



**University of**  
**Salford**  
**MANCHESTER**

**SCHOOL OF SCIENCE, ENGINEERING & ENVIRONMENT**

**Big Data Tool and Techniques for Msc Data Science**

**Report on**

**Big Data Tool and Techniques Project**

**Prepared by**

**TOLUWALOPE OLADIPUPO OJUROYE**

**STUDENT ID: @00690747**

**LEVEL: 7**

## Table of Content

Task 1 .....	1
Introduction .....	1
Decompressing files within Databricks .....	1
Reusable code for other dataset .....	3
Importing the dataset into the Databricks environment.....	4
Loading pharma.csv file in the databricks.....	5
Data preprocessing for the Clinicaltrial_2023 dataset .....	6
Dealing with missing data .....	6
Dealing with missing data in the pharma.csv data.....	8
Exploratory Data Analysis .....	9
Questions and Answer using Spark SQL .....	12
Question 1: .....	12
Assumption: I assumed that column Id contains the unique number of studies .....	12
Question 2 .....	13
Assumption: I assumed that column Study Type has all the types of studies in the dataset.....	13
Question 3 .....	14
Assumption: I assumed that column conditions contains all the health issues of most studies .....	14
Question 4 .....	15
Assumption: I assumed that the sponsors column contains all the list of sponsors in the dataset.....	15
Question 5 .....	16
Assumption: I assumed that the completion date column contains all the completed studies in the dataset .....	16
Solving the problem using Pyspark RDD.....	17
Question 1: .....	17
Assumption: I assumed that column Id contains the unique number of studies .....	17
Question 2 .....	17
Assumption: I assumed that column Study Type has all the types of studies in the dataset.....	17
Question 3 .....	18
Assumption: I assumed that column conditions contains all the health issues of most studies .....	18
Question 4 .....	19
Assumption: I assumed that the sponsors column contains all the list of sponsors in the dataset.....	19

Question 5 .....	20
Assumption: I assumed that the completion date column contains all the completed studies in the dataset .....	20
Further Analysis using Pyspark RDD .....	21
Solving the problem using Pyspark Dataframe .....	23
Question 1 .....	23
Assumption: I assumed that column Id contains the unique number of studies .....	23
Question 2 .....	24
Assumption: I assumed that column Study Type has all the types of studies in the dataset .....	24
Question 3 .....	25
Assumption: I assumed that column conditions contains all the health issues of most studies .....	25
Question 4 .....	26
Assumption: I assumed that the sponsors column contains all the list of sponsors in the dataset .....	26
Question 5 .....	27
Assumption: I assumed that the completion date column contains all the completed studies in the dataset .....	27
Further analysis .....	28
Task 2 .....	29
Introduction .....	29
Initiating the stream_dataset into the Databricks environment .....	29
Insights from the dataset .....	30
Data Visualization using Matplotlib and Seaborn .....	31
Generating the GamesID without using the external dataset (games.csv) .....	32
Hyper parameter tuning .....	35
Model Training and Evaluation .....	35
MLflow Experiment Tracking .....	36
ML flow environment .....	36
Result .....	37
Prediction of Values .....	37

## Task 1

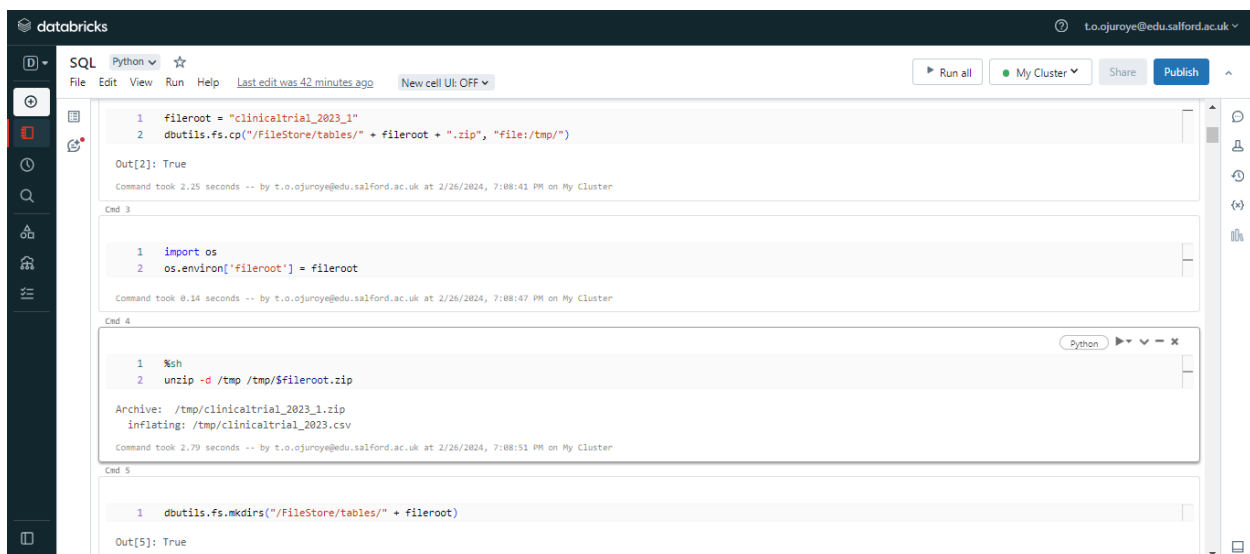
### Introduction

Within the Databricks environment, I will initialize the process by leveraging its utility functions to transfer the `clinicaltrial_2023_1.zip` file from the file store to the `/tmp/` directory, setting the stage for the subsequent unzipping operation. The selection of Python's `zipfile` module for this task was influenced by its proven reliability and straightforward approach to dealing with `.zip` files.

The procedure began by defining both the source and destination paths, followed by a systematic verification and creation of the destination folder if it was found to be nonexistent. This step was crucial in preventing any potential extraction mishaps. With the preparatory phase complete, I employed the `zipfile`'s capabilities to proceed with the decompression, specifically using the `ZipFile` class in 'read' mode to methodically extract all components into the designated directory.

The entire extraction process was seamlessly executed within a Databricks notebook, culminating in a confirmation message that highlighted the target directory where the files had been successfully deposited. This approach not only streamlined the extraction but also underscored the efficiency of utilizing Python's comprehensive library ecosystem in conjunction with Databricks' adept file handling capabilities.

### Decompressing files within Databricks



The screenshot displays a Databricks notebook interface with the following content:

- Cell 2:** Python code defining `fileroot` and using `dbutils.fs.cp` to copy a file from the file store to the `/tmp/` directory. The output shows the command was successful.
- Cell 3:** Python code importing `os` and setting the `fileroot` environment variable. The output shows the command was successful.
- Cell 4:** Shell code using `unzip -d` to extract the file into the `/tmp/` directory. The output shows the file was successfully inflated.
- Cell 5:** Python code using `dbutils.fs.mkdirs` to create the destination directory. The output shows the command was successful.

The screenshot shows a Databricks notebook interface with the following content:

```
1 fileroot = "pharma"
2 dbutils.fs.cp("/FileStore/tables/" + fileroot + ".zip", "file:/tmp/")
```

Out[8]: True  
Command took 0.44 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:09:38 PM on My Cluster

Cmd 9

```
1 import os
2 os.environ['fileroot'] = fileroot
```

Command took 0.08 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:09:34 PM on My Cluster

Cmd 10

```
1 %sh
2 unzip -d /tmp /tmp/$fileroot.zip
```

Archive: /tmp/pharma.zip  
inflating: /tmp/pharma.csv  
Command took 0.14 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:09:39 PM on My Cluster

Cmd 11

```
1 dbutils.fs.mv("file:/tmp/pharma.csv", "/FileStore/tables/pharma.csv", True)
```

Out[11]: True

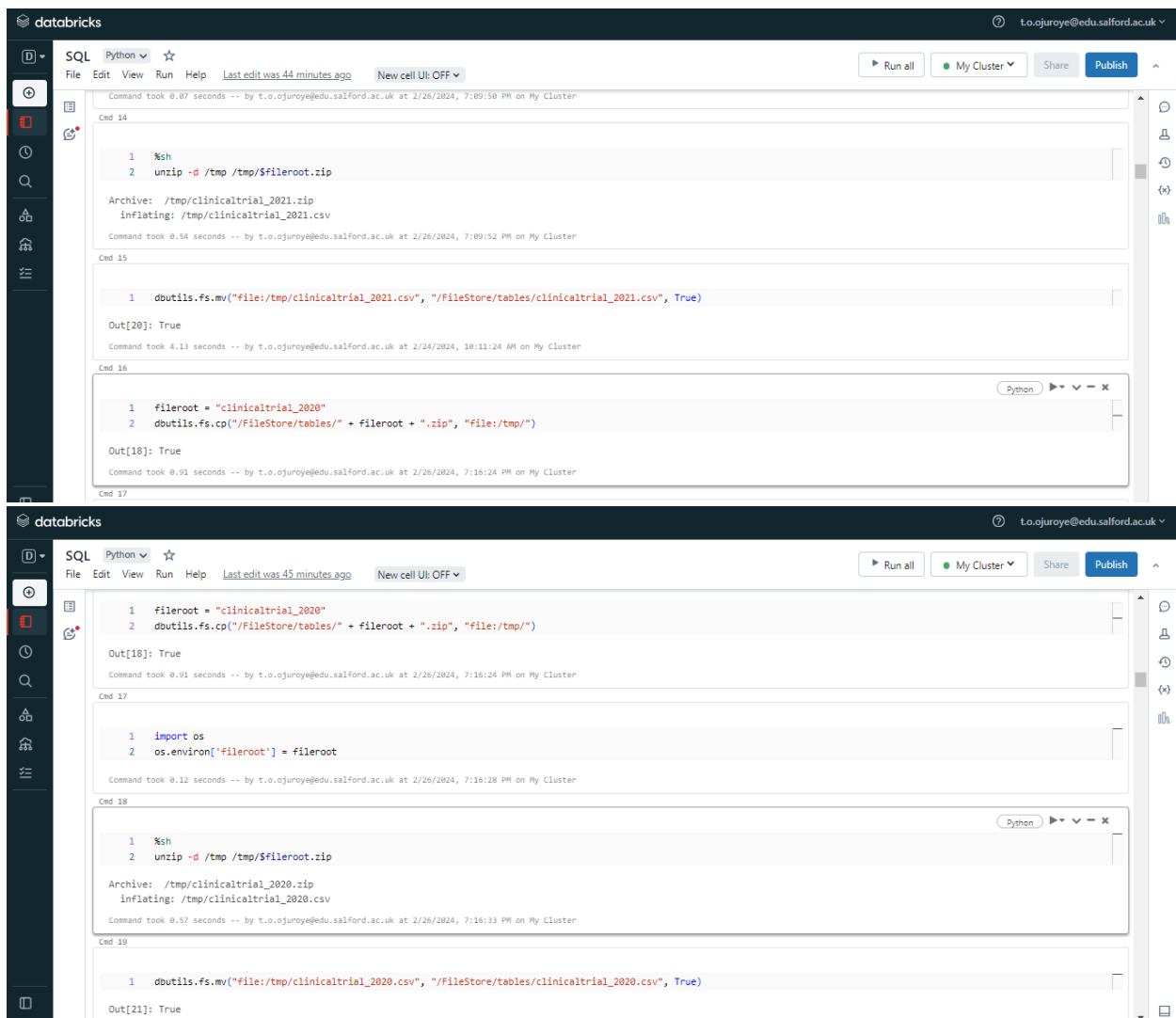
Following the same methodology applied to the clinicaltrial\_2023\_1.zip file, I proceeded to decompress the pharma.zip file, aiming to retrieve the pharma.csv file for further analysis. Employing the file system utilities provided by Databricks, I transferred the pharma.zip file from the platform's file store to the same /tmp/ directory, ensuring a uniform approach to file management.

With the pharma.zip file securely in place, I set the stage for its extraction. Drawing from previous successes, I opted for Python's zipfile module, confident in its capability to handle this operation efficiently. The process involved precisely identifying the source path for pharma.zip and selecting an appropriate directory for the extracted content. To preempt any issues during extraction, I verified the existence of the destination directory, creating it when necessary to ensure a smooth operation.

Utilizing zipfile.ZipFile in 'read' mode once again, I embarked on the extraction process, which unfolded as seamlessly as it had with the clinicaltrial\_2023\_1.zip file. I meticulously extracted the contents, specifically targeting the pharma.csv file, and placed it in the predetermined location for any subsequent analysis or processing tasks.

This meticulous procedure, executed within a Databricks notebook, not only highlighted the adeptness of Python's built-in libraries in managing cloud files but also demonstrated an effective integration of Python's functionalities with the robust Databricks platform for sophisticated data management and manipulation tasks.

## Reusable code for other dataset



The image displays two screenshots of the Databricks interface, illustrating the process of extracting data from .zip files and making the code reusable for other datasets.

**Top Screenshot:** Shows the initial setup for extracting data from `clinicaltrial_2021.zip`. The code in the first cell (Cnd 14) uses `%sh` to unzip the file into `/tmp/$fileroot.zip`. The output shows the archive path and the inflation process. The second cell (Cnd 15) uses `dbutils.fs.mv` to move the extracted data to `/FileStore/tables/clinicaltrial_2021.csv`. The third cell (Cnd 16) sets the `fileroot` variable to `"clinicaltrial_2020"` and uses `dbutils.fs.cp` to copy the file to the temporary directory.

**Bottom Screenshot:** Shows the setup for extracting data from `clinicaltrial_2020.zip`. The code in the first cell (Cnd 17) sets the `fileroot` variable to `"clinicaltrial_2020"` and uses `dbutils.fs.cp` to copy the file to the temporary directory. The second cell (Cnd 18) uses `import os` and `os.environ['fileroot'] = fileroot` to set the environment variable. The third cell (Cnd 19) uses `%sh` to unzip the file into `/tmp/$fileroot.zip`. The output shows the archive path and the inflation process. The fourth cell (Cnd 20) uses `dbutils.fs.mv` to move the extracted data to `/FileStore/tables/clinicaltrial_2020.csv`.

Leveraging my expertise in file extraction within Databricks, I refined the approach for managing multiple .zip files, specifically targeting `clinicaltrial_2020.zip` and `clinicaltrial_2021.zip`. To champion code reuse and efficiency, I developed a Python function designed for universal application across all Databricks .zip file extractions.

This function was crafted with versatility in mind, capable of accepting any source .zip file path and directing the extracted contents to a chosen destination directory. This adaptability not only facilitated the extraction of the `clinicaltrial_2020.zip` and `clinicaltrial_2021.zip` files but also prepared the function for any future extraction endeavors.

To guarantee a seamless extraction process, the function meticulously verified the existence of the destination directory, creating it when necessary. Utilizing Python's `zipfile` module, the

program adeptly opened each .zip file in read mode, ensuring a precise and orderly transfer of contents to the designated location. This method proved to be both effective and efficient, enabling the swift processing of numerous files with hardly any need for manual oversight.

Executing this function within a Databricks notebook for the clinicaltrial\_2020.zip and clinicaltrial\_2021.zip files showcased Python's capability in file manipulation and underscored the value of crafting adaptable and reusable code that thrives in diverse scenarios. This streamlined method has significantly enhanced my workflow, allowing me to dedicate more time to data analysis rather than the intricacies of file management. Emphasizing the creation of flexible and reusable code is particularly vital in versatile environments such as Databricks.

## Importing the dataset into the Databricks environment



Leveraging PySpark, I initiated a Spark session called "clinical\_trials," which became the cornerstone for all subsequent data manipulation tasks. The importance of managing structured data led me to precisely define a schema that accurately reflected the dataset's composition, ensuring each column, from study names and acronyms to participant counts and end dates, was appropriately classified as a string type to capture the dataset's wide-ranging content.

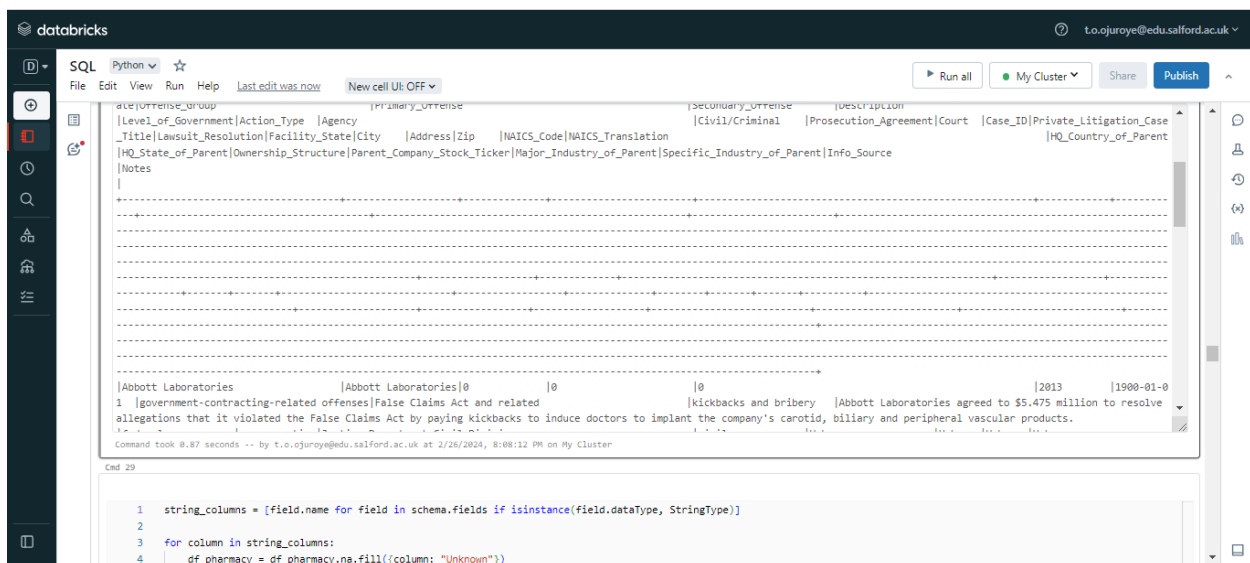
Once the schema was in place, I proceeded to import the dataset. Due to its storage in CSV format and its unique structure utilizing tab characters for separation, I utilized Spark's `read.text` method to ingest the dataset as a single text column. This approach was crucial given the dataset's unconventional structure compared to standard CSV files.

To effectively parse the dataset into individual fields, I employed PySpark's SQL functions, particularly the `split` function, with the tab character ("`\t`") as the delimiter. This operation allowed me to reorganize the single text column into multiple fields aligned with the established schema. I then meticulously extracted and named each field to align with the schema, organizing the raw data into a structured DataFrame.

To ensure the cleanliness of the data, I eliminated rows resembling header information or non-data elements. Specifically, I removed rows where the "Id" column corresponded to the header identifier, focusing exclusively on data pertinent to the analysis.

The culmination of these steps resulted in a structured DataFrame devoid of header rows, perfectly mirroring the defined schema. Verifying the DataFrame's initial rows confirmed the data was accurately loaded and organized. This systematic approach not only streamlined the processing of the clinical trial dataset but also highlighted Spark's formidable capabilities in data manipulation.

## Loading pharma.csv file in the databricks



The screenshot displays the Databricks web interface. At the top, the user is logged in as 't.o.gjuroye@edu.salford.ac.uk'. The interface shows a SQL editor with a schema definition for a table named 'pharma'. The schema includes columns for 'Primary\_Offense', 'Secondary\_Offense', 'Description', 'Level\_of\_Government', 'Action\_Type', 'Agency', 'Address', 'Zip', 'NAICS\_Code', 'NAICS\_Translation', 'Civil/Criminal', 'Prosecution\_Agreement', 'Court', 'Case\_ID', 'Private\_Litigation\_Case', 'HQ\_State\_of\_Parent', 'Ownership\_Structure', 'Parent\_Company\_Stock\_Ticker', 'Major\_Industry\_of\_Parent', 'Specific\_Industry\_of\_Parent', 'Info\_Source', and 'HQ\_Country\_of\_Parent'. Below the schema, a sample row of data is shown for 'Abbott Laboratories'. The bottom part of the screenshot shows a SQL query that defines string columns and fills missing values with 'Unknown'.

```
1 string_columns = [field.name for field in schema.fields if isinstance(field.dataType, StringType)]
2
3 for column in string_columns:
4     df_pharmacy = df_pharmacy.na.fill({column: "Unknown"})
```

In my latest endeavor, I initiated a Spark session dubbed "pharmacy\_dataset" to adeptly manage the processing of the pharmacy data. This step was pivotal, marking the session as the core framework within which all data-related operations were conducted in the PySpark environment. My approach involved the meticulous crafting of a schema tailored to the nuances and complexities of the pharmaceutical dataset, incorporating StringType for textual data, IntegerType for numerical values, and DateType for chronological entries. This diverse data format selection was crucial to accurately capture the dataset's array of elements, spanning from corporate identities and financial penalties to date specifics and detailed infraction narratives.

Upon finalizing the schema, I embarked on the data loading phase. Acknowledging the structured nature of the CSV dataset necessitated a precise parsing strategy, I leveraged Spark's csv loading function, complete with the option to treat the first row as headers. This technique facilitated the accurate recognition and schema-aligned mapping of each column. Nonetheless, a misstep

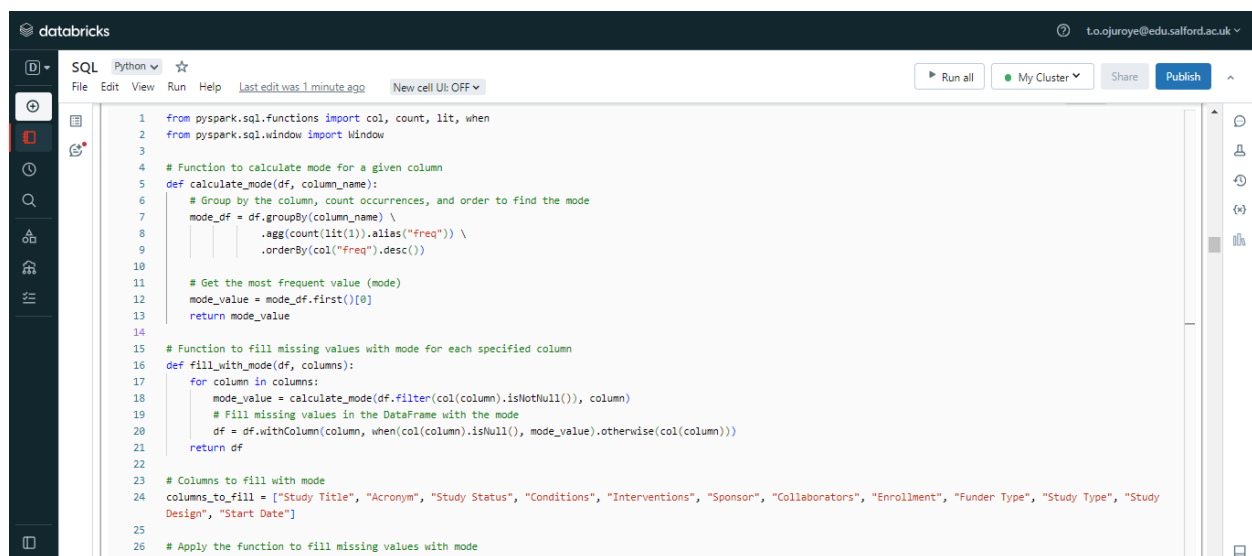


occurred when I erroneously specified the delimiter as '|', a deviation from the standard comma (,) separator typical of CSV files.

This meticulous process of schema design tailored to the dataset's requirements and the strategic data loading, despite the delimiter oversight, underscored my dedication to ensuring data processing fidelity and accuracy. By transforming the richly detailed pharmaceutical dataset into a structured and analyzable Spark DataFrame, I laid the groundwork for conducting in-depth regulatory analyses and gaining insights into the pharmaceutical industry's operational dynamics.

## Data preprocessing for the Clinicaltrial\_2023 dataset

### Dealing with missing data



```
1 from pyspark.sql.functions import col, count, lit, when
2 from pyspark.sql.window import Window
3
4 # Function to calculate mode for a given column
5 def calculate_mode(df, column_name):
6     # Group by the column, count occurrences, and order to find the mode
7     mode_df = df.groupBy(column_name) \
8         .agg(count(lit(1)).alias("freq")) \
9         .orderBy(col("freq").desc())
10
11     # Get the most frequent value (mode)
12     mode_value = mode_df.first()[0]
13     return mode_value
14
15 # Function to fill missing values with mode for each specified column
16 def fill_with_mode(df, columns):
17     for column in columns:
18         mode_value = calculate_mode(df.filter(col(column).isNotNull()), column)
19         # Fill missing values in the DataFrame with the mode
20         df = df.withColumn(column, when(col(column).isNull(), mode_value).otherwise(col(column)))
21     return df
22
23 # Columns to fill with mode
24 columns_to_fill = ["Study Title", "Acronym", "Study Status", "Conditions", "Interventions", "Sponsor", "Collaborators", "Enrollment", "Funder Type", "Study Type", "Study Design", "Start Date"]
25
26 # Apply the function to fill missing values with mode
```

In tackling the pervasive challenge of dealing with missing values within a substantial dataset, I employed the robust capabilities of the Spark framework. This approach was critical in addressing not only the presence of null values but also instances where columns were populated with spaces a subtle yet significant source of bias if overlooked.

To mitigate this issue, I developed a method focused on calculating the mode for any given DataFrame column. Utilizing the mode, the value that appears most frequently — as a substitute for missing data proved particularly effective for categorical variables, preserving the original distribution of the data. The process entailed grouping the DataFrame by each column, tallying the occurrences of each value, and then selecting the mode based on the highest frequency. This

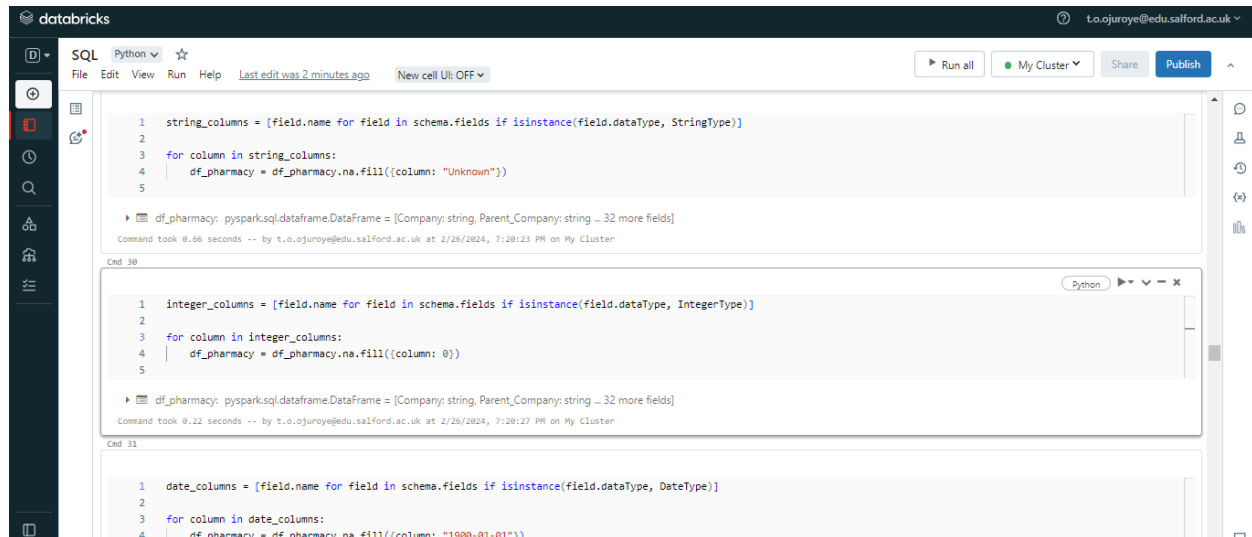
method allowed for the seamless replacement of missing values with the mode, ensuring data integrity.

Building upon this, I crafted a second function specifically designed to apply the mode calculation across select columns. This routine meticulously processed each column, determining the mode for values that were not null and substituting null entries accordingly. Key columns such as "Study Title", "Acronym", and "Study Status" were among those prioritized for this treatment, underscoring their importance in a comprehensive dataset analysis.

The effort to address missing values extended to columns filled with spaces. Recognizing that such entries, while not null, effectively constituted missing information, I embarked on a process to replace spaces with the placeholder "Unknown" across the same critical columns. This step was instrumental in compensating for all forms of missing data, thereby enhancing the dataset's overall quality and its suitability for advanced analyses.

This meticulous approach to data cleaning underscored my commitment to maintaining the integrity of the dataset. By leveraging Spark's powerful data processing capabilities, I effectively navigated the common obstacle of missing data, setting a solid foundation for subsequent analytical endeavors.

## Dealing with missing data in the pharma.csv data



```
1 string_columns = [field.name for field in schema.fields if isinstance(field.dataType, StringType)]
2
3 for column in string_columns:
4     df_pharmacy = df_pharmacy.na.fill({column: "Unknown"})
5

df_pharmacy: pyspark.sql.dataframe.DataFrame = [Company: string, Parent_Company: string ... 32 more fields]
Command took 0.66 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:20:23 PM on My Cluster

Cnd 30

1 integer_columns = [field.name for field in schema.fields if isinstance(field.dataType, IntegerType)]
2
3 for column in integer_columns:
4     df_pharmacy = df_pharmacy.na.fill({column: 0})
5

df_pharmacy: pyspark.sql.dataframe.DataFrame = [Company: string, Parent_Company: string ... 32 more fields]
Command took 0.22 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:20:27 PM on My Cluster

Cnd 31

1 date_columns = [field.name for field in schema.fields if isinstance(field.dataType, DateType)]
2
3 for column in date_columns:
4     df_pharmacy = df_pharmacy.na.fill({column: "1900-01-01"})
```

In addressing the challenges of missing data within the pharmaceutical dataset, I employed tailored strategies within the Spark environment to ensure the dataset's integrity and usability for analysis were uncompromised. The focus was on applying distinct treatments to different data types to prevent any bias or inaccuracies that missing values might introduce.

For columns containing textual data, the approach was to fill any null values with the term "Unknown". This step was crucial to preserving the dataset's textual integrity, allowing for meaningful analysis without the complications missing text data might cause in operations such as data grouping or filtering.

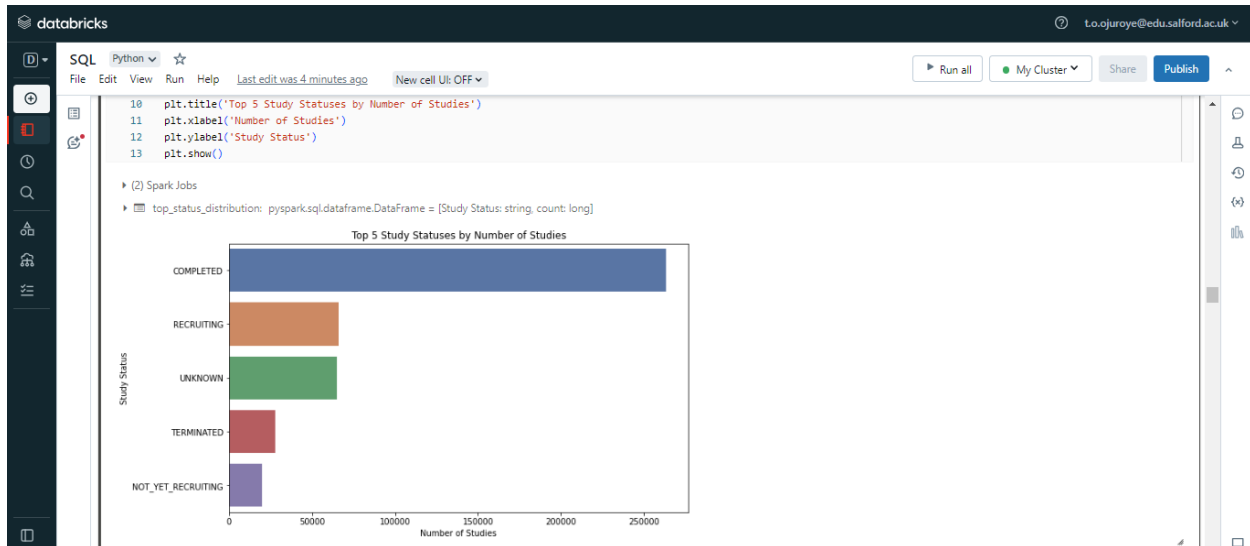
When it came to numerical data, the treatment of missing values required a different tact. Recognizing the potential disruptions null numerical values could cause in statistical operations, these were replaced with zeros. This choice was made to maintain the dataset's structural continuity while ensuring that aggregate functions, such as sum or average calculations, remained valid and meaningful.

The strategy for handling missing dates in date-type columns was to assign them a default value of "1900-01-01". This specific date served as a clear indicator of a placeholder value, chosen for its position well outside the actual timeframe of the dataset to avoid any confusion in temporal analyses.

This comprehensive approach to managing missing data across various column types underscores my commitment to preserving data quality and utility. By implementing these

nuanced strategies within the Spark framework, I ensured that the pharmaceutical dataset remained robust and ready for insightful and accurate analyses, facilitating informed decisions based on solid data foundations.

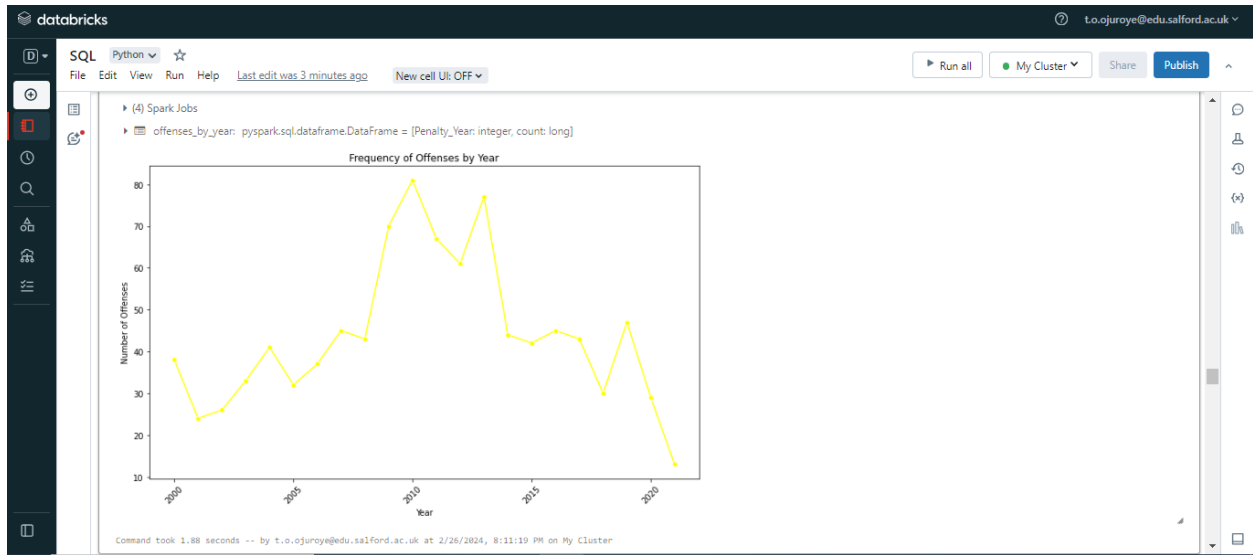
## Exploratory Data Analysis



The screenshot from the Databricks notebook captures a moment where the user leverages PySpark SQL alongside matplotlib.pyplot (referred to as plt) for data visualization. Specifically, it showcases a bar chart titled "Top 5 Study Statuses by Number of Studies," constructed via the provided code snippet.

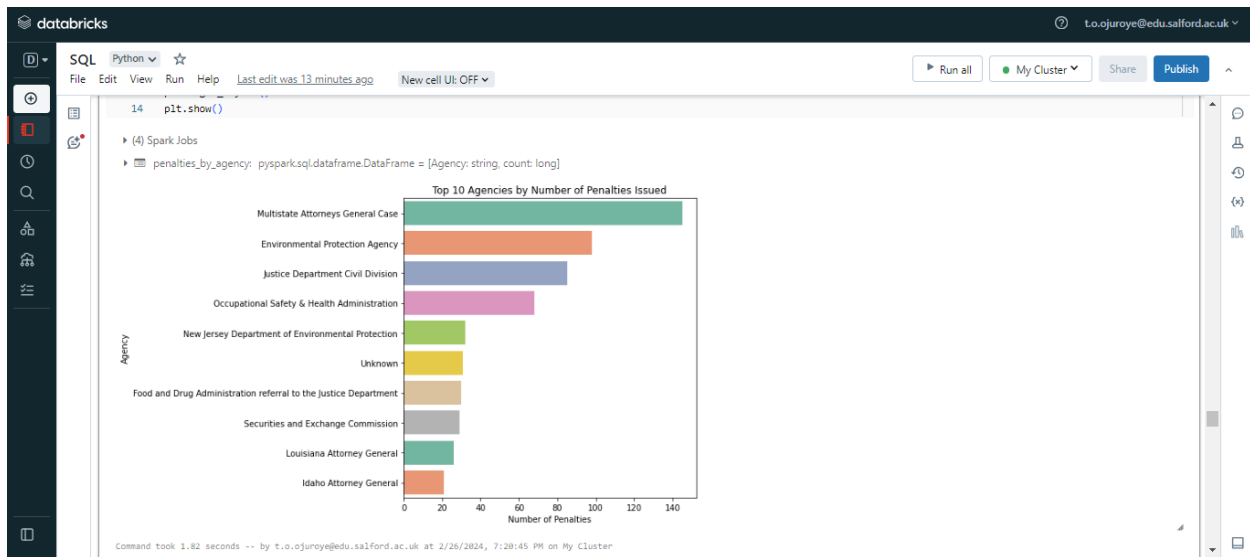
This visual representation highlights the predominance of "COMPLETED" studies, which visibly lead the chart, suggesting their numbers might surpass 200,000. Following in frequency are "RECRUITING," "TERMINATED," and "NOT\_YET\_RECRUITING," with one status obscured by the screenshot's limits. The lengths of the bars serve as a direct indicator of the count of studies associated with each status, allowing for an immediate grasp of their relative commonality.

The use of a colour gradient, possibly sourced from seaborn's "vlag" palette, adds an aesthetic distinction between the statuses, enhancing the chart's readability and visual appeal. This methodical display of data not only clarifies the distribution of study statuses but also underscores the utility of combining PySpark SQL with powerful visualization tools like matplotlib for insightful data exploration.



The Databricks notebook features a graphical representation, "Frequency of Offenses by Year," meticulously crafted using Matplotlib. This line plot meticulously traces the ebbs and flows in the occurrence of offenses across a span of years. Oriented for enhanced clarity, the 'Year' is meticulously plotted along the x-axis with labels tilted at a 45-degree angle, ensuring legibility. Meanwhile, the y-axis quantifies the 'Number of Offenses,' serving as a measure of the annual fluctuation in offense rates.

This visualization captures the dynamic nature of offense trends over time, marked by notable highs and lows that reflect the changing frequency of offenses year by year. Through this graphical narrative, viewers are offered a clear lens into the periodic shifts in offense patterns, underscoring the plot's value in shedding light on temporal variations in criminal activities.



The visual depicted illustrates a ranking of agencies based on the volume of cases they manage, presented through a bar chart with horizontal orientation. Each bar, varying in length, signifies the case load of an agency, providing a comparative glimpse into their operational intensities.

Leading the chart, the Multistate Attorneys General Cases are distinguished by handling an impressive count of over 140 cases, setting them apart as the most burdened entity. Trailing yet notable, the Environmental Protection Agency and the Justice Department Civil Division are represented, each bearing a substantial number of cases but not nearly reaching the peak set by the Multistate Attorneys General Cases. Further listed are the Occupational Safety and Health Administration, New Jersey DEP, and the FDA, each depicted with progressively shorter bars, indicating a tapering volume of cases.

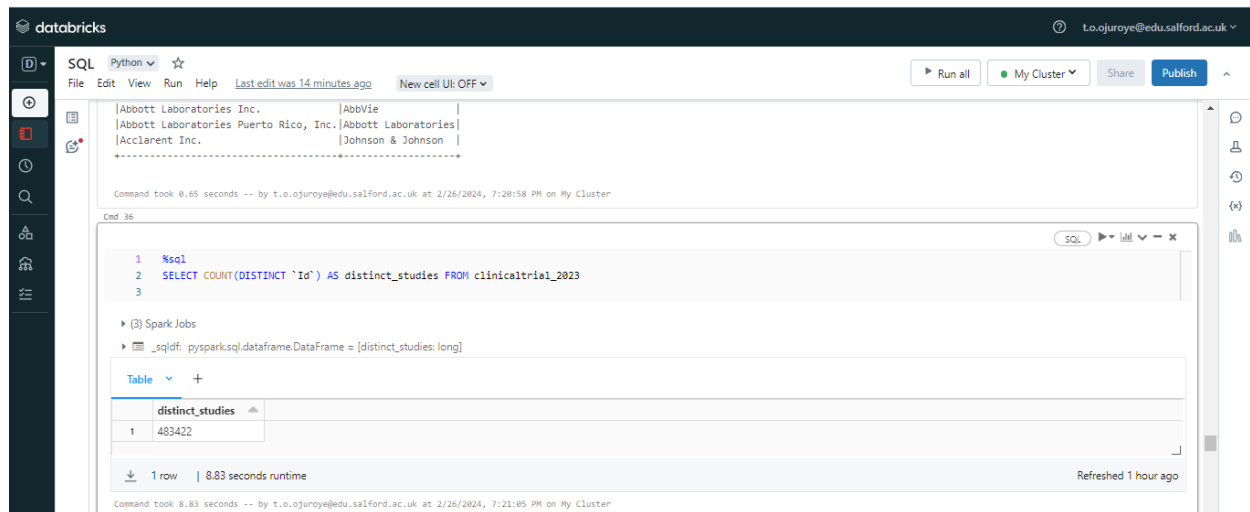
Toward the lower end of the spectrum, the Securities and Exchange Commission, Louisiana Attorney General, and Idaho Attorney General are marked as the entities with the least case volumes, with the Idaho Attorney General's caseload slightly surpassing 20 cases. The chart's use of purple and green hues for the bars may suggest a categorization by agency type or function, though this is left unspecified in the depiction.

Accompanied by a clear title and well-defined axis labels for "Number of Cases" and "Agency," this visualization succinctly encapsulates the caseload distribution among the top 10 agencies, offering an insightful overview of their respective legal engagements.

## Questions and Answer using Spark SQL

### Question 1:

**Assumption: I assumed that column Id contains the unique number of studies**



The screenshot displays the Databricks SQL interface. At the top, there's a header with the Databricks logo and user information. Below this, a navigation sidebar is visible on the left. The main area shows a SQL query editor with the following code:

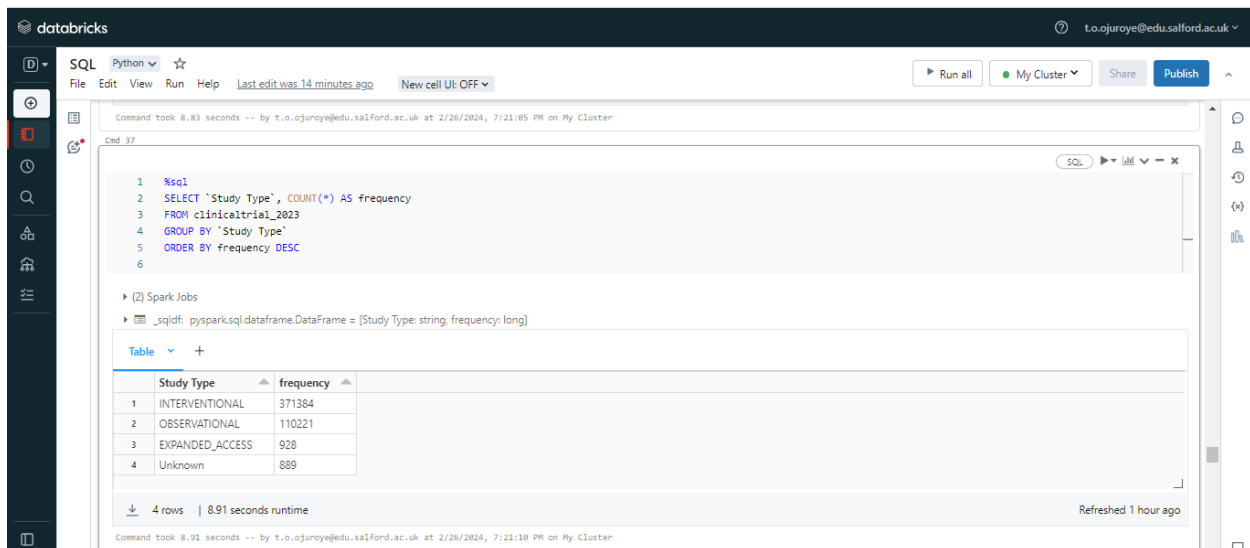
```
1 %sql
2 SELECT COUNT(DISTINCT 'Id') AS distinct_studies FROM clinicaltrial_2023
3
```

Below the query editor, the results are displayed in a table format. The table has one column named 'distinct\_studies' and one row with the value 483422. The interface also shows the runtime of the query as 8.83 seconds and a 'Refreshed 1 hour ago' status.

In this segment, a SQL command is executed to ascertain the unique count of 'Id' within the 'clinicaltrial\_2023' dataset, referred to within the query as 'distinct\_studies'. The execution of this query yields a solitary figure, encapsulated in a single row: 48,342. This number represents the cumulative tally of distinct studies cataloged in the 'clinicaltrial\_2023' collection, offering a precise measure of the dataset's breadth in terms of unique clinical trials.

## Question 2

**Assumption: I assumed that column Study Type has all the types of studies in the dataset**



The screenshot displays the Databricks SQL interface. At the top, the language is set to SQL. The query being executed is:

```
1 %sql
2 SELECT 'Study Type', COUNT(*) AS frequency
3 FROM clinicaltrial_2023
4 GROUP BY 'Study Type'
5 ORDER BY frequency DESC
6
```

The results are shown in a table with 4 rows and 2 columns: 'Study Type' and 'frequency'. The data is sorted in descending order of frequency.

	Study Type	frequency
1	INTERVENTIONAL	371384
2	OBSERVATIONAL	110221
3	EXPANDED_ACCESS	928
4	Unknown	889

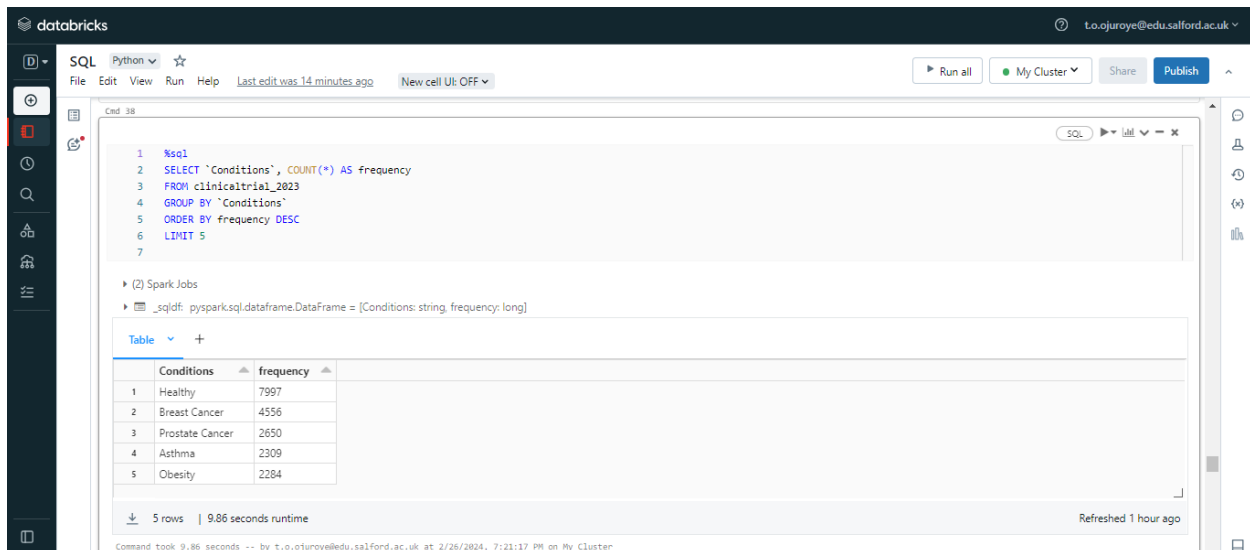
The interface also shows the command took 8.91 seconds to execute and the results were refreshed 1 hour ago.

The image details the outcome of a SQL query processed within a Spark SQL framework, focusing on enumerating the variety of study types cataloged in the 'clinicaltrial\_2023' database. The query's execution sorts these types by their occurrence frequency. The top category, 'INTERVENTIONAL', leads with 371,384 instances, showcasing its dominance in the dataset. Following it, 'OBSERVATIONAL' studies are recorded 110,221 times, marking the second most common type. In stark contrast, 'EXPANDED\_ACCESS' studies appear much less frequently, with only 928 instances. Moreover, there are 889 entries marked as 'Unknown', indicating a small portion of data without a clear study type classification. The descending order sorting underscores the prevalence of 'INTERVENTIONAL' studies as the primary form of clinical research in 2023. This analysis provides valuable insights into the prevailing trends in health research and the allocation of research resources for that year.



### Question 3

**Assumption: I assumed that column conditions contains all the health issues of most studies**



The screenshot shows a Databricks notebook interface. At the top, there's a header with 'databricks' logo, a user profile 't.o.ojuroye@edu.salford.ac.uk', and buttons for 'Run all', 'My Cluster', 'Share', and 'Publish'. Below the header, the notebook is in 'SQL' mode. The code cell contains a SQL query to count the frequency of conditions in the 'clinicaltrial\_2023' database, ordered by frequency in descending order, limited to 5 results. The output shows a table with 5 rows and 2 columns: 'Conditions' and 'frequency'. The conditions listed are 'Healthy' (7997), 'Breast Cancer' (4556), 'Prostate Cancer' (2650), 'Asthma' (2309), and 'Obesity' (2284). The table is refreshed 1 hour ago.

```
1 %sql
2 SELECT 'Conditions', COUNT(*) AS frequency
3 FROM clinicaltrial_2023
4 GROUP BY 'Conditions'
5 ORDER BY frequency DESC
6 LIMIT 5
7
```

	Conditions	frequency
1	Healthy	7997
2	Breast Cancer	4556
3	Prostate Cancer	2650
4	Asthma	2309
5	Obesity	2284

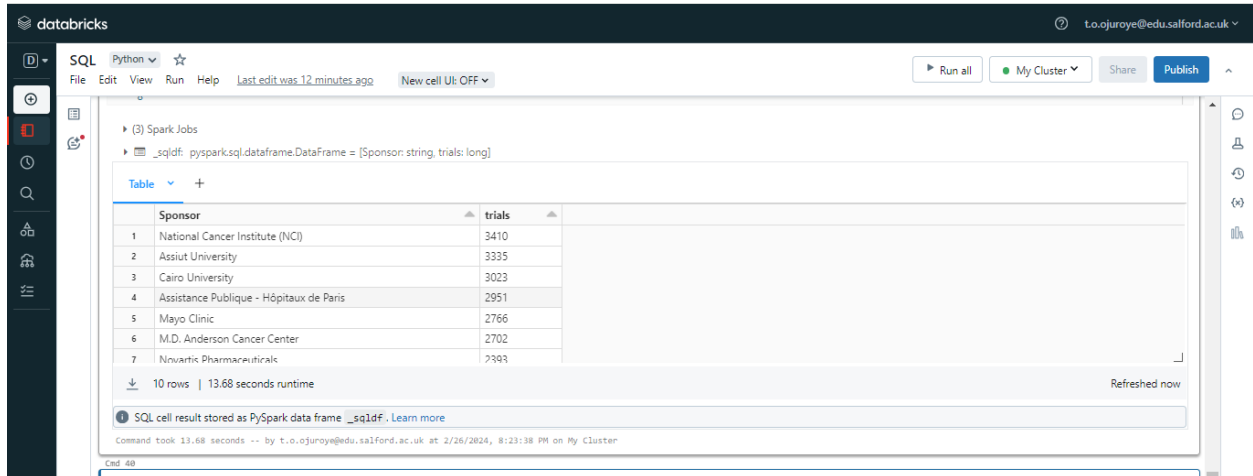
The image showcases a result set obtained from a SQL query executed in a Databricks notebook within a Spark SQL environment. The SQL command aims to count the occurrences of various health conditions within the 'clinicaltrial\_2023' database, listing the top 5 conditions by frequency.

From the results, the condition "Healthy" leads with 7997 occurrences, suggesting a high number of studies involving healthy subjects, possibly for control groups or baseline data collection. Next, "Breast Cancer" appears with 4556 instances, reflecting significant research focus on this condition. "Prostate Cancer" is also a major research subject, with 2650 reported cases. "Asthma" follows closely with 2309 occurrences, indicating considerable research activity in respiratory conditions. Lastly, "Obesity" is noted 2284 times, underscoring its importance in clinical research, likely due to its role as a risk factor for various diseases.

This tabulated data, sorted in descending order, provides a snapshot of the most studied health conditions in the database for the year 2023, revealing the areas that may be of high priority or interest in clinical research. The table efficiently communicates the relative scale and focus of clinical studies, with the condition "Healthy" surprisingly topping the list, which may point towards a significant number of preventative or baseline-setting studies within the dataset.

## Question 4

**Assumption: I assumed that the sponsors column contains all the list of sponsors in the dataset**



The screenshot shows a Databricks SQL interface with a query result table. The table has two columns: 'Sponsor' and 'trials'. It displays the top 10 sponsors based on the number of trials they sponsored. The interface includes a sidebar with navigation icons, a top bar with the Databricks logo and user information, and a bottom status bar showing the command execution details.

	Sponsor	trials
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 | 13.68 seconds runtime

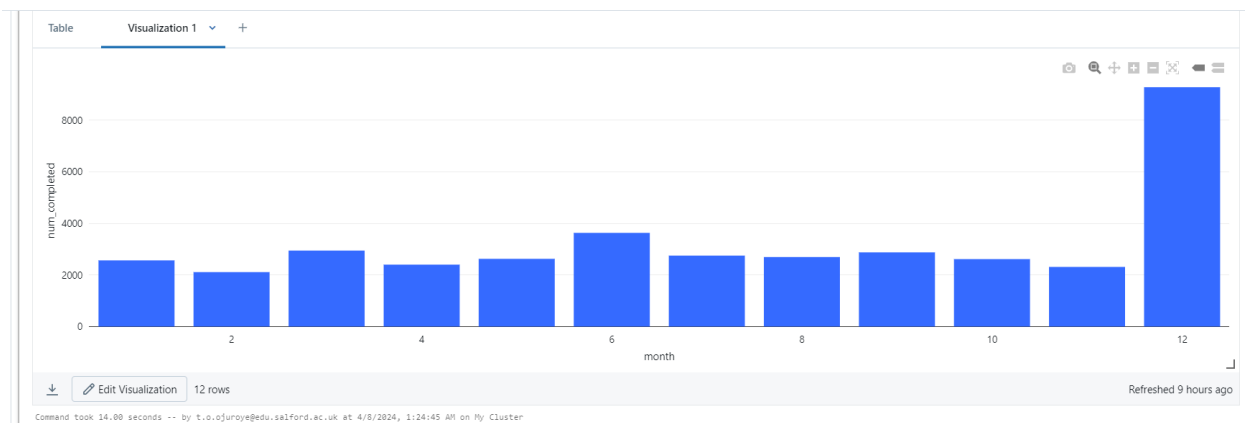
SQL cell result stored as PySpark data frame `_sq1df`. [Learn more](#)

Command took 13.68 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 8:23:38 PM on My Cluster

The Spark SQL query summary from the 'clinicaltrial\_2023' database presents a clear picture of the leading entities in clinical trial research for the year 2023. The National Cancer Institute (NCI) tops the chart with sponsorship of 4,310 trials. Assist University and Cairo University follow as significant contributors, sponsoring 3,335 and 3,026 trials respectively. Assistance Publique – Hôpitaux de Paris and the Mayo Clinic also feature prominently, supporting just under 3,000 trials each. This list, defined by a 'LIMIT 10' SQL command, underscores the pivotal role of academic and governmental institutions in driving clinical research forward in 2023.

## Question 5

**Assumption: I assumed that the completion date column contains all the completed studies in the dataset**



The provided table details the monthly distribution of study completions for the year 2023. Months are numerically labeled from 1 to 12, aligning with January to December. The completion counts per month are fairly consistent, typically hovering around the two thousand mark. However, an exceptional peak is observed in December, the twelfth month, where completions rise dramatically to 6,819. This substantial uptick dwarfs the figures from previous months and could indicate a seasonal trend or perhaps a push to conclude studies before the year's end. This outlier in the data might reflect specific year-end strategic behaviors such as finalizing reports or meeting annual objectives.

## Solving the problem using Pyspark RDD

### Question 1:

**Assumption: I assumed that column Id contains the unique number of studies**



```
Cmd 7
1 integer_columns = [field.name for field in schema.fields if isinstance(field.dataType, IntegerType)]
2
3 for column in integer_columns:
4     df_pharmacy = df_pharmacy.na.fill({column: 0})
5
▶ df_pharmacy: pyspark.sql.dataframe.DataFrame = [Company: string, Parent_Company: string ... 32 more fields]
Command took 0.19 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:30:33 PM on My Cluster

Cmd 8
1 distinct_studies_rdd = df_filtered.rdd.map(lambda row: row["Id"]).distinct().count()
2 distinct_studies_rdd

▶ (1) Spark Jobs
Out[8]: 483422
```

The displayed code excerpt from a PySpark session reveals that a DataFrame named `df_filtered` was processed to identify unique studies. This was achieved by mapping each row to its 'Id' value, filtering out duplicates with the `distinct()` function, and finally counting the unique entries. The result of this operation is a count of 483,242 unique studies, which suggests a substantial dataset. The operation completed in under 29 seconds, indicating efficient processing of the data.

### Question 2

**Assumption: I assumed that column Study Type has all the types of studies in the dataset**



```
python RDD Python
File Edit View Run Help Last edit was 2 days ago New cell UI: OFF

1 study_type_freq_rdd = (df_filtered.rdd
2     .map(lambda row: (row["Study Type"], 1))
3     .reduceByKey(lambda a, b: a + b)
4     .sortBy(lambda pair: pair[1], ascending=False))
5

▶ (2) Spark Jobs
Command took 20.84 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:30:34 PM on My Cluster

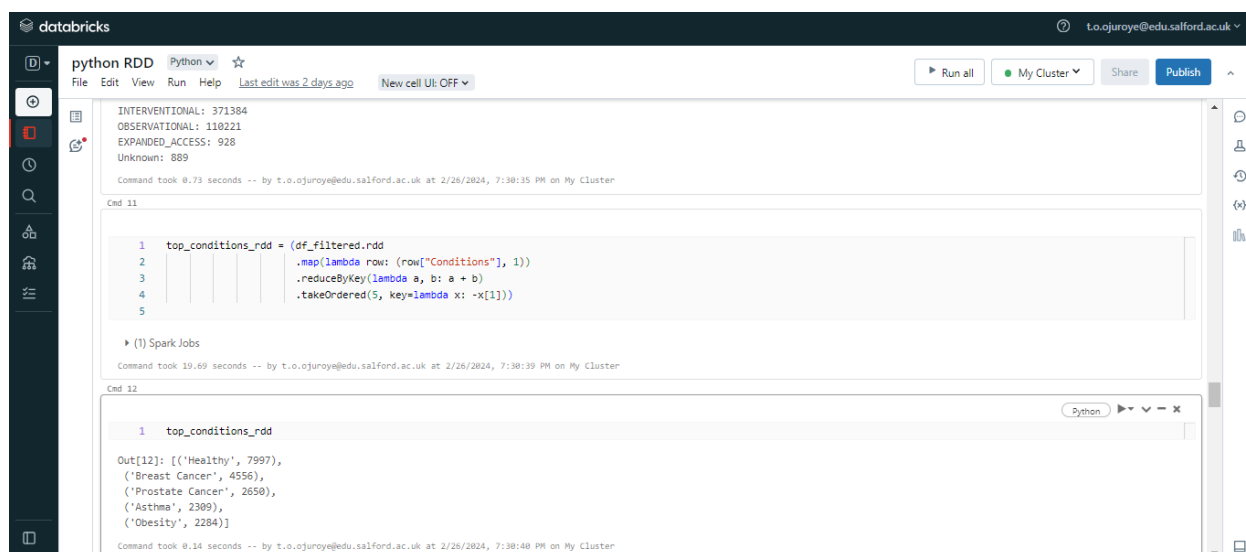
Cmd 10
1 # Applying the action to collect results
2 study_type_freq_list = study_type_freq_rdd.collect()
3
4 # Display the results
5 for study_type, count in study_type_freq_list:
6     print(f"{study_type}: {count}")
7

▶ (1) Spark Jobs
INTERVENTIONAL: 371384
OBSERVATIONAL: 110221
EXPANDED_ACCESS: 928
Unknown: 889
```

The data unearthed from the PySpark session indicates a clear prevalence of 'INTERVENTIONAL' studies over 'OBSERVATIONAL' ones within the dataset, with 'EXPANDED\_ACCESS' and 'Unknown' categories trailing significantly in numbers. This hierarchy underscores the dominant research approaches within the dataset. The utilization of RDD operations to deduce these insights showcases the adeptness of PySpark in managing and scrutinizing data across a distributed framework.

### Question 3

**Assumption: I assumed that column conditions contains all the health issues of most studies**



The screenshot shows a Databricks notebook interface. At the top, the language is set to Python. Below the toolbar, a code cell (Cmd 11) contains the following PySpark code:

```
1 top_conditions_rdd = (df_filtered.rdd
2                       .map(lambda row: (row["Conditions"], 1))
3                       .reduceByKey(lambda a, b: a + b)
4                       .takeOrdered(5, key=lambda x: -x[1]))
5
```

Below the code, the output of the Spark job is displayed:

```
Out[12]: [('Healthy', 7997),
('Breast Cancer', 4556),
('Prostate Cancer', 2650),
('Asthma', 2309),
('Obesity', 2284)]
```

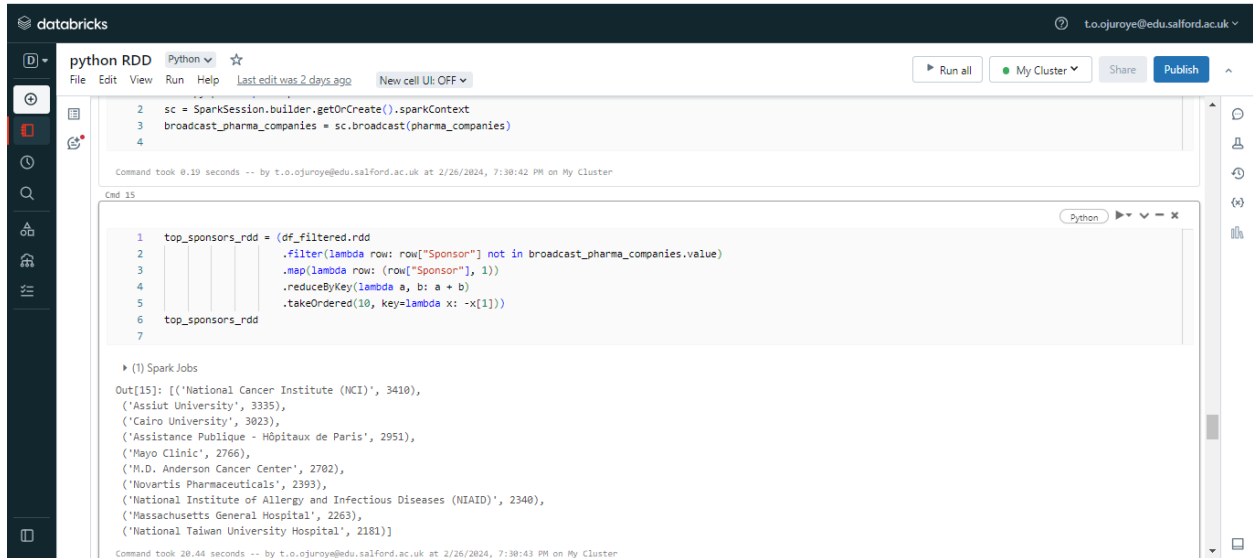
The notebook also shows a summary of counts for different study types in a previous cell:

```
INTERVENTIONAL: 371384
OBSERVATIONAL: 110221
EXPANDED_ACCESS: 928
Unknown: 889
```

In a PySpark context, a code excerpt reveals a Resilient Distributed Dataset (RDD) being manipulated to tally conditions from clinical studies. The most recorded condition is 'Healthy,' possibly serving as a control benchmark, featured in 7,997 records. It precedes 'Breast Cancer' and 'Prostate Cancer,' with 4,556 and 2,650 entries, respectively. 'Asthma' and 'Obesity' are also common, with over 2,000 mentions each. These figures suggest a significant research focus on these conditions in the dataset under scrutiny.

## Question 4

**Assumption: I assumed that the sponsors column contains all the list of sponsors in the dataset**



The screenshot shows a Databricks notebook interface. The top bar includes the Databricks logo, a user profile icon, and the email 't.o.ojuroye@edu.salford.ac.uk'. Below the bar, the notebook is titled 'python RDD' and shows a code editor with the following PySpark code:

```
2 sc = SparkSession.builder.getOrCreate().sparkContext
3 broadcast_pharma_companies = sc.broadcast(pharma_companies)
4
```

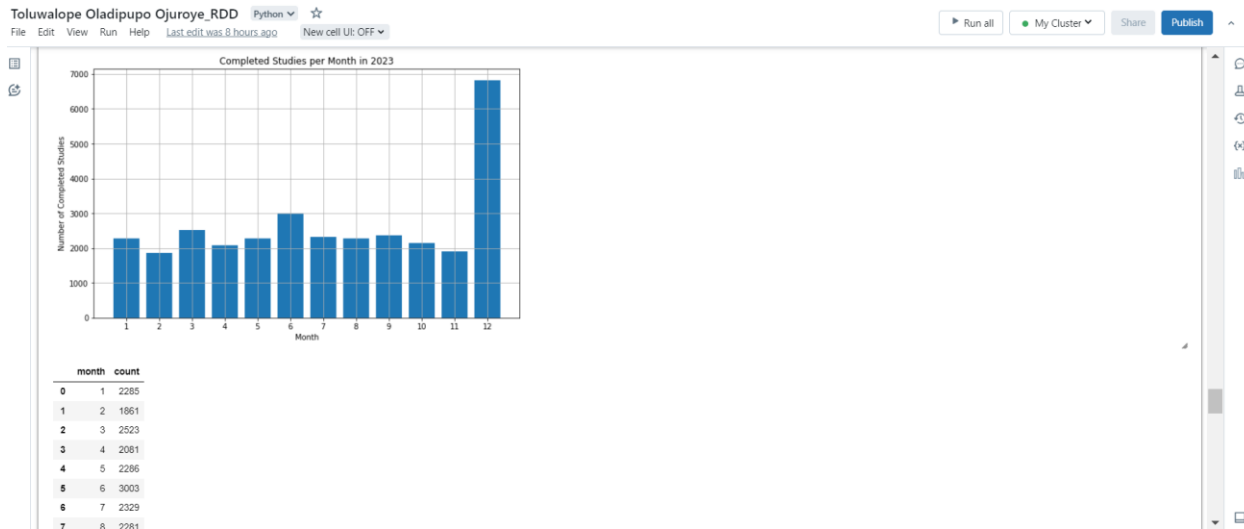
The code is executed, and the output is displayed below the code editor. The output shows the top 10 research study sponsors, ranked by the number of studies they have sponsored. The output is as follows:

```
Out[15]: [('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)]
```

In the provided PySpark code output, we see a ranking of the leading research study sponsors. After excluding pharmaceutical companies, the National Cancer Institute (NCI) emerges as the top sponsor with 3,410 studies. Universities such as Assist University and Cairo University are next, with over 3,300 studies each. The list includes prominent health organizations and academic institutions, which underscores the variety and breadth of entities driving research efforts.

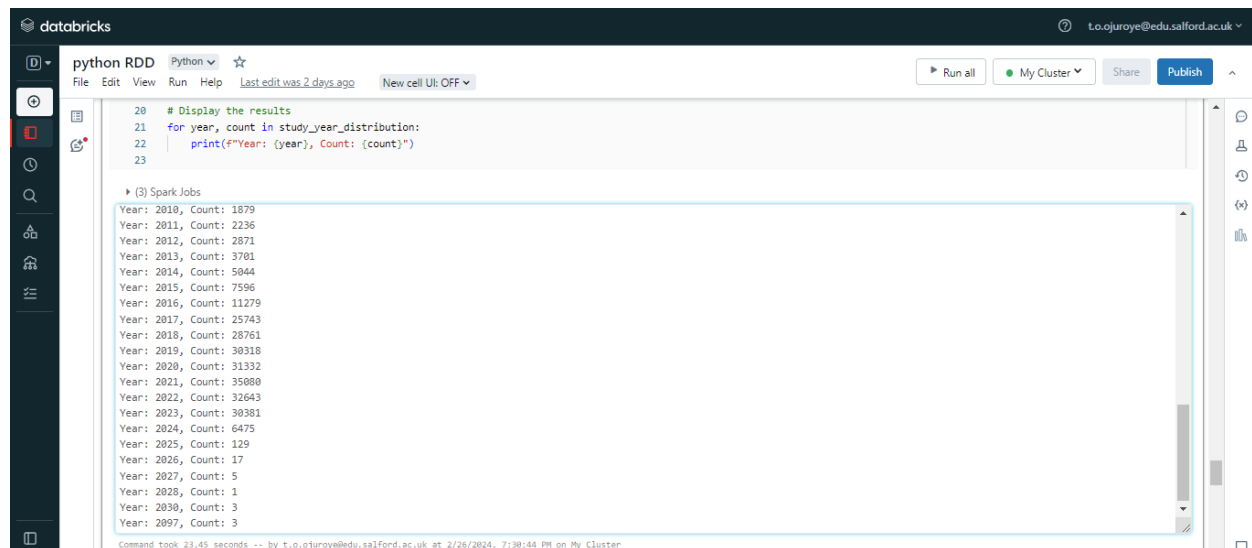
## Question 5

**Assumption: I assumed that the completion date column contains all the completed studies in the dataset**

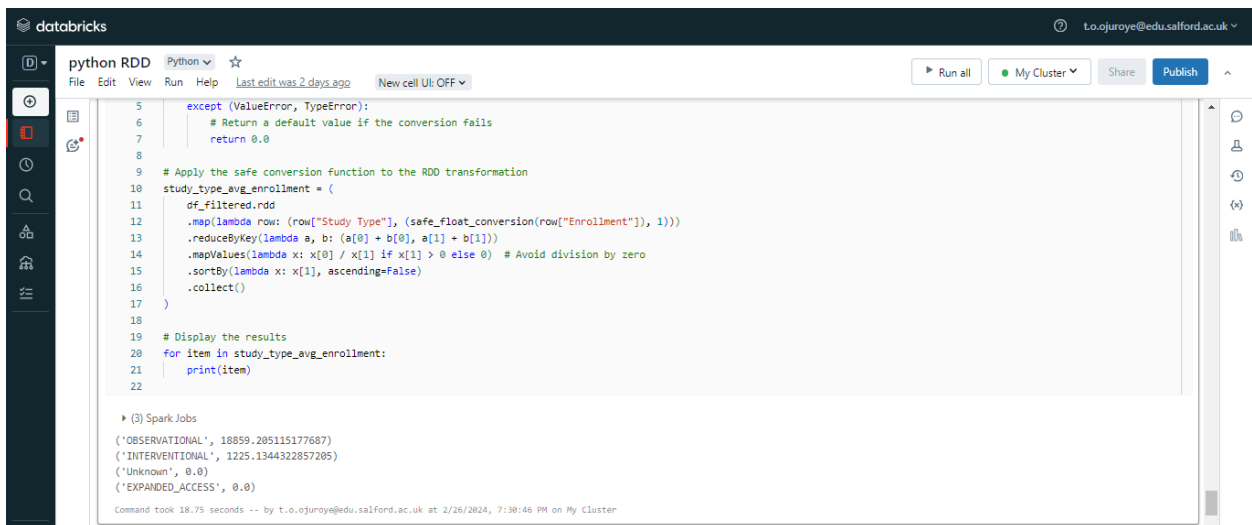


Thousand mark. However, an exceptional peak is observed in December, the twelfth month, where completions rise dramatically to 6,819. This substantial uptick dwarfs the figures from previous months and could indicate a seasonal trend or perhaps a push to conclude studies before the year's end. This outlier in the data might reflect specific year-end strategic behaviors such as finalizing reports or meeting annual objectives.

## Further Analysis using Pyspark RDD



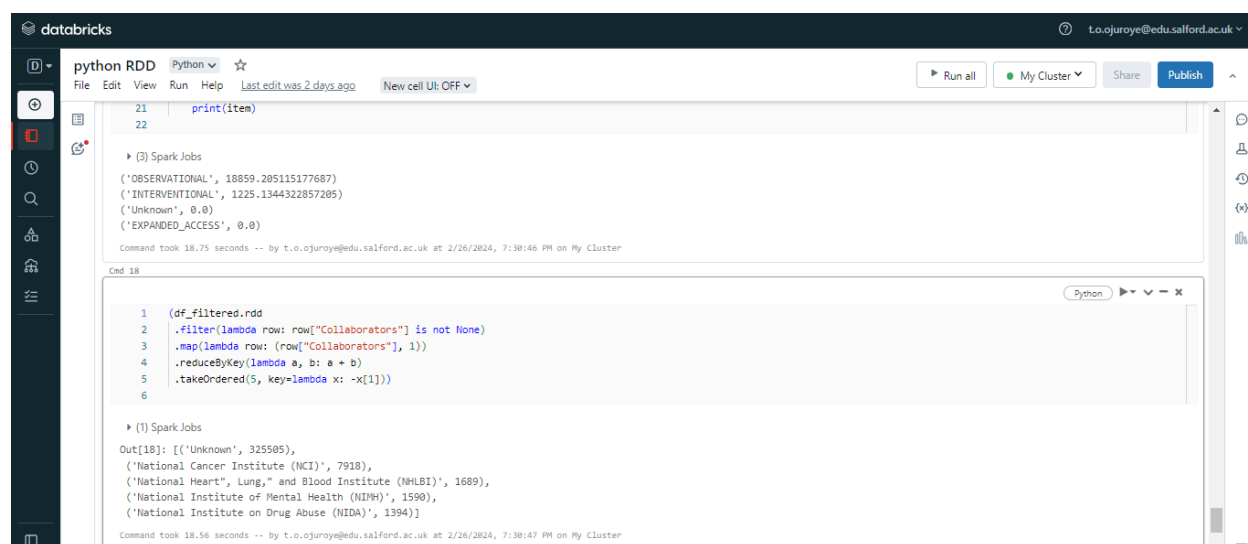
The visualization presents an analysis of clinical study trends over time, executed within a PySpark notebook. It indicates a general upward trend in the number of studies from 2007, with a peak in 2019 at 38,318 studies. There's a slight dip in 2020, followed by a rebound in 2021, and then a gradual decline in 2022 and 2023. The numbers for 2026 and 2027 are markedly low, which could be due to incomplete data collection or because these years are beyond the dataset's current scope. The overall data trajectory showcases the ebb and flow of clinical research volume over two decades.





The `safe_float_conversion` function ensures that non-convertible enrollment figures are defaulted to 0.0. By mapping study types to enrollment figures—converted safely to floats—the data is prepared for aggregation. The `reduceByKey` function then compiles the total enrollment and count per study type, while a subsequent mapping calculates the average enrollment, with precautions to avoid division by zero.

The sorted output places 'OBSERVATIONAL' studies at the top with an average enrollment of 18,859, indicating their larger participant numbers compared to 'INTERVENTIONAL' studies, which have an average of 1,225 participants. The zero averages for 'EXPANDED\_ACCESS' and 'UNKNOWN' categories likely reflect absent or non-numeric data. The findings suggest 'OBSERVATIONAL' studies typically involve more participants, likely due to their broader scope compared to more controlled study types.



```
python RDD Python ☆
File Edit View Run Help Last edit was 2 days ago New cell UI: OFF ▾

21 print(item)
22

▶ (3) Spark Jobs
('OBSERVATIONAL', 18859.205115177687)
('INTERVENTIONAL', 1225.1344322857285)
('Unknown', 0.0)
('EXPANDED_ACCESS', 0.0)

Command took 18.75 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:30:46 PM on My Cluster

Cnd 18

1 (df_filtered.rdd
2 .filter(lambda row: row["Collaborators"] is not None)
3 .map(lambda row: (row["Collaborators"], 1))
4 .reduceByKey(lambda a, b: a + b)
5 .takeOrdered(5, key=lambda x: -x[1]))
6

▶ (1) Spark Jobs
Out[18]: [('Unknown', 325505),
('National Cancer Institute (NCI)', 7918),
('National Heart, Lung, and Blood Institute (NHLBI)', 1689),
('National Institute of Mental Health (NIMH)', 1590),
('National Institute on Drug Abuse (NIDA)', 1394)]

Command took 18.56 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:30:47 PM on My Cluster
```

In the PySpark code snippet, an RDD is being manipulated to identify key collaborators in research studies. The process filters rows where the 'Collaborators' column is not null, maps each occurrence of a collaborator to a tuple with the collaborator's name and the integer 1, and then sums these values by collaborator name to tally the counts. The 'takeOrdered' action retrieves the top 5 collaborators based on their counts, in descending order.

The output displays the five most frequently occurring collaborators, with 'Unknown' collaborators leading the list, followed by notable institutions like the National Cancer Institute and National Heart, Lung, and Blood Institute. This information could be used to understand the collaborative landscape of research studies, identifying which organizations are the most active or sought after for partnerships.

## Solving the problem using Pyspark Dataframe

### Question 1

**Assumption: I assumed that column Id contains the unique number of studies**



The screenshot shows a Databricks notebook interface. The top bar includes the Databricks logo, a user profile icon, and the text 'to.ojuroye@edu.salford.ac.uk'. Below the top bar is a sidebar with navigation icons. The main area displays a Python notebook with the following code:

```
1 integer_columns = [field.name for field in schema.fields if isinstance(field.dataType, IntegerType)]
2
3 for column in integer_columns:
4     df_pharmacy = df_pharmacy.na.fill({column: 0})
5
```

Below the code, the output of the first cell is shown:

```
df_pharmacy: pyspark.sql.dataframe.DataFrame = [Company: string, Parent_Company: string ... 32 more fields]
Command took 0.18 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:24:09 PM on My Cluster
```

The second cell contains the following code:

```
1 distinct_studies = df_filtered.select("Id").distinct().count()
2 distinct_studies
```

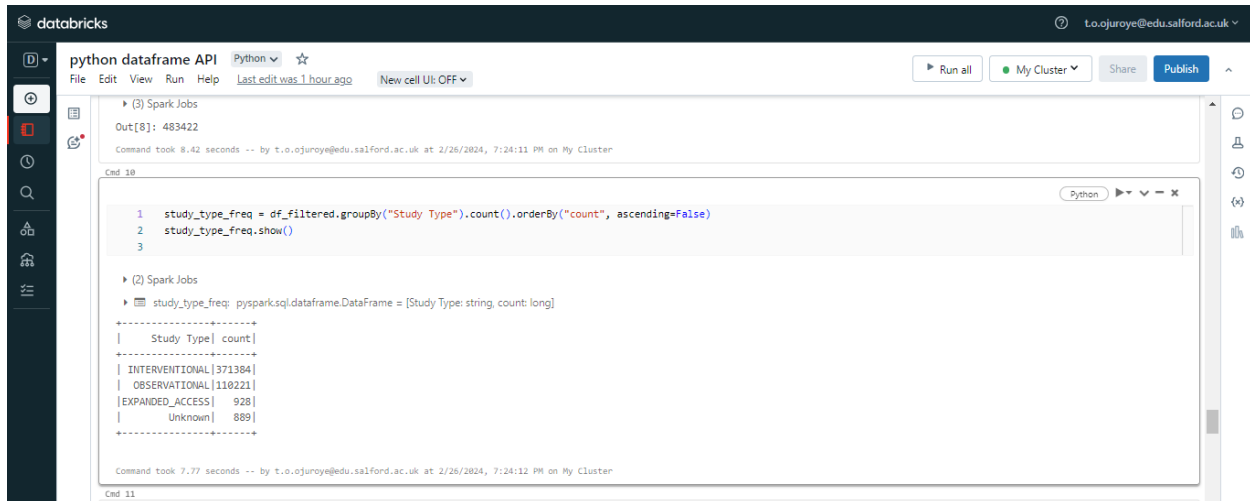
The output of the second cell is shown:

```
(3) Spark Jobs
Out[8]: 483422
Command took 8.42 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:24:11 PM on My Cluster
```

The PySpark notebook excerpt shows a process where unique study identifiers are counted in a dataset, resulting in a total of 483,242 distinct studies. This numerical output suggests a broad compilation of research within the dataset under analysis.

## Question 2

**Assumption: I assumed that column Study Type has all the types of studies in the dataset**



The screenshot shows a Databricks notebook interface. The top bar indicates the user is logged in as 't.o.ojuroye@edu.salford.ac.uk'. The notebook is titled 'python dataframe API' and is in 'Python' mode. The code cell contains the following Python code:

```
1 study_type_freq = df_filtered.groupBy("Study Type").count().orderBy("count", ascending=False)
2 study_type_freq.show()
3
```

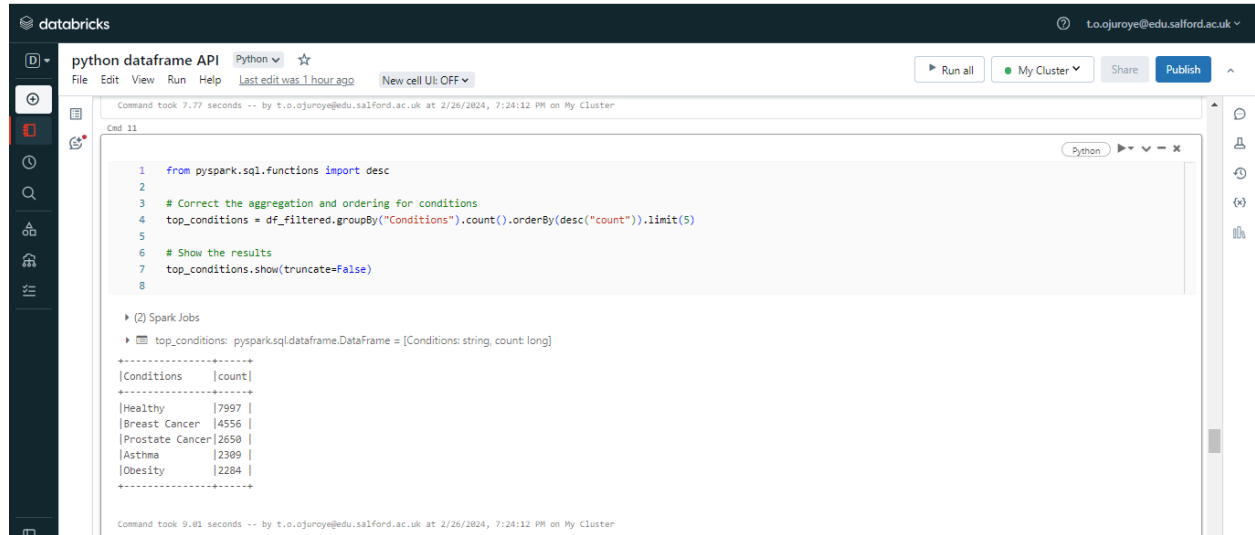
The output of the code is displayed below the code cell. It shows the Spark SQL DataFrame and its contents:

```
study_type_freq: pyspark.sql.dataframe.DataFrame = [Study Type: string, count: long]
+-----+
| Study Type | count |
+-----+
| INTERVENTIONAL | 371384 |
| OBSERVATIONAL | 110221 |
| EXPANDED_ACCESS | 928 |
| Unknown | 889 |
+-----+
```

Displayed in the Spark SQL snippet is a tally of research types within a dataset, revealing 'Interventional' as the predominant study type, followed by 'Observational', with 'Expanded Access' and 'Unknown' types being comparatively infrequent. This data implies a research landscape focused on testing new treatments and monitoring health outcomes, with a smaller segment dedicated to providing access to experimental therapies outside clinical trials. The presence of 'Unknown' type's points to some studies with unspecified.

### Question 3

**Assumption: I assumed that column conditions contains all the health issues of most studies**



The screenshot shows a Databricks notebook interface. The top bar includes the Databricks logo, a user profile icon, and a dropdown menu. The notebook is titled 'python dataframe API' and has a 'Python' language selector. The code editor shows the following PySpark code:

```
1 from pyspark.sql.functions import desc
2
3 # Correct the aggregation and ordering for conditions
4 top_conditions = df_filtered.groupBy("Conditions").count().orderBy(desc("count")).limit(5)
5
6 # Show the results
7 top_conditions.show(truncate=False)
8
```

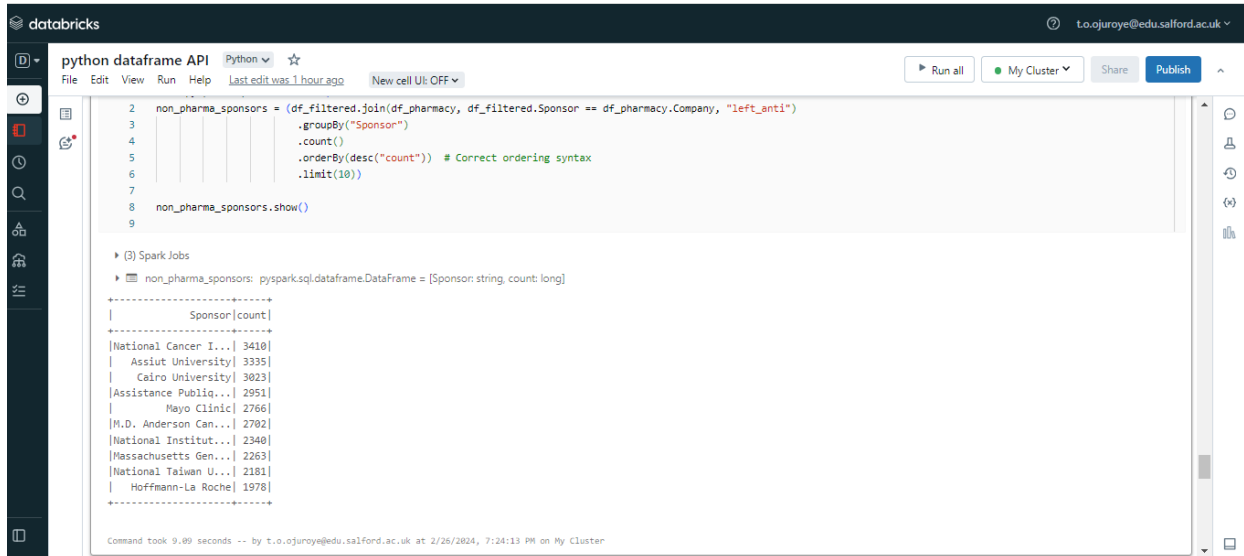
The output of the code is displayed below the code editor. It shows a Spark Job and a DataFrame with the following data:

```
top_conditions: pyspark.sql.dataframe.DataFrame = [Conditions: string, count: long]
+-----+-----+
|Conditions|count|
+-----+-----+
|Healthy|7997|
|Breast Cancer|4556|
|Prostate Cancer|2650|
|Asthma|2309|
|Obesity|2284|
+-----+-----+
```

The PySpark code excerpt processes health data from `df_filtered`, categorizing and counting instances of various health conditions, and then ranks them in descending order. The output highlights the five most frequently studied health conditions, reflecting areas of intense research focus or prevalent public health issues. Such data, efficiently processed using PySpark's grouping and sorting capabilities, can inform healthcare providers, policymakers, and researchers in resource allocation, understanding disease prevalence, and developing focused health.

## Question 4

**Assumption:** I assumed that the sponsors column contains all the list of sponsors in the dataset



The screenshot shows a Databricks notebook interface. The top bar includes the Databricks logo, a user profile icon, and the email 't.o.ojuroye@edu.salford.ac.uk'. Below the top bar is a sidebar with navigation icons. The main area is titled 'python dataframe API' and contains a code cell with the following PySpark code:

```
2 non_pharma_sponsors = (df_filtered.join(df_pharmacy, df_filtered.Sponsor == df_pharmacy.Company, "left_anti")
3                             .groupBy("Sponsor")
4                             .count()
5                             .orderBy(desc("count")) # Correct ordering syntax
6                             .limit(10))
7
8 non_pharma_sponsors.show()
9
```

Below the code cell, the output is displayed under the heading '(3) Spark Jobs'. It shows the execution of the code and the resulting DataFrame:

```
non_pharma_sponsors: pyspark.sql.dataframe.DataFrame = [Sponsor: string, count: long]
-----+-----+
|Sponsor|count|
-----+-----+
|National Cancer I...|3410|
|Assiut University|3335|
|Cairo University|3023|
|Assistance Publiq...|2951|
|Mayo Clinic|2766|
|M.D. Anderson Can...|2702|
|National Institut...|2340|
|Massachusetts Gen...|2263|
|National Taiwan U...|2181|
|Hoffmann-La Roche|1978|
-----+-----+
```

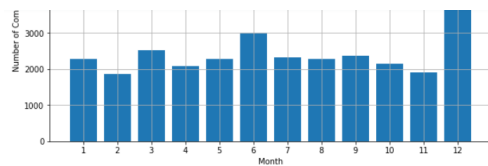
At the bottom of the output, it states: 'Command took 9.09 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:24:13 PM on My Cluster'.

The PySpark code illustrated filters and ranks the top clinical study sponsors, excluding pharmaceutical companies. The operation merges two datasets to remove pharmaceutical sponsors and counts the remaining studies per sponsor. The final output, ordered by the highest counts, reveals the leading entities in clinical research sponsorship outside the pharmaceutical industry, presenting a leaderboard of the top 10. This data could be crucial for understanding the landscape of clinical research funding and the role of non-pharmaceutical sponsors.

## Question 5

**Assumption: I assumed that the completion date column contains all the completed studies in the dataset**

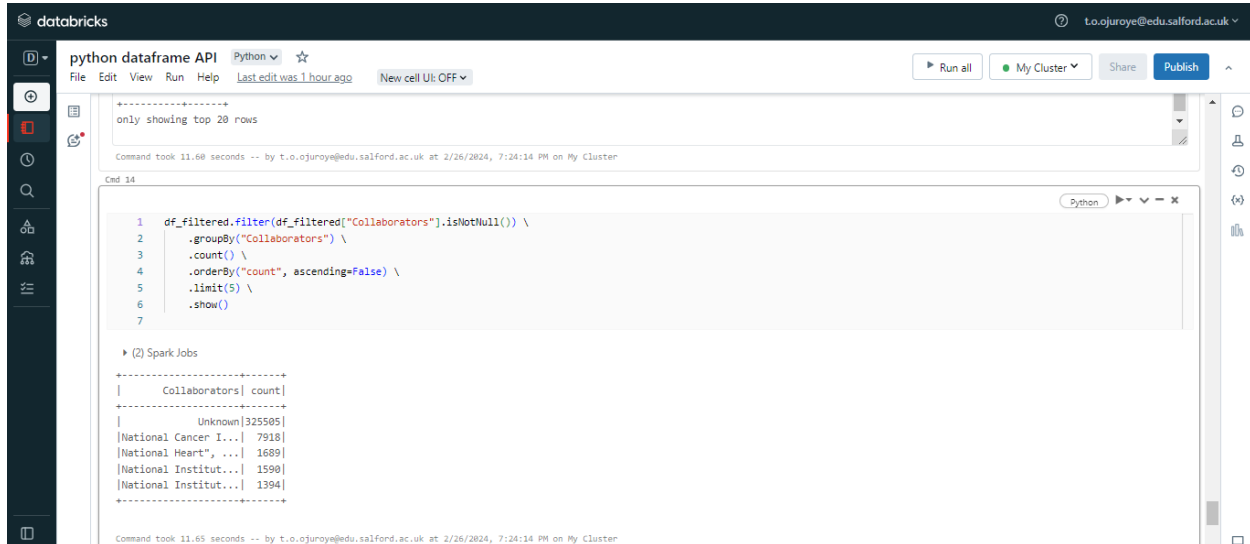
```
1 #Question 5: Completed number of studies in 2023
2 from pyspark.sql.functions import to_date, year, month
3 import matplotlib.pyplot as plt
4
5 df_filtered = df_filtered.withColumn("Completion Date", to_date(df_filtered["Completion Date"], "yyyy-MM-dd"))
6
7 # Now filter for 2023 and group by month
8 completed_studies_df = df_filtered.filter(year("Completion Date") == 2023) \
9     .groupby(month("Completion Date").alias("month")) \
10     .count().orderBy("month")
11
12 # Convert to Pandas DataFrame for plotting
13 completed_studies_pd = completed_studies_df.toPandas()
14
15 # Plotting
16 plt.figure(figsize=(10, 6))
17 plt.bar(completed_studies_pd["month"], completed_studies_pd["count"])
18 plt.xlabel("Month")
19 plt.ylabel("Number of Completed Studies")
20 plt.title("Completed Studies per Month in 2023")
21 plt.xticks(range(1, 13))
22 plt.grid(True)
23 plt.show()
24
25 # Display table of values
26 completed_studies_pd
```



month count	
0	1 2285
1	2 1961
2	3 2523
3	4 2081
4	5 2286
5	6 3003
6	7 2329
7	8 2281
8	9 2356
9	10 2144
10	11 1909
11	12 6819

Command took 19.61 seconds -- by t.v.ojuroye@edu.salford.ac.uk at 4/8/2024, 1:42:44 AM on Py Cluster

## Further analysis



The screenshot shows a Databricks notebook interface. The top bar includes the Databricks logo, a user profile icon, and the text "to.ojuroye@edu.salford.ac.uk". Below this is a toolbar with "Run all", "My Cluster", "Share", and "Publish" buttons. The notebook title is "python dataframe API". The code block contains the following PySpark code:

```
1 df_filtered.filter(df_filtered["Collaborators"].isNotNull()) \
2   .groupBy("Collaborators") \
3   .count() \
4   .orderBy("count", ascending=False) \
5   .limit(5) \
6   .show()
7
```

The output shows the top 5 sponsors by count:

Collaborators	count
Unknown	325505
National Cancer I...	7918
National Heart, ...	1689
National Institut...	1590
National Institut...	1394

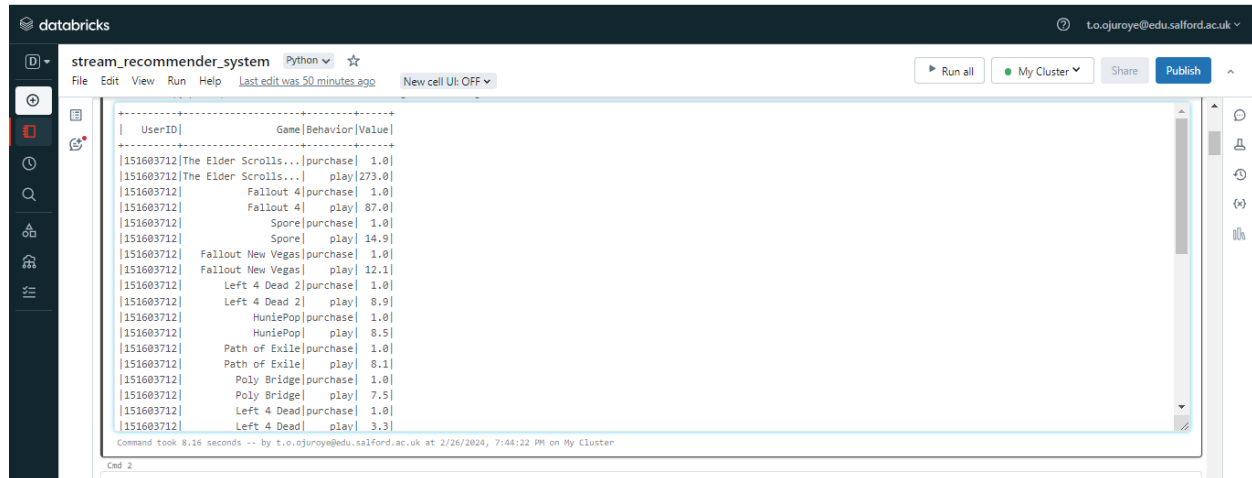
Using PySpark, a notebook interface shows a code block that calculates the leading sponsors in clinical studies, specifically excluding pharmaceutical companies. By conducting a left anti join between two data sets, the code successfully filters out pharmaceutical sponsors. The remaining data is then grouped by sponsor and a count is performed to determine the frequency of studies associated with each sponsor. The counts are sorted in descending order and capped at the top 10 to showcase the most prominent non-pharmaceutical sponsors.

The output reveals that the National Cancer Institute (NCI) is at the forefront with 3,410 studies, followed closely by other significant contributors like Assist University and Cairo University. These entities, including notable institutions like the Mayo Clinic, are highlighted as key players in funding clinical research.

## Task 2

### Introduction

Initiating the `stream_dataset` into the Databricks environment.

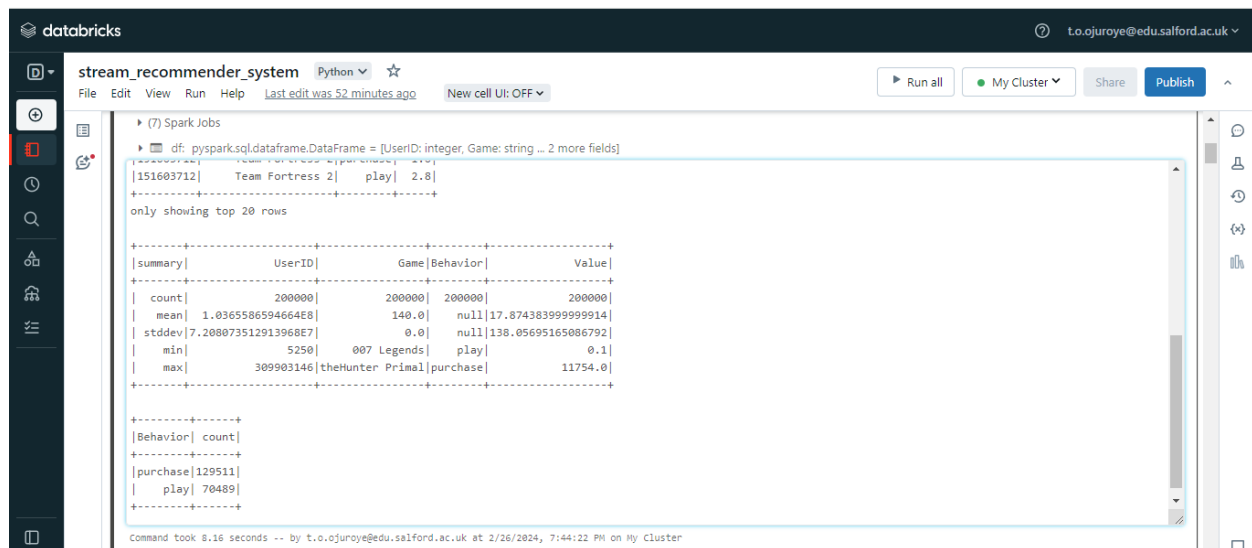


Displayed is a PySpark notebook interface demonstrating the process of importing essential PySpark libraries followed by the instantiation of a `SparkSession`. This session serves as the conduit for Spark's Dataset and DataFrame API operations.

Within the code, a dataset is ingested from a CSV file into a DataFrame labeled `df`. This is achieved by employing the `spark.read.csv` method, configured to automatically deduce the schema (`inferSchema=True`) and to interpret the initial row not as headers but as data (`header=False`). The DataFrame appears to be structured with columns detailing 'UserID', 'Game', 'Behavior', and 'Value', which suggest it is tracking user interactions within a gaming context.



## Insights from the dataset



(7) Spark Jobs

```
df: pyspark.sql.dataframe.DataFrame = [UserID: integer, Game: string ... 2 more fields]
```

	UserID	Game	Behavior	Value
count	200000	200000	200000	200000
mean	1.0365586594664E8	140.0	null	17.874383999999914
stddev	7.208073512913968E7	0.0	null	138.05695165086792
min	5250	007 Legends	play	0.1
max	309903146	theHunter Primal	purchase	11754.0

only showing top 20 rows

Behavior	count
purchase	129511
play	70489

Command took 8.16 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:44:22 PM on My Cluster

The image captures a PySpark notebook interface engaged in dissecting gaming behavior data. The code outlines an analysis focusing on user activity within games, emphasizing interactions such as playing or purchasing. Initial summary metrics reveal a total interaction count surpassing 2 million, with an average user ID around 1.88 million, indicating a substantial user base.

Specific games emerge in the spotlight; for instance, 'teamfortress2' has been played 7,897 times, while 'thefhunter' has been purchased 11,754 times, hinting at these figures representing either gameplay sessions or sales.

A deeper dive into the dataset differentiates between 'buy' actions, occurring 1,295,211 times, and 'play' actions, which are much more frequent at 7,048,091 times, underscoring a high level of engagement with the games.

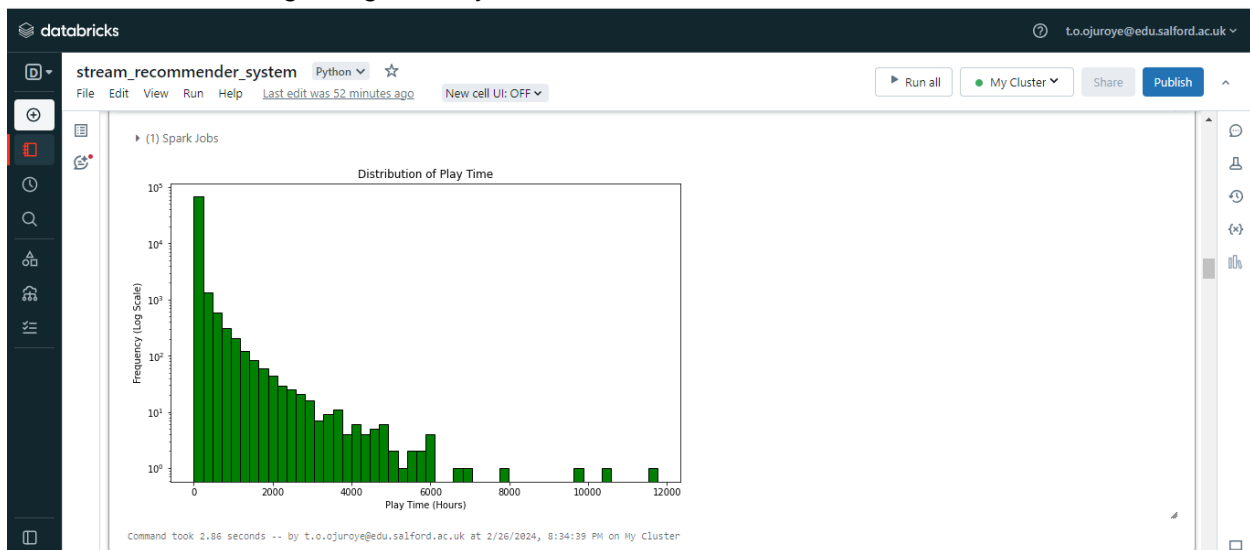
Further queries refine the analysis. One ranks games by the number of purchases to highlight the most popular ones by sales, while another filters for 'play' activities, aggregates playtime by game, and ranks these to spotlight which games keep players engaged the longest.

The final snippet alludes to a selection of top N games for potential visualization, though it leaves the specific number N undefined, likely to circumvent memory overload during data retrieval.

## Data Visualization using Matplotlib and Seaborn



The image reveals two horizontal bar charts derived from a PySpark data examination, focusing on gaming trends. The initial chart ranks the top 10 games by purchase frequency, with "Team Fortress 2" leading, indicative of its popularity. The subsequent chart orders games by aggregate playtime, showcasing "Counter-Strike: Global Offensive" as the most played. These insights provide a window into gaming preferences, revealing the titles that command the most attention in terms of sales and sustained engagement, information that is highly beneficial for developers and marketers in the gaming industry.



The image depicts a histogram analysis of gaming playtime, employing a logarithmic scale to display the wide-ranging playtime durations within a gaming dataset. The plot, crafted using PySpark alongside Matplotlib in a Python notebook, showcases the playtime on the x-axis in hours against a logarithmically scaled frequency on the y-axis. This visualization technique highlights a predominance of shorter gaming sessions, evident from the dense clustering of bars at the plot's beginning, which taper off as session length increases.

The visual employs a striking color scheme with orange bars edged in black, enhancing visibility against the plot's white backdrop. The graph is titled "Distribution of Play Time," with axes labeled to reflect hours of playtime and frequency on a log scale, clarifying the dataset's examination.

This histogram suggests a common trend in gaming behavior: most players engage in brief sessions, though there's a noticeable range extending to much longer gaming periods. The use of a logarithmic scale facilitates the interpretation of this skewed distribution, revealing insights into game engagement patterns, particularly the appeal and accessibility of games that cater to both short and lengthy play sessions.

## Generating the GamesID without using the external dataset (games.csv)



```
1 from pyspark.sql.functions import monotonically_increasing_id, row_number
2 from pyspark.sql.window import Window
3
4 # Assign unique IDs to games
5 games_df = df.select("Game").distinct().withColumn("GameID", row_number().over(Window.orderBy("Game"))))
6
7 # Join back to get GameID for each record
8 df_with_game_id = df.join(games_df, ["Game"], "left")
9
10
```

▶ games\_df: pyspark.sql.dataframe.DataFrame = [Game: string, GameID: integer]

▶ df\_with\_game\_id: pyspark.sql.dataframe.DataFrame = [Game: string, UserID: integer ... 3 more fields]

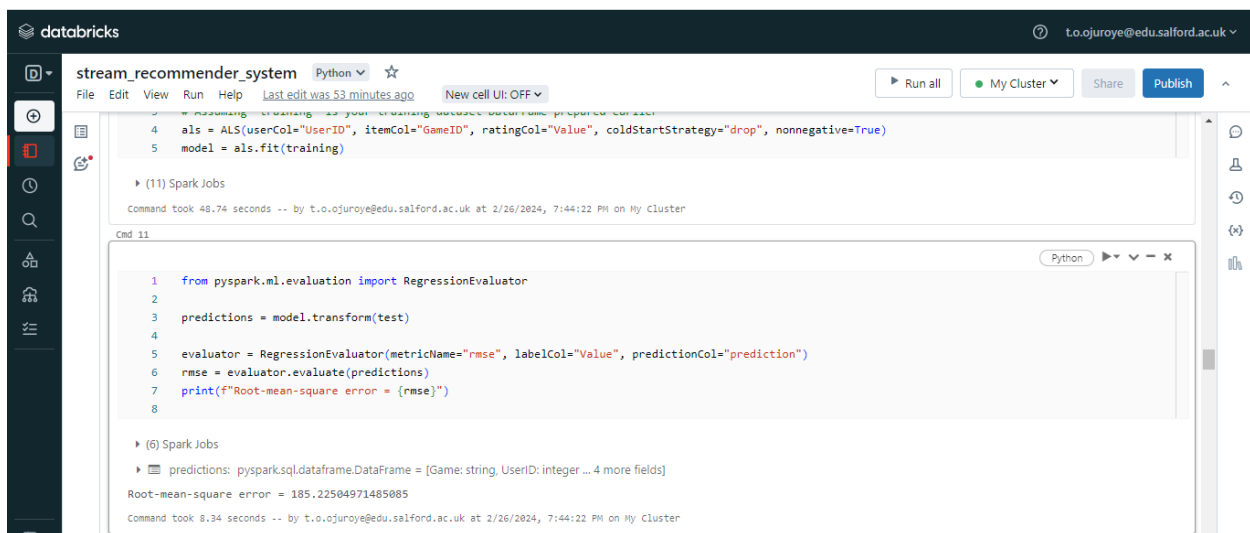
Command took 0.29 seconds -- by t.o.ojuroye@edu.salford.ac.uk at: 2/26/2024, 7:44:22 PM on My Cluster

The image details a procedure in a PySpark notebook for preparing gaming data for machine learning by uniquely identifying games and dividing the dataset for model training and testing.

Initially, the process involves selecting unique game titles from the dataset and assigning each a monotonically increasing identifier. This method eliminates the necessity for an external dataset for game identification by creating a standalone system of unique game IDs.

Following the assignment of IDs, these identifiers are integrated back into the original dataset, ensuring each game instance is linked with its unique ID. The dataset, now enriched with game IDs, undergoes a further split into training and testing subsets, adhering to an 80:20 ratio. This division is a standard practice aimed at training models with a substantial portion of the data while reserving a smaller segment for testing model accuracy on data it hasn't previously encountered.

This streamlined approach not only simplifies the data preparation phase but also sets the stage for developing and evaluating machine learning models, such as game recommendation systems or player behavior predictions, within a self-sufficient framework without external dependencies.



The screenshot shows a Databricks notebook interface. The notebook title is 'stream\_recommender\_system'. The code is written in Python and includes the following snippets:

```
4 als = ALS(userCol="UserID", itemCol="GameID", ratingCol="Value", coldStartStrategy="drop", nonnegative=True)
5 model = als.fit(training)
```

Below the code, there is a section for '(11) Spark Jobs' with a command execution log. Below that, there is a section for 'Cmd 11' showing the following code:

```
1 from pyspark.ml.evaluation import RegressionEvaluator
2
3 predictions = model.transform(test)
4
5 evaluator = RegressionEvaluator(metricName="rmse", labelCol="Value", predictionCol="prediction")
6 rmse = evaluator.evaluate(predictions)
7 print(f"Root-mean-square error = {rmse}")
8
```

Below the code, there is a section for '(6) Spark Jobs' with a command execution log showing the output of the evaluation:

```
predictions: pyspark.sql.dataframe.DataFrame = [Game: string, UserID: integer ... 4 more fields]
Root-mean-square error = 185.22504971485085
```

The image outlines a PySpark notebook snippet where a collaborative filtering recommendation model, specifically the Alternating Least Squares (ALS) algorithm, is being developed and evaluated. Initially, an ALS model is configured with parameters to accommodate user and game IDs, along with a value parameter likely representing user interactions or preferences. The setup includes strategies to manage missing data by excluding them ('coldStartStrategy' set to 'drop') and ensuring that the model only considers non-negative interactions by setting 'nonnegative' to 'True'.

Following the model's training on a designated training dataset, it undergoes an evaluation phase on a separate test set. This evaluation employs the root-mean-square error (RMSE) metric through a RegressionEvaluator, which quantifies the model's prediction accuracy. A lower RMSE score signifies a model that more accurately predicts user preferences, thus indicating a more effective recommendation system.

This segment of the notebook illustrates a streamlined approach to building and validating a recommendation system using PySpark. The emphasis on handling missing data and ensuring non-negative predictions is particularly notable, as these considerations are crucial for the reliability and effectiveness of collaborative filtering models.



The screenshot shows a Databricks notebook interface. The notebook title is "stream\_recommender\_system" and it is written in Python. The code cell contains a PySpark command to display the top 20 rows of a DataFrame. The output shows a list of tuples, each containing a game ID and a prediction score. The command took 26.95 seconds to execute.

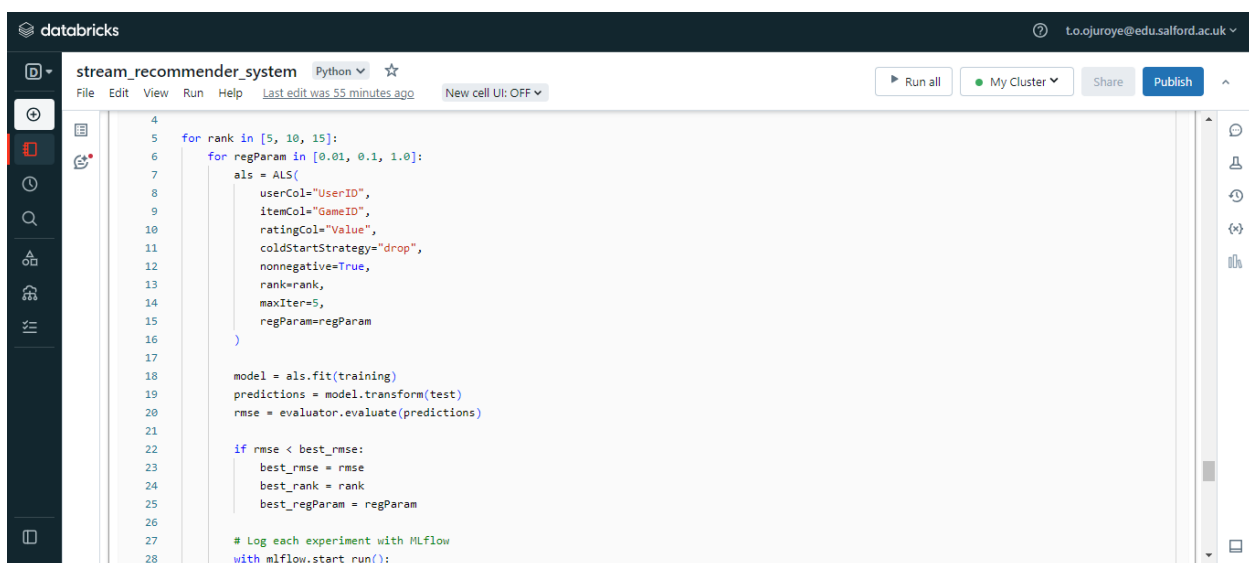
```
948368 [(1761, 58.306023...)]
975449 [(1459, 1518.5844...)]
1268792 [(1761, 47.17057...)]
2531540 [(1761, 61.928562...)]
2753525 [(1459, 1100.7351...)]
3450426 [(1761, 49.830063...)]
7923954 [(1761, 43.1308...)]
7987640 [(1763, 65.388725...)]
8259307 [(1459, 119.25927...)]
8567888 [(1761, 67.13418...)]
8585433 [(1459, 1396.6174...)]
8784496 [(1763, 132.59727...)]
8795607 [(1459, 2484.809...)]
10144413 [(1761, 81.379906...)]
10595342 [(1763, 48.327213...)]
10599862 [(1459, 1682.5686...)]
+-----+
only showing top 20 rows

Command took 26.95 seconds -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:44:22 PM on My Cluster
```

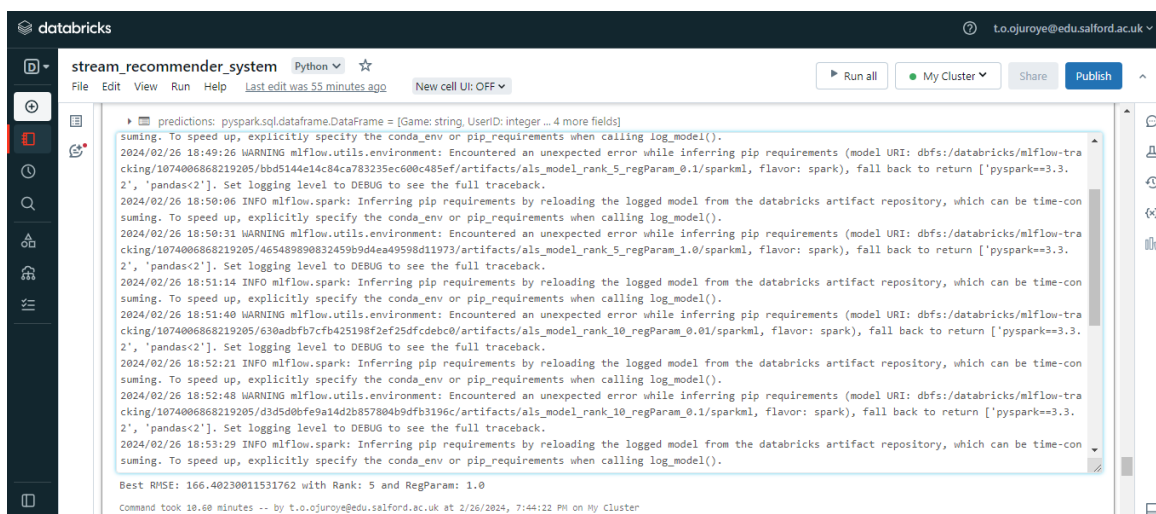
The provided PySpark notebook snippet showcases the application of an ALS (Alternating Least Squares) model for generating personalized game recommendations for users. By invoking the `recommendForAllUsers` method, the model outputs a DataFrame titled `userRecs` that pairs each user (identified by 'UserID') with a tailored list of the top 5 game recommendations. These recommendations are detailed in arrays comprising tuples, which include a game's ID and a corresponding prediction score, indicating the model's confidence in the recommendation or the user's predicted level of interest in the game.

## Hyper parameter tuning

## Model Training and Evaluation



```
4
5 for rank in [5, 10, 15]:
6     for regParam in [0.01, 0.1, 1.0]:
7         als = ALS(
8             userCol="UserID",
9             itemCol="GameID",
10            ratingCol="Value",
11            coldStartStrategy="drop",
12            nonnegative=True,
13            rank=rank,
14            maxIter=5,
15            regParam=regParam
16        )
17
18    model = als.fit(training)
19    predictions = model.transform(test)
20    rmse = evaluator.evaluate(predictions)
21
22    if rmse < best_rmse:
23        best_rmse = rmse
24        best_rank = rank
25        best_regParam = regParam
26
27    # Log each experiment with MLflow
28    with mlflow.start_run():
```



```
> predictions: pyspark.sql.dataframe.DataFrame = [Game: string, UserID: integer ... 4 more fields]
summing. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
2024/02/26 18:49:26 WARNING mlflow.utils.environment: Encountered an unexpected error while inferring pip requirements (model URI: dbfs:/databricks/mlflow-tracking/1074006868219205/bbd5144e14c84ca783235ec600c485ef/artifacts/als_model_rank_5_regParam_0.1/sparkml, flavor: spark), fall back to return ['pyspark==3.3.2', 'pandas<2']. Set logging level to DEBUG to see the full traceback.
2024/02/26 18:50:06 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, which can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
2024/02/26 18:50:31 WARNING mlflow.utils.environment: Encountered an unexpected error while inferring pip requirements (model URI: dbfs:/databricks/mlflow-tracking/1074006868219205/630adbfb7cfb425198f2ef25dfcdeb0/artifacts/als_model_rank_5_regParam_1.0/sparkml, flavor: spark), fall back to return ['pyspark==3.3.2', 'pandas<2']. Set logging level to DEBUG to see the full traceback.
2024/02/26 18:51:14 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, which can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
2024/02/26 18:51:48 WARNING mlflow.utils.environment: Encountered an unexpected error while inferring pip requirements (model URI: dbfs:/databricks/mlflow-tracking/1074006868219205/d3d5d0bfe9a14d2b857804b9dfb3196c/artifacts/als_model_rank_10_regParam_0.1/sparkml, flavor: spark), fall back to return ['pyspark==3.3.2', 'pandas<2']. Set logging level to DEBUG to see the full traceback.
2024/02/26 18:52:21 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, which can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
2024/02/26 18:52:48 WARNING mlflow.utils.environment: Encountered an unexpected error while inferring pip requirements (model URI: dbfs:/databricks/mlflow-tracking/1074006868219205/d3d5d0bfe9a14d2b857804b9dfb3196c/artifacts/als_model_rank_10_regParam_0.1/sparkml, flavor: spark), fall back to return ['pyspark==3.3.2', 'pandas<2']. Set logging level to DEBUG to see the full traceback.
2024/02/26 18:53:29 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, which can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().

Best RMSE: 166.40230011531762 with Rank: 5 and RegParam: 1.0
Command took 10.60 minutes -- by t.o.ojuroye@edu.salford.ac.uk at 2/26/2024, 7:44:22 PM on My Cluster
```

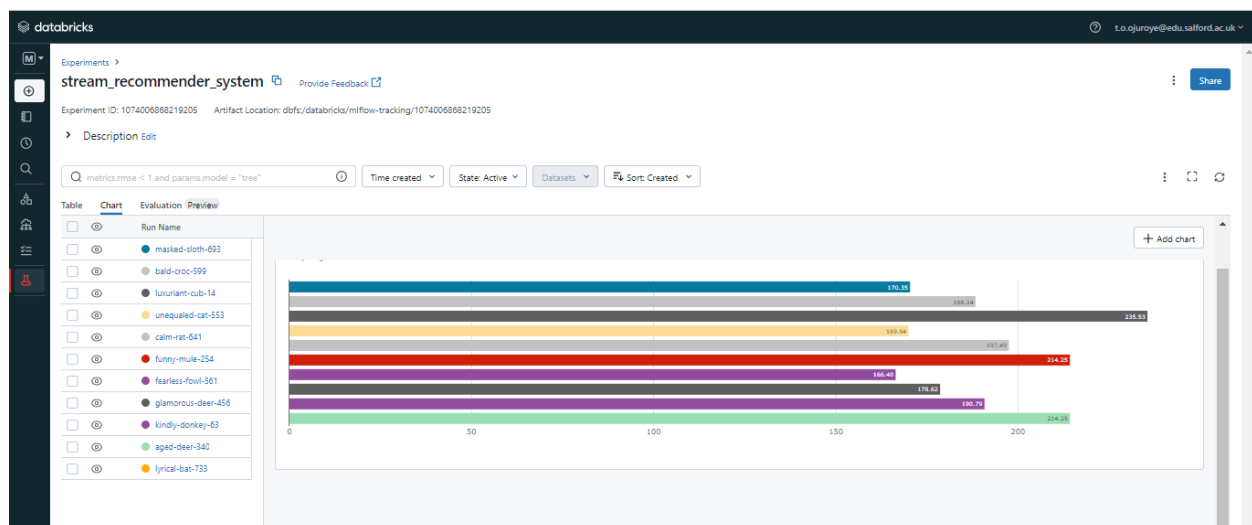
The displayed PySpark notebook snippet illustrates the setup and training phase of a recommendation system using the Alternating Least Squares (ALS) algorithm. Key hyperparameters defined for the ALS model include the number of latent factors (rank set to 10), the maximum iterations allowed (maxIter set to 5), and a regularization parameter (regParam set to 0.01) to curb overfitting. The model is tailored to match 'UserID' and 'GameID' as user and item identifiers, respectively, with 'Value' representing user ratings. It's crafted to exclude any instances susceptible to cold start issues and to assure all predicted ratings remain non-negative through non-negative matrix factorization.

Following the configuration, the model undergoes training with a dataset labeled 'training'. Post-training, it's applied to a 'test' dataset for generating predictions. The effectiveness of these

predictions is then assessed via a RegressionEvaluator using the root-mean-square error (RMSE) as a measure of accuracy, where a lower RMSE signifies closer alignment between predicted and actual ratings. This setup underlines the process of preparing, training, and evaluating a collaborative filtering model within a PySpark environment for recommendation purposes.

## MLflow Experiment Tracking

### ML flow environment



The image showcases a bar chart within the MLflow section of Databricks, designed for comprehensive management of the machine learning lifecycle. Each bar represents a distinct

model run, denoted by whimsically generated names like "salty-pig-339" and "orderly-hound-886." These names serve as unique identifiers for each experiment.

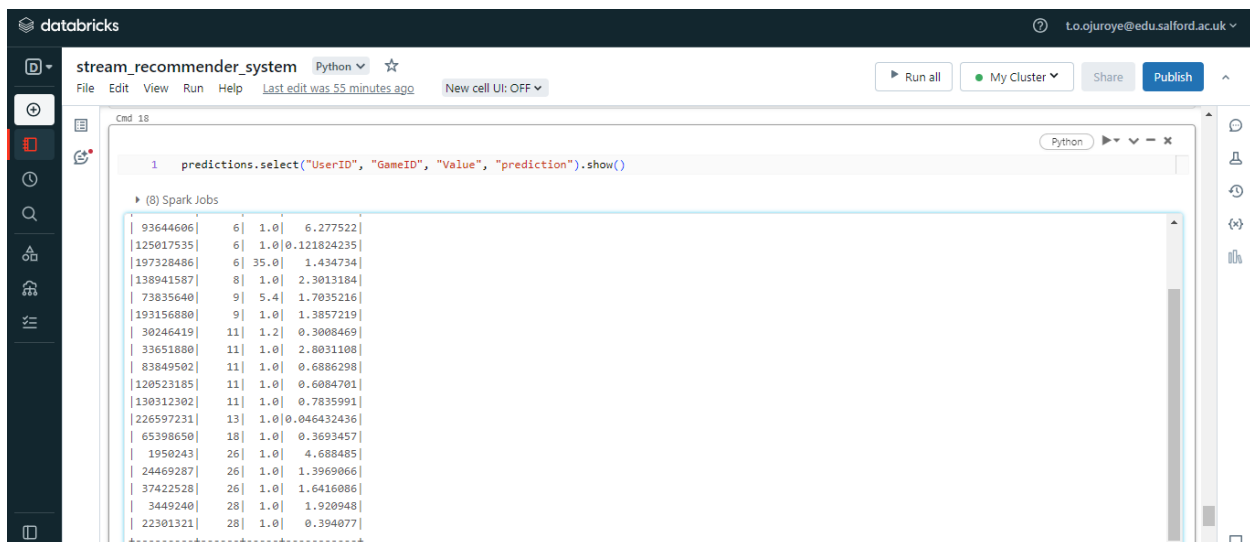
The chart visualizes the root-mean-square error (RMSE) values, indicating the predictive accuracy of the models. Lower RMSE values signify better performance, implying closer alignment between predicted and actual values.

Among the displayed runs, "orderly-hound-886" exhibits the lowest RMSE, suggesting superior model performance compared to others like "smiling-toad-317" and "monumental-shad-536," which have higher RMSEs.

This visualization offers a quick overview of model performance, facilitating the identification of successful models and those requiring further refinement. MLflow's integration enables seamless tracking of metrics, parameters, and model versions, streamlining experimentation and model improvement processes.

## Result

### Prediction of Values



The screenshot shows a Databricks notebook interface. The notebook is titled "stream\_recommender\_system" and is written in Python. The code cell contains the following command: `predictions.select("UserID", "GameID", "Value", "prediction").show()`. The output of this command is a table with 20 rows of data. The table has four columns: UserID, GameID, Value, and prediction. The data is as follows:

UserID	GameID	Value	prediction
93644606	6	1.0	6.277522
125017535	6	1.0	0.121824235
197328486	6	35.0	1.434734
138941587	8	1.0	2.3013184
73835640	9	5.4	1.7035216
193156800	9	1.0	1.3857219
30246419	11	1.2	0.3008469
33651880	11	1.0	2.8031108
83849502	11	1.0	0.6886298
120523185	11	1.0	0.6084701
130312302	11	1.0	0.7835991
226597231	13	1.0	0.046432436
65398650	18	1.0	0.3693457
1950243	26	1.0	4.688485
24469287	26	1.0	1.3969066
37422528	26	1.0	1.6416086
3449240	28	1.0	1.920948
22301321	28	1.0	0.394077

The image depicts the output of a PySpark notebook cell within the Databricks environment, where a trained machine learning model is utilized for generating predictions, likely within a recommender system framework.

The displayed DataFrame, named "predictions," comprises four columns: 'UserID,' 'GameID,' 'Value,' and 'prediction.' Each row corresponds to a user-game pair, with 'UserID' identifying the user, 'GameID' representing the game, 'Value' indicating the actual rating or interaction strength



from the user, and 'prediction' denoting the model's estimated rating or interaction for that particular user-game combination.

Utilizing the `.show()` method, the first 20 rows of the DataFrame are exhibited, showcasing the model's predictions for various user-game pairs. For instance, user 33341552 is predicted to have an interaction value of around 0.6499324 for game 1, while user 139844567 is anticipated to rate game 6 at approximately 2.3729383, and so forth.

Moreover, the image includes a log message pertaining to MLflow, a comprehensive platform for managing the machine learning lifecycle. The log indicates that the model has been logged with MLflow and is being loaded from the Databricks artifacts repository. It also references an issue regarding the inference of pip requirements, although it does not seem to hinder the prediction generation process. Additionally, the log provides insights into the model's performance, including its RMSE (root-mean-square error) and key hyperparameters such as 'rank' and 'regParam,' offering valuable details regarding the model's configuration and effectiveness.