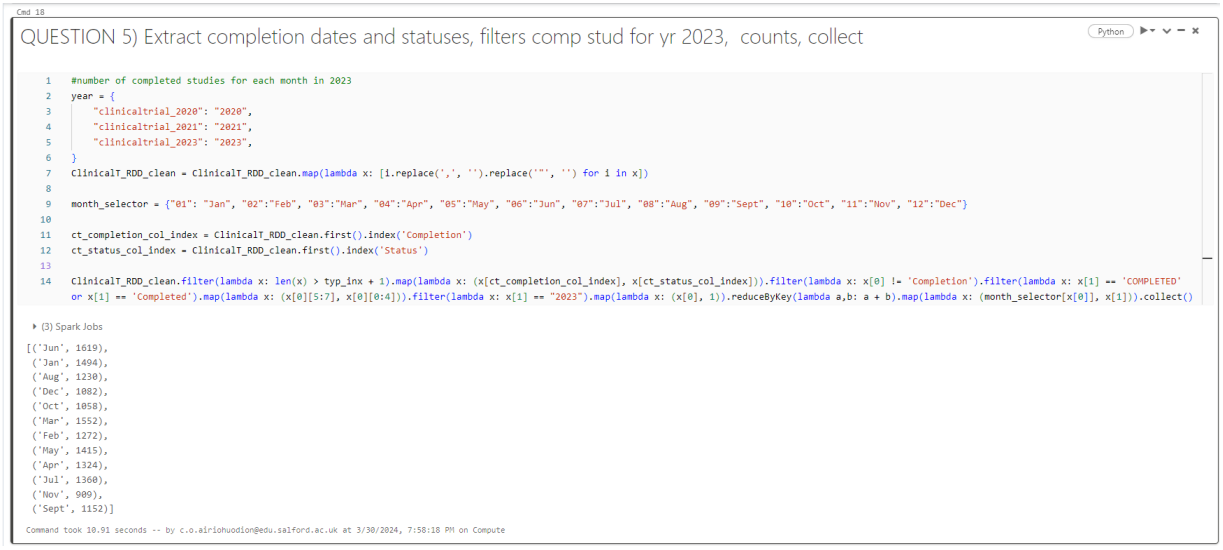


Figure 1.4.5 This code calculates the total number of studies completed each month in 2023 by mapping file names to corresponding years using a dictionary called year, eliminating commas and double quotes, and using the **month_selector** dictionary to translate month names into numerical representations. 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

Cmd 7

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

Python

```
from pyspark.sql.functions import split, regexp_replace, expr

# Apply transformations to the ClinicalT_dataframe
for col in ClinicalT_dataframe.columns:
    ClinicalT_dataframe = ClinicalT_dataframe.withColumnRenamed(col, col.strip(",").strip('"'))

# Extracting month and year from Completion column
completed_cd = ClinicalT_dataframe.withColumn('Year', split('Completion', "-")[0]) \
    .withColumn('Month', split('Completion', "-")[1]) \
    .withColumn('Month', regexp_replace("Month", ",", "")) \
    .withColumn('Month', regexp_replace("Month", '"', "")) \
    .filter(ClinicalT_dataframe.Status.isin(["COMPLETED"])) \
    .select("Month", "Year", "Status")

# Filter and aggregate to get counts for each month in 2023
month_counts = completed_cd.filter(completed_cd.Year.isin(["2023"])) \
    .groupBy("Month").count().orderBy("Month", ascending=True)

# Apply the mapping using expr to convert month numbers to month names
month_df = month_counts.withColumn("month",
                                   expr("""
                                   CASE
                                       WHEN month = '01' THEN 'Jan'
                                       WHEN month = '02' THEN 'Feb'
                                       WHEN month = '03' THEN 'Mar'
                                       WHEN month = '04' THEN 'Apr'
                                       WHEN month = '05' THEN 'May'
                                       WHEN month = '06' THEN 'June'
                                       WHEN month = '07' THEN 'July'
                                       WHEN month = '08' THEN 'Aug'
                                       WHEN month = '09' THEN 'Sep'
                                       WHEN month = '10' THEN 'Oct'
                                       WHEN month = '11' THEN 'Nov'
                                       WHEN month = '12' THEN 'Dec'
                                       ELSE 'Null'
                                   END AS month
                                   """))

# Select the month and count columns
result_df = month_df.select("month", "count")
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.airiohuodion@edu.salford.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

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**.

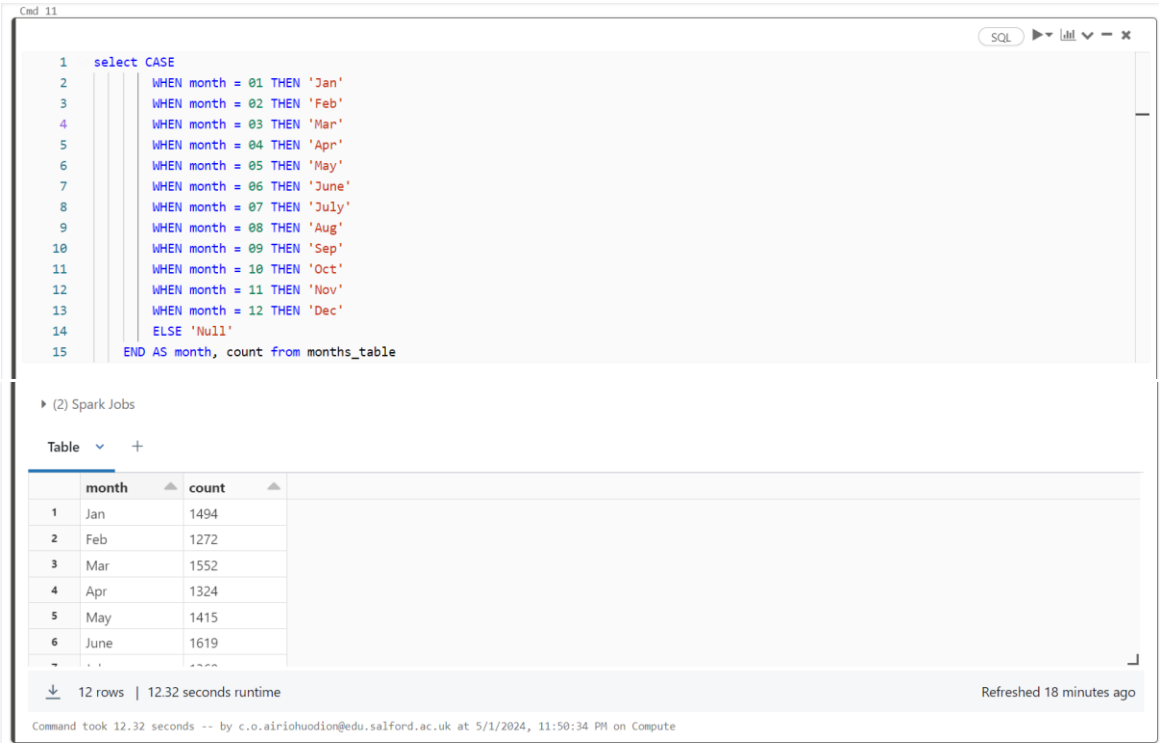


Figure 1.4.9 The data is then retrieved from the Months_Table view using the **SELECT** query to access the total counts of studies for each month in 2023 and then replace each month with its respective names.

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.

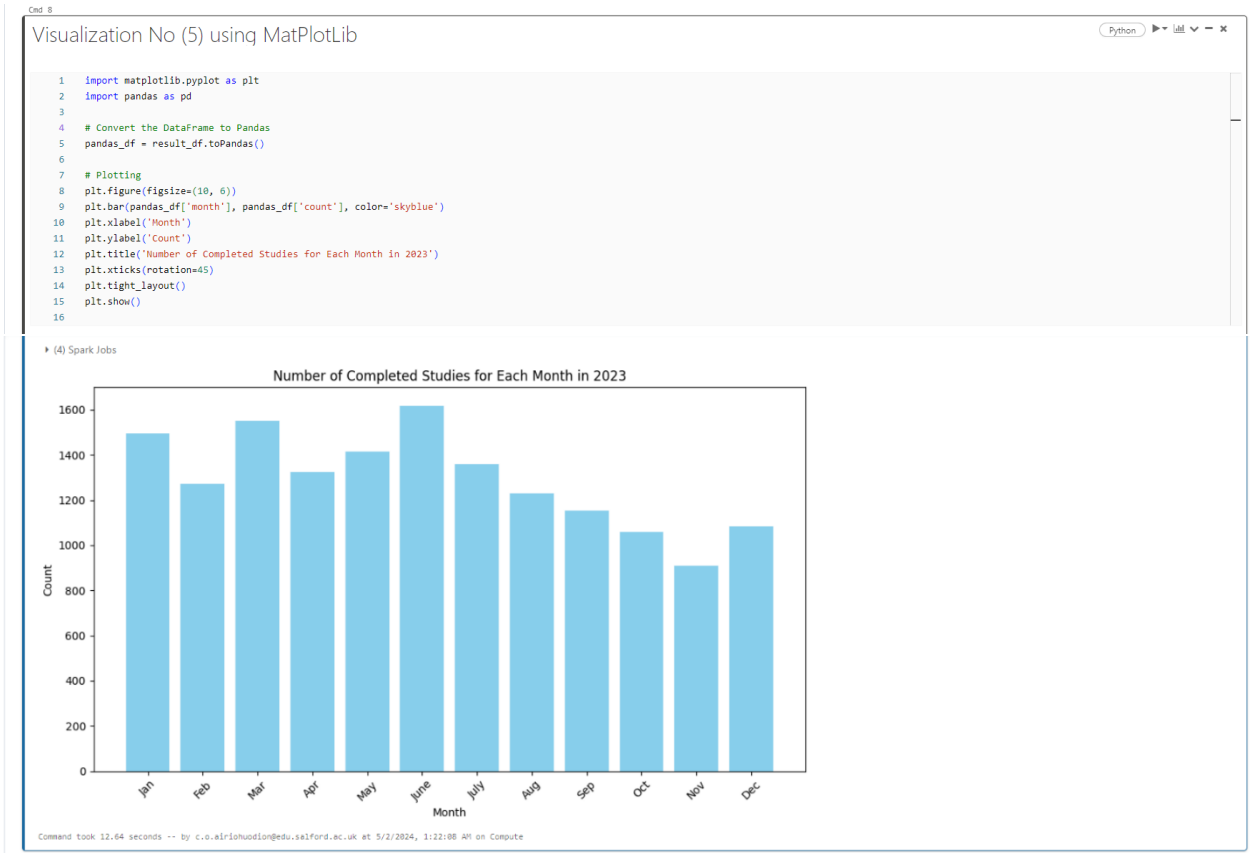


Figure 1.5.0 Visualization using **MatPlotLib** for question (5) in Dataframe NoteBook

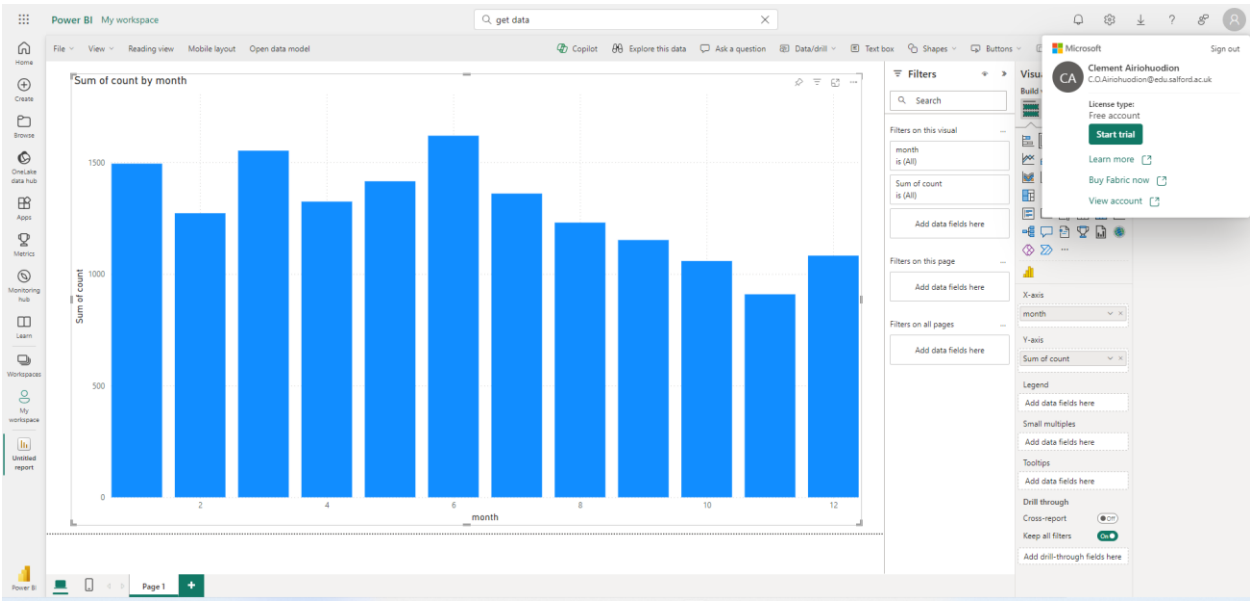


Figure 1.5.1 Visualization using **PowerBi** (Web) for question (5)

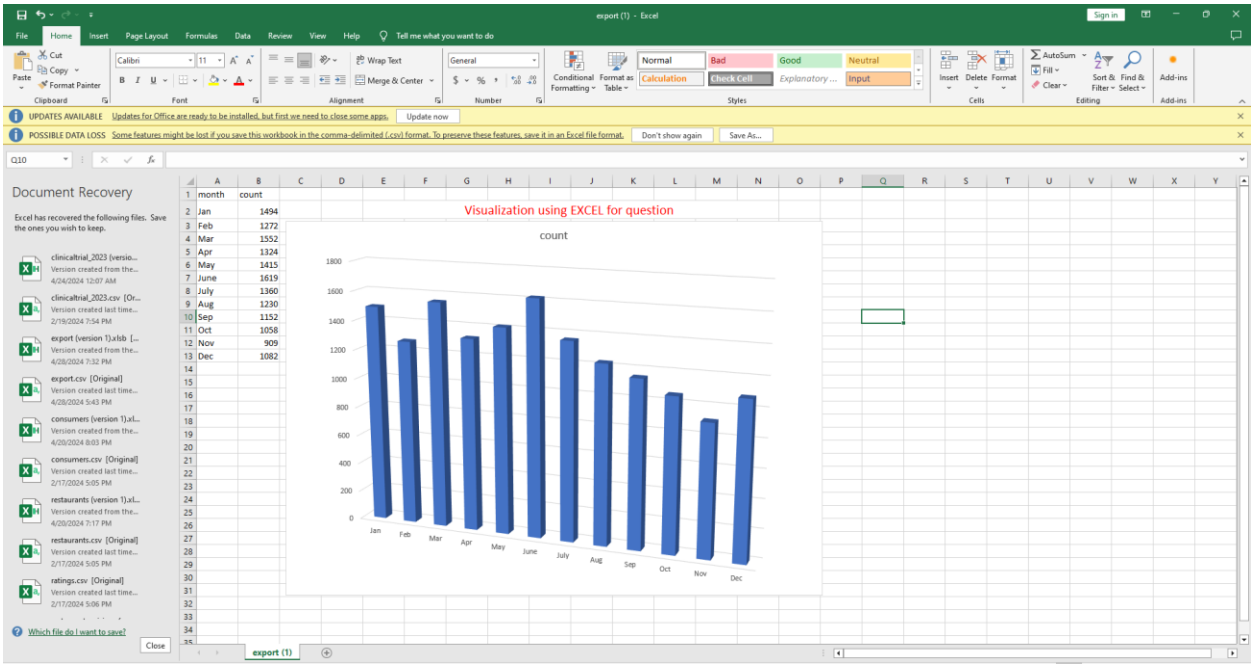


Figure 1.5.2 Visualization using **EXCEL Sheet** for question (5)

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

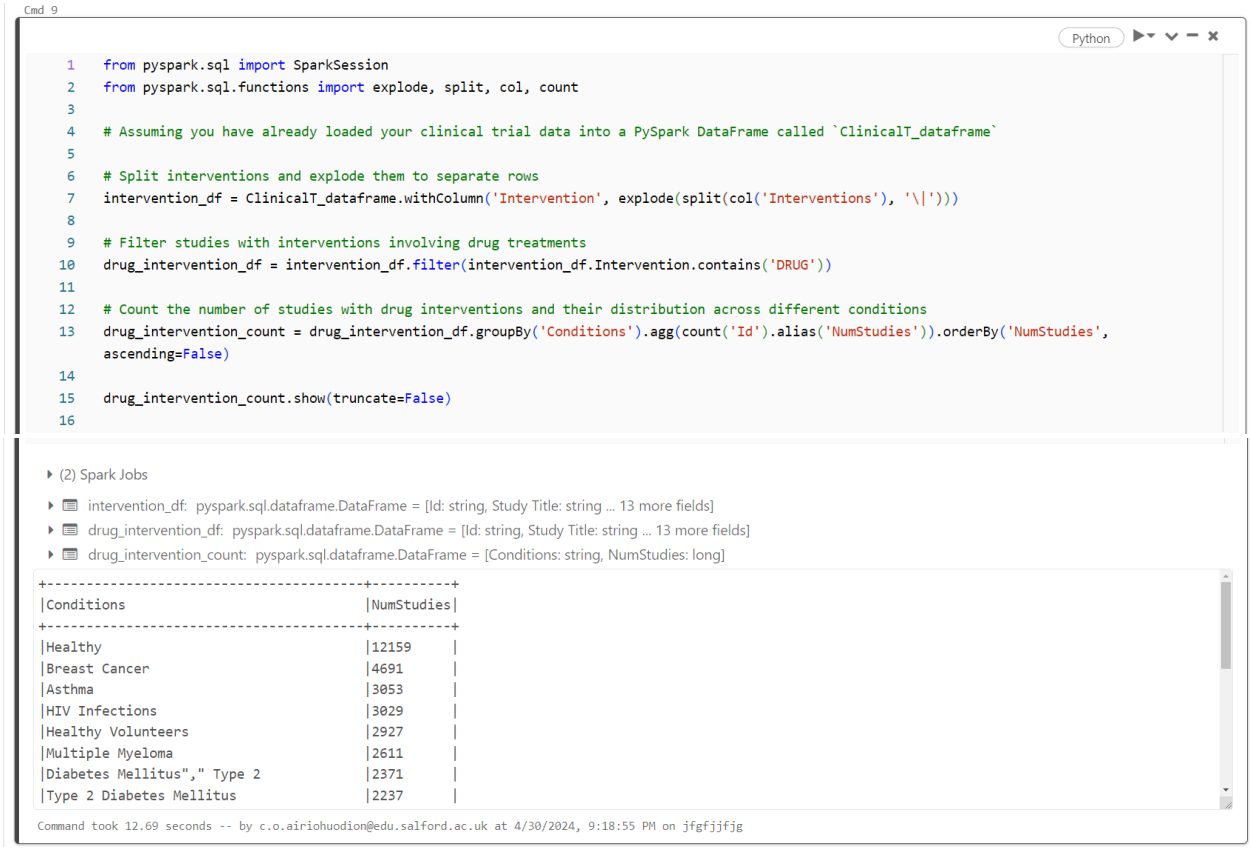


Figure 1.5.2 This code counts the number of studies and their distribution across various conditions after first splitting the interventions and then filtering for studies utilizing medication treatments.

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

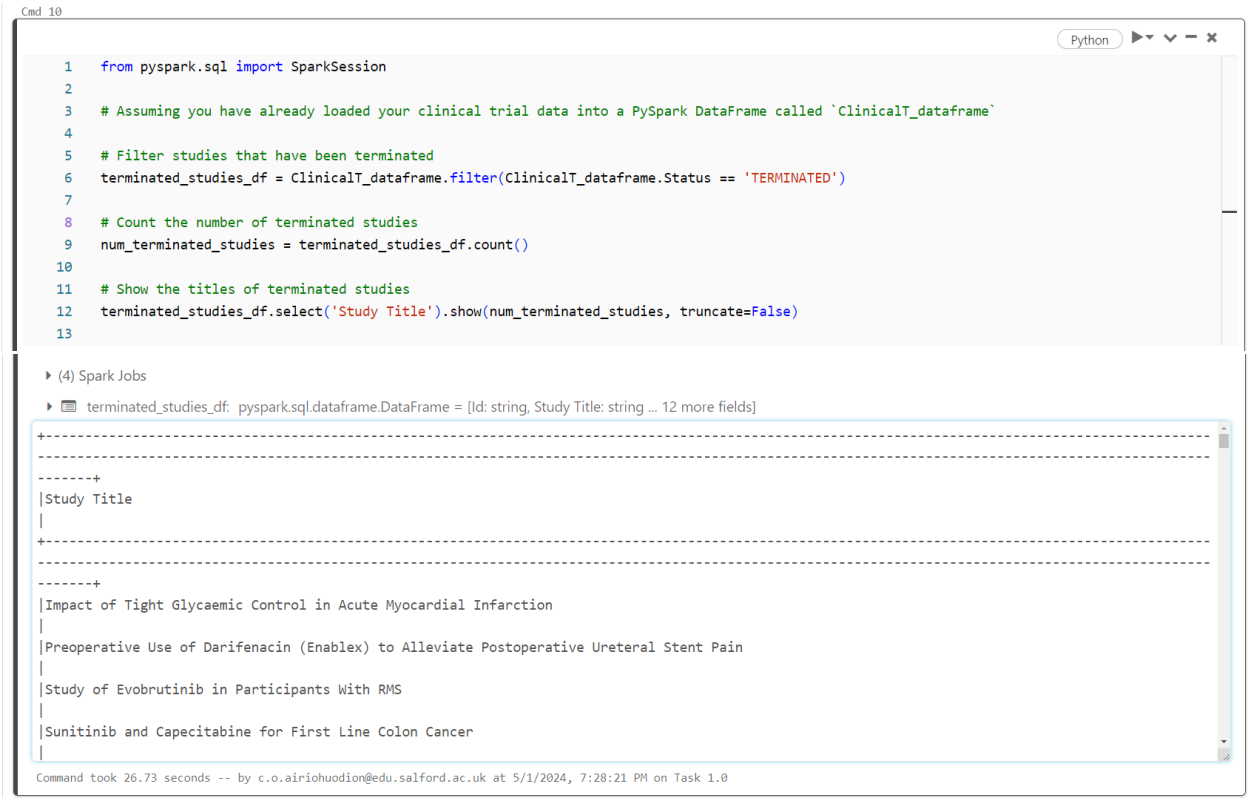
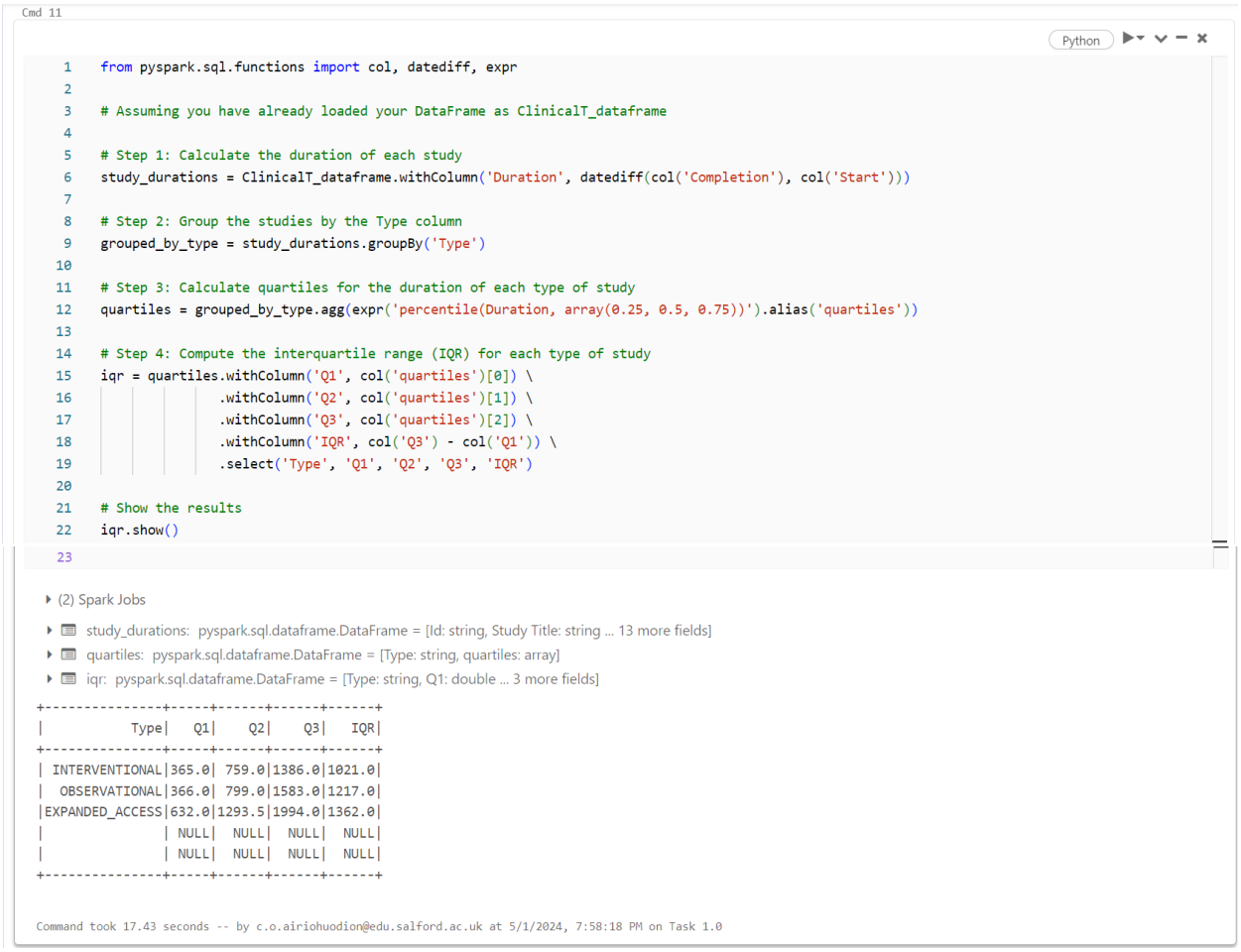


Figure 1.5.3 The code filters the DataFrame to include only the terminated studies, counts the number of terminated studies, and displays their respective titles.

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

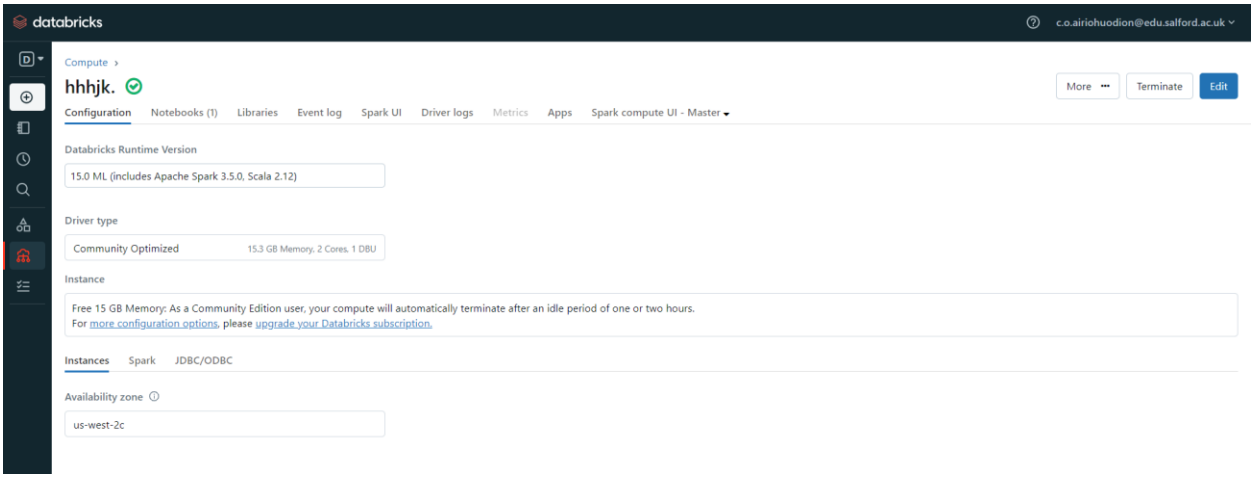


Figure 2.1.0 Compute Setup

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

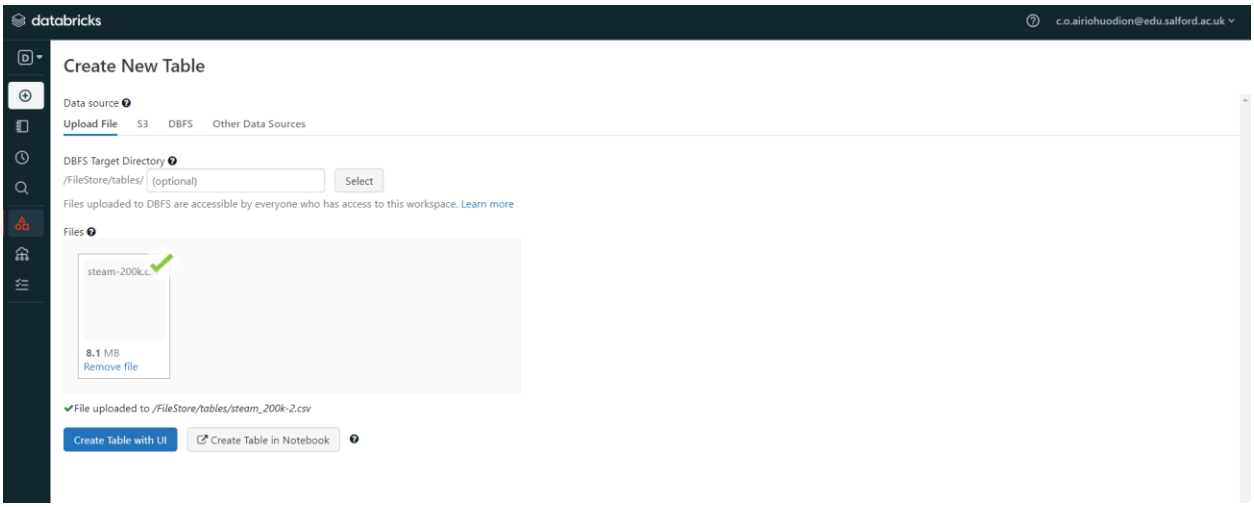


Figure 2.1.1 Uploaded steam-200k.csv dataset using the (GUI)

Loading The Data

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".

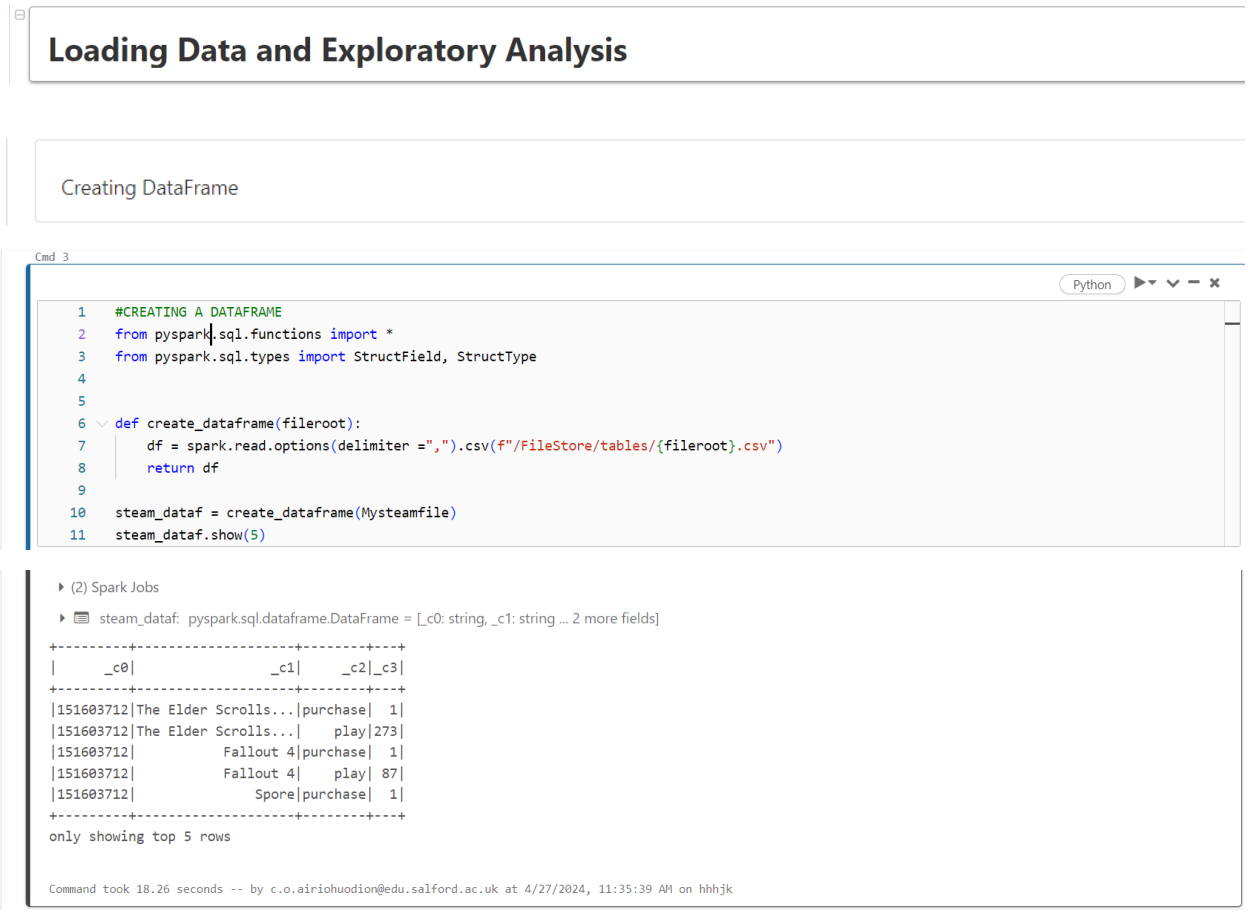


Figure 2.1.3 This section of code takes the CSV file containing Steam game data and uses it to generate a DataFrame called steam_dataf. It provides a method named create_dataframe, with a comma (,) as the delimeter, that uses Spark's DataFrame API to read the CSV file. The function accepts an argument 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
def alter_schema_name(df):
# Rename columns
    df = df.withColumnRenamed("_c0", "User_ID") \
           .withColumnRenamed("_c1", "Game") \
           .withColumnRenamed("_c2", "Member_Behaviour") \
           .withColumnRenamed("_c3", "PlayTime_Length")
    df = df.withColumn("User_ID", df["User_ID"].cast("int")).withColumn("PlayTime_Length", df["PlayTime_Length"].cast("float"))
    return df
# Show the updated DataFrame
steam_dataf = alter_schema_name(steam_dataf)
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 other to replace the generic column names from the last data frame, I've written this function to alter the previous scheme 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.

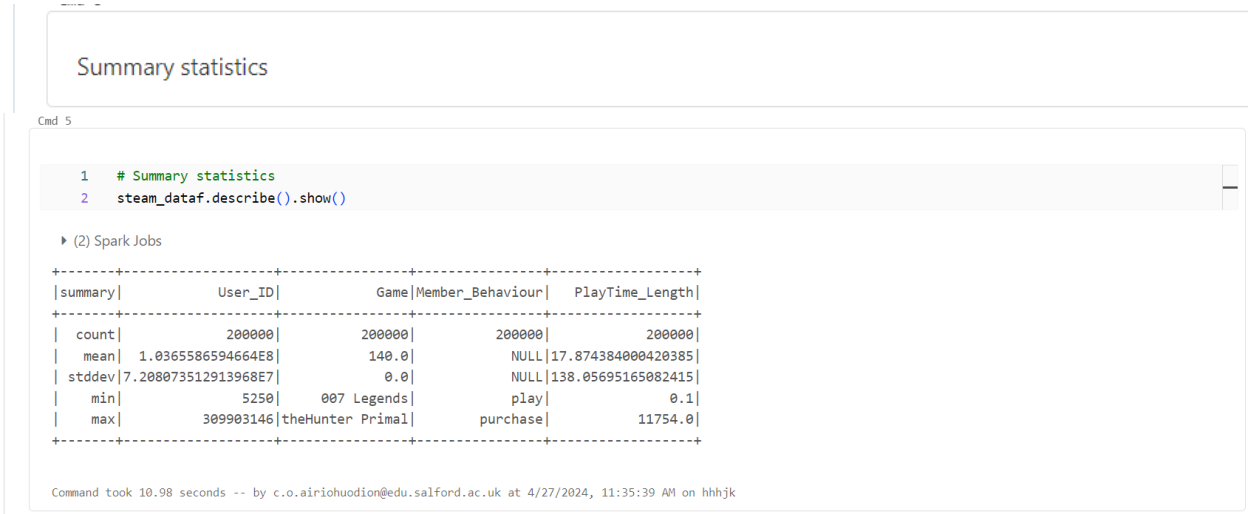


Figure 2.1.4 The summary statistics for the DataFrame counting Steam data include count, mean, standard deviation, minimum, and maximum values for numerical columns like User_ID and Playtime_Length.

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

```python
#Analysis;  Unique Games

number_games = steam_dataf.select("Game").distinct().count()
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

```python
# Analysis;  Unique users

number_users = steam_dataf.select("User_ID").distinct().count()
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

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

(2) Spark Jobs

Member_Behaviour: pyspark.sql.dataframe.DataFrame = [Member_Behaviour: string, count: long]

```
+----------------+------+
|Member_Behaviour| count|
+----------------+------+
|        purchase|129511|
|            play| 70489|
+----------------+------+
```

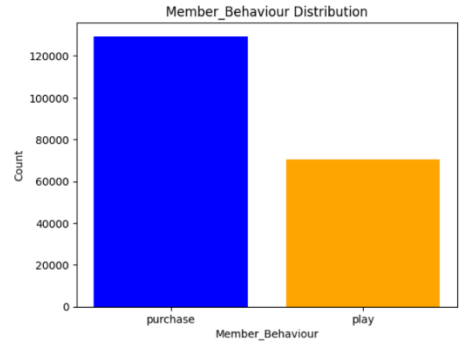
Command took 2.89 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

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

```
import matplotlib.pyplot as plt

# Get Member_Behaviour counts
steam_counts = steam_dataf.groupBy('Member_Behaviour').count().toPandas()

# Define colors for the bars
colors = ['Blue', 'Orange']

# Create a bar chart of the counts
plt.bar(steam_counts['Member_Behaviour'], steam_counts['count'], color=colors)

# Set the chart title and axis labels
plt.title('Member_Behaviour Distribution')
plt.xlabel('Member_Behaviour')
plt.ylabel('Count')

# Show the chart
plt.show()
```



Command took 4.01 seconds -- by c.o.airiohuodion@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

Python

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

```
#Analysis; Most Often played games (Highest played game)

# Filter the dataframe for "play" behavior, group by "Game", count occurrences, and order by count
most_played_games = steam_dataf.filter(steam_dataf["Member_Behaviour"] == "play") \
                               .groupBy("Game") \
                               .count() \
                               .orderBy('count', ascending=False) \
                               .limit(10)  # Limit the result to top 10 games
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

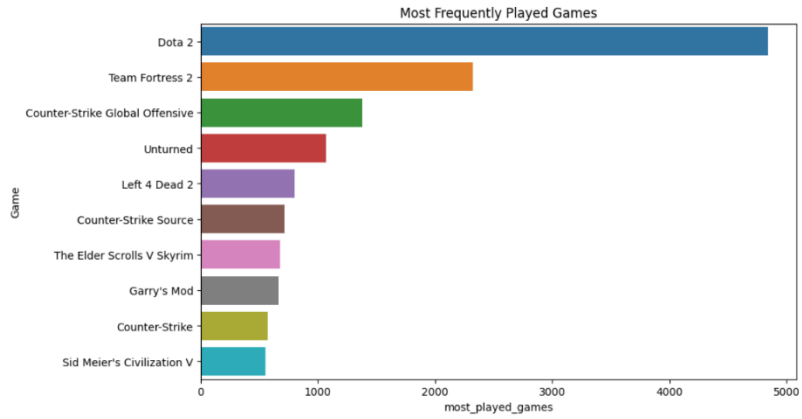
Figure 2.2.1 This code analyzes the top played games using the DataFrame **steam_dataframe**. It filters rows with play behavior, aggregates information, **counts** instances, **sorts** them in **descending order**, and displays the results. The **limit()** function restricts results to the top 10 most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

Most Frequently Played Games

Cmd 13

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Filter the dataframe for "play" behavior, group by "game", count occurrences, and order by count
most_played_games = steam_dataf.filter(steam_dataf["Member_Behaviour"] == "play") \
                               .groupBy("Game") \
                               .count() \
                               .orderBy('count', ascending=False) \
                               .limit(10) \
                               .toPandas()

# Plotting the top 10 most frequently played games
plt.figure(figsize=(10, 6))
sns.barplot(x='count', y='Game', data=most_played_games)
plt.title('Most Frequently Played Games')
plt.xlabel('most_played_games')
plt.ylabel('Game')
plt.show()
```

(2) Spark Jobs



Command took 7.20 seconds -- by c.o.airiohuodion@edu.salford.ac.uk at 4/27/2024, 11:35:39 AM on hhhjk

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

```
# Calculate the count of purchases per game
most_purchased_games = steam_dataf.filter(steam_dataf["Member_Behaviour"] == "purchase") \
                                  .groupBy("Game") \
                                  .count() \
                                  .orderBy('count', ascending=False) \
                                  .limit(10)  # Limit the result to top 10 games

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

Cmd 16

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Filter the dataframe for "purchase" behavior, group by "Game", count occurrences, and order by count
most_purchased_games = steam_dataf.filter(steam_dataf["Member_Behaviour"] == "purchase") \
                                  .groupBy("Game") \
                                  .count() \
                                  .orderBy('count', ascending=False) \
                                  .limit(10) \
                                  .toPandas()

# Plotting
plt.figure(figsize=(10, 6))
sns.barplot(x='count', y='Game', data=most_purchased_games)
plt.title('Most Frequently Purchased Games')
plt.xlabel('most_purchased_games')
plt.ylabel('Game')
plt.show()
```

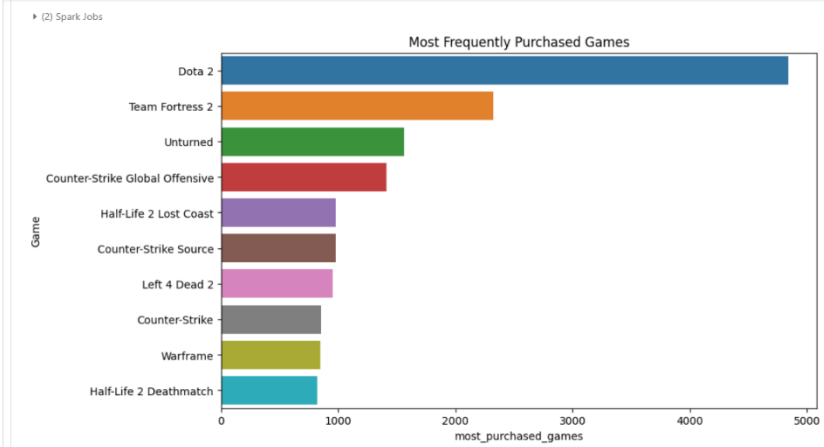


Figure 2.2.4 It employs Seaborn to generate a bar plot with x-axis and y-axis, providing a visual representation of the most purchased games.

Games with the highest number of PlayTime_Length in hours

Cmd 17

Python

```
#Analysis; Games with the highest number of PlayTime_Length in hours

# Calculate the total play time length per game
total_playtime_per_game = steam_dataf.filter(steam_dataf["Member_Behaviour"] == "play") \
                                     .groupBy("Game") \
                                     .agg(round(sum(steam_dataf["PlayTime_Length"]), 2).alias("Total_PlayTime_IN_HOUR")) \
                                     .orderBy('Total_PlayTime_IN_HOUR', ascending=False) \
                                     .limit(10)
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 hhhjk

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.

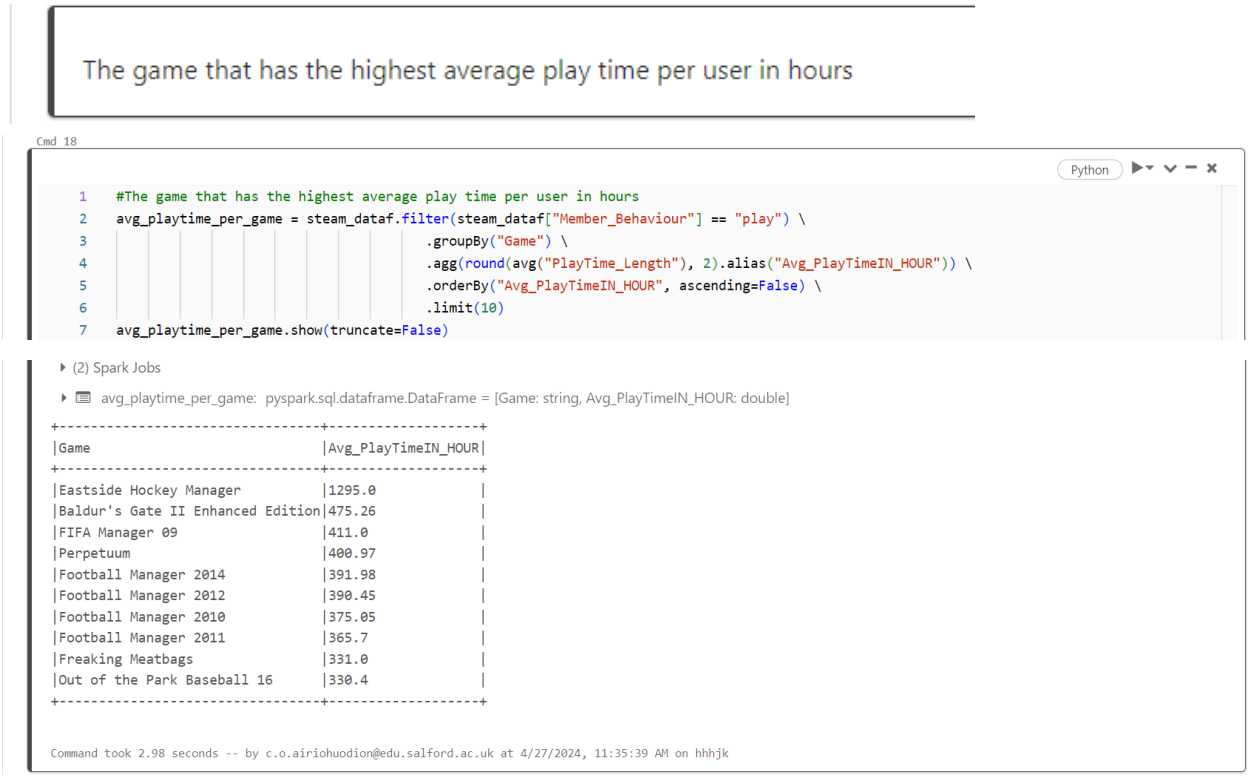


Figure 2.2.6 The code filters the DataFrame **steam_dataframe** to show **average** user game play behavior. It then **groups** game records and uses the '**avg**' aggregation function to calculate average playtime length for each game, aliased as '**Avg PlayTimeIN_HOUR'**. The data is **sorted** by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

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.

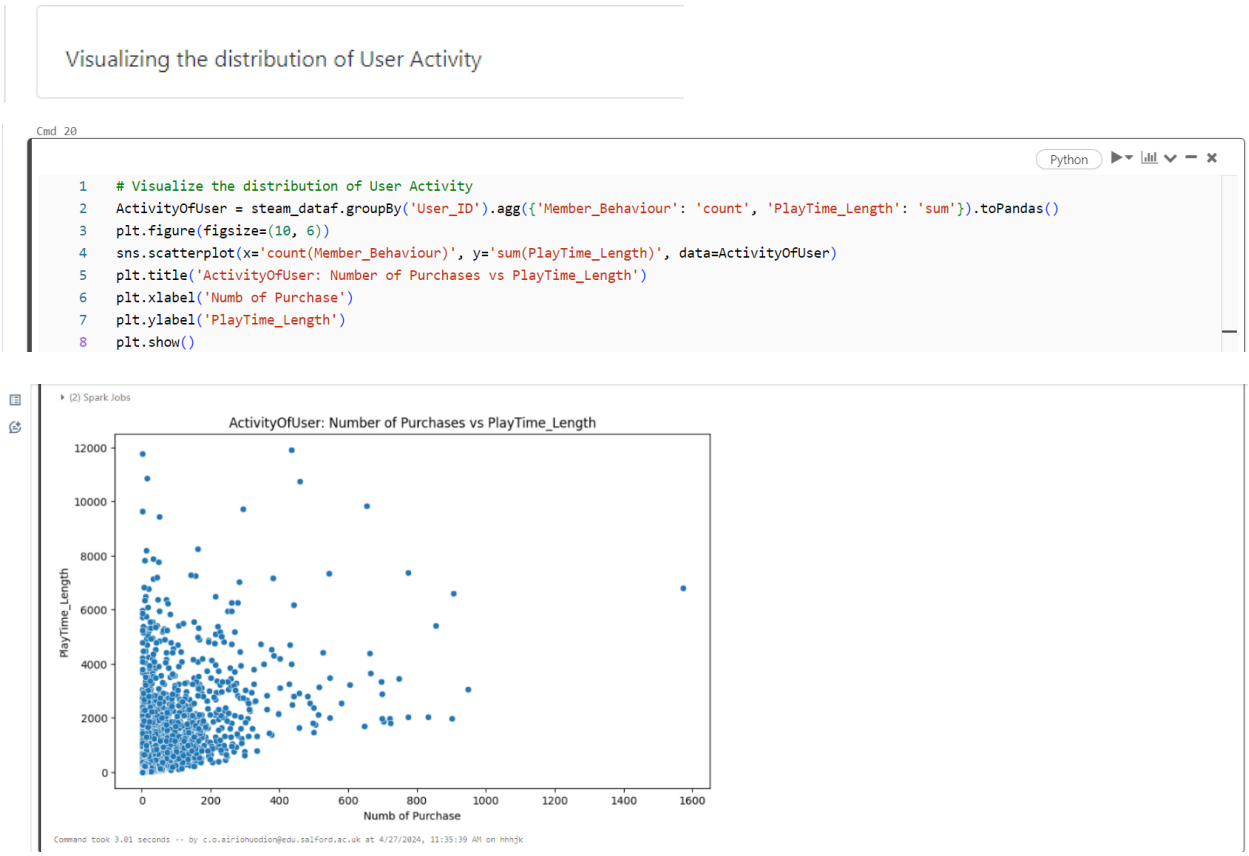


Figure 2.2.8 This code visualizes user activity in the Steam dataset by grouping data by "User_ID", adding total "PlayTime_Behaviour" for each user, and generating a scatter plot with Seaborn, revealing gameplay duration and purchase quantity.

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.

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.

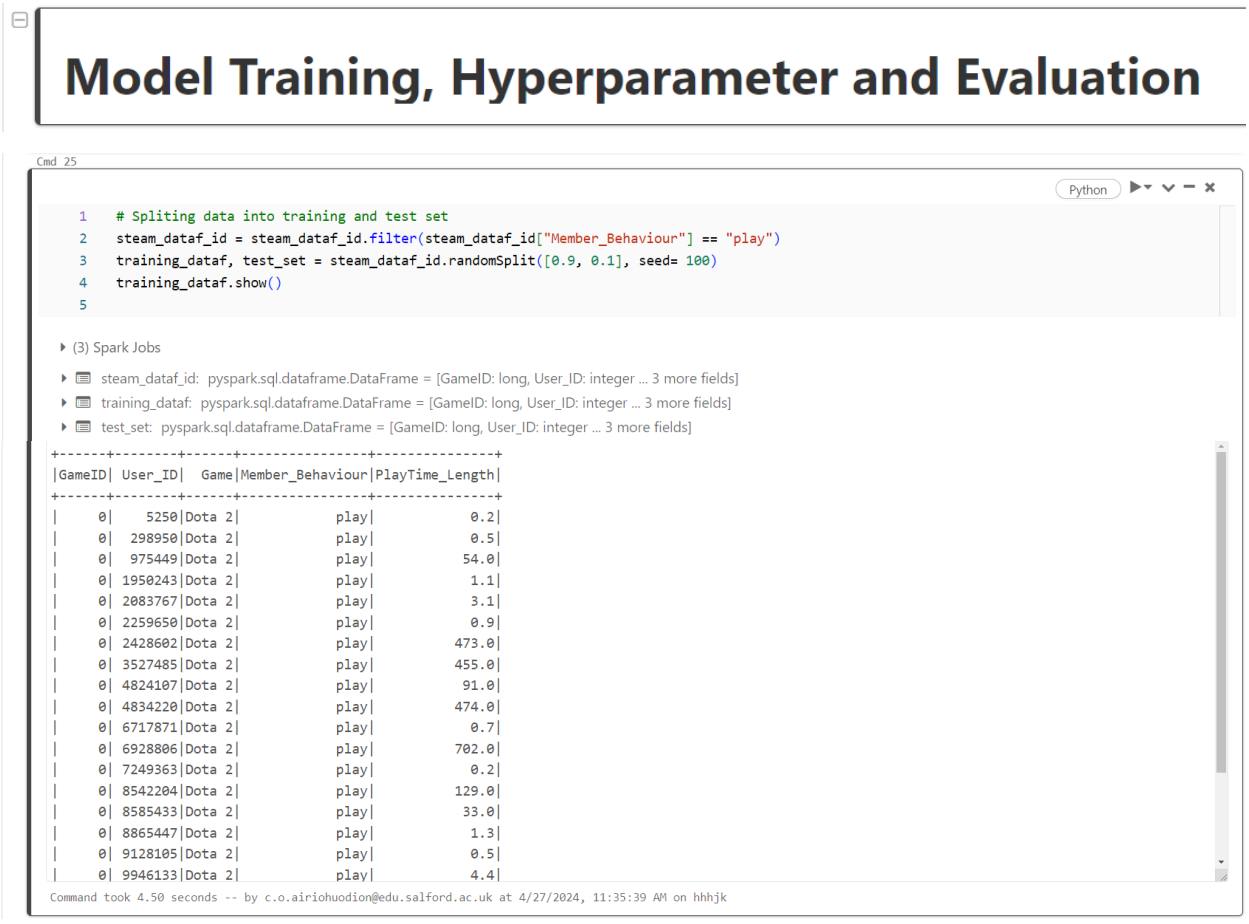


Figure 2.3.1 This section of code divides the data into test and training sets for the purpose of machine learning. First, only rows with 'play' selected in the '**Member_Behaviour**' column of the **steam_dataframe_id** DataFrame are included. Next, it splits the filtered DataFrame into the **trainning_set** (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

Python

```
# Training a model using Alternating least squares(ALS) with the training set
from pyspark.ml.recommendation import ALS

recommender = ALS(userCol="User_ID", ratingCol="PlayTime_Length", itemCol="GameID", coldStartStrategy="drop", nonnegative=True)
recommender = recommender.fit(training_dataf)
```

(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

```
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

predicts: pyspark.sql.dataframe.DataFrame = [GameID: integer, User_ID: integer ... 4 more fields]

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

Command took 10.80 seconds -- by c.o.airiohuodion@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

Python

```
# Evaluate the model by computing the RMSE
from pyspark.ml.evaluation import RegressionEvaluator

evaluator = RegressionEvaluator(metricName="rmse", labelCol="PlayTime_Length", predictionCol="prediction")
rmse = evaluator.evaluate(predicts)
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

Python

```
# Testing the recommendation system on a single user

single_user = steam_dataf_id.filter(steam_dataf_id["User_ID"] == 32125590)
single_recommendation = recommender.transform(single_user)
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.

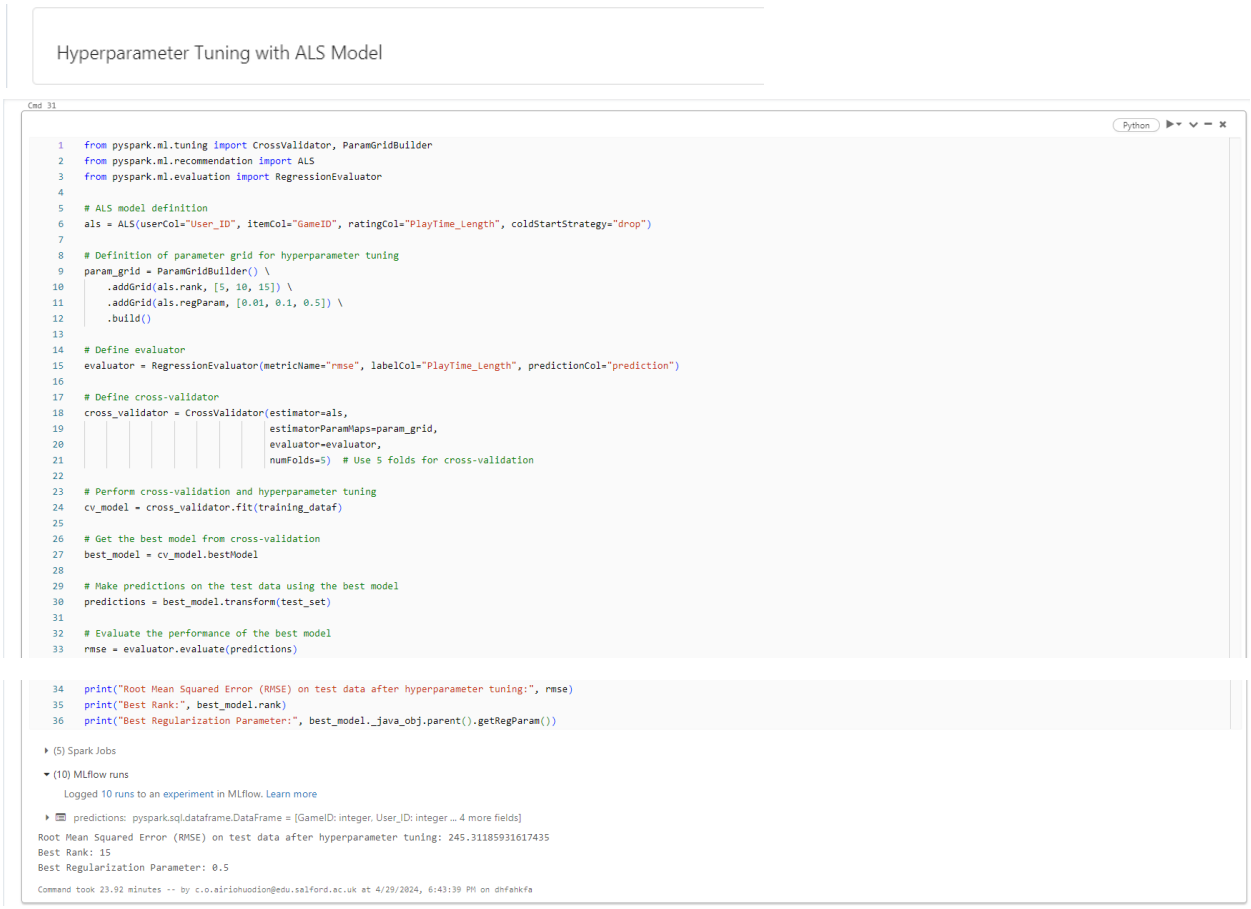


Figure 2.3.6 Cross-validation is used to adjust the ALS model by experimenting with various hyperparameter combinations. Based on how well the model performs as determined by the test data's root mean squared error (RMSE) measure, the optimal model is chosen. The RMSE score is provided together with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

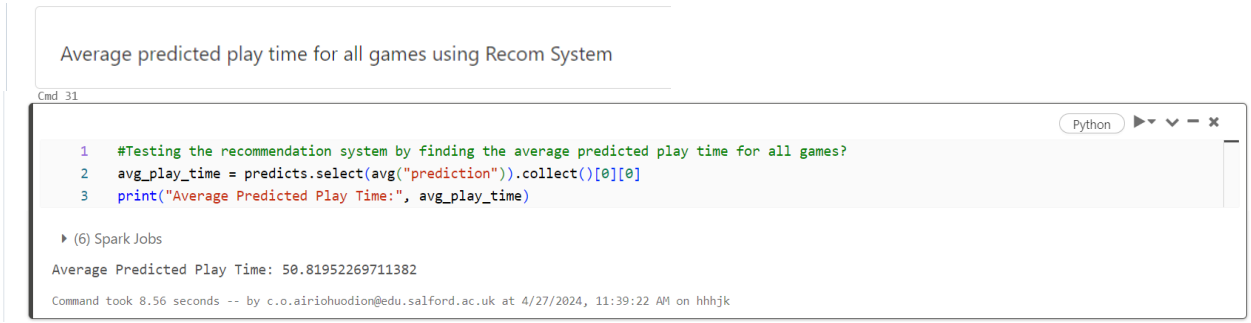


Figure 2.3.7 This average projected play time for each game suggested in the '**preds**' DataFrame is computed by this code to test the recommendation system. The '**avg()**'

function is used to 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.