Coursework

Big Data Tools and Technology

MSc Data Science

University of Salford MANCHESTER

ASSIGNMENT

ON

BIG DATA TOOLS AND TECHNIQUES

ONYEAJUNWANNE JUDE UFOH

@00324993

APRIL, 2023

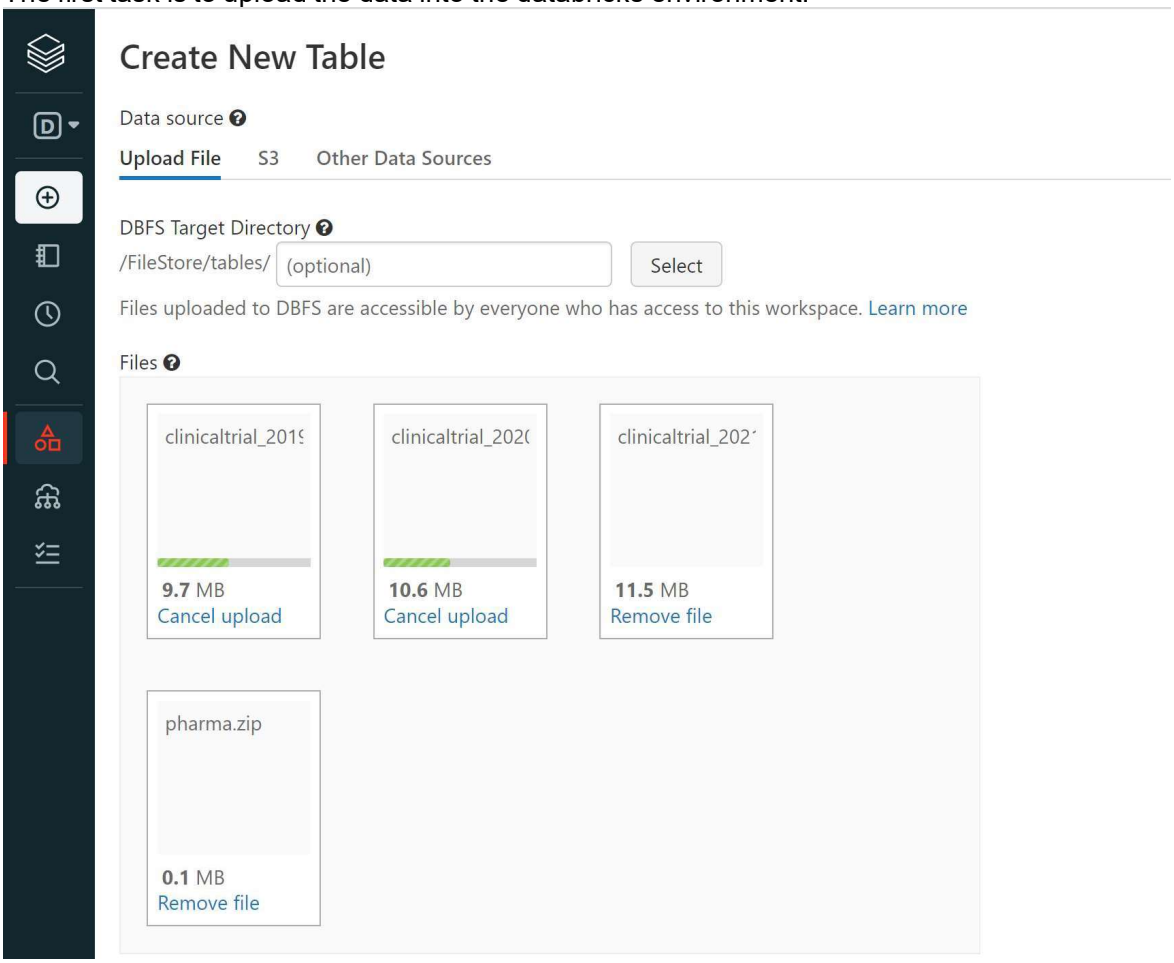
TASK 1

INTRODUCTION

The assignment requires the analysis of four datasets relating to clinicaltrials using pyspark and spark SQL in order to gain insights into the data. The analysis is done in databricks community edition environment.

SETUP

The first task is to upload the data into the databricks environment.



the files are saved in the /FileStore/tables/ directory in Databricks File System (DBFS).

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

```
FileInfo(path='dbfs:/FileStore/tables/clinicaltrial_2019.csv', name='clinicaltrial_2019.csv', size=42400056, modificationTime=1682166196000),
FileInfo(path='dbfs:/FileStore/tables/clinicaltrial_2019.zip', name='clinicaltrial_2019.zip', size=9707871, modificationTime=1679338032000),
FileInfo(path='dbfs:/FileStore/tables/clinicaltrial_2020/', name='clinicaltrial_2020/', size=0, modificationTime=0),
FileInfo(path='dbfs:/FileStore/tables/clinicaltrial_2020.csv', name='clinicaltrial_2020.csv', size=46318151, modificationTime=1682166077000),
FileInfo(path='dbfs:/FileStore/tables/clinicaltrial_2020.zip', name='clinicaltrial_2020.zip', size=10599182, modificationTime=1679338037000),
FileInfo(path='dbfs:/FileStore/tables/clinicaltrial_2021/', name='clinicaltrial_2021/', size=0, modificationTime=0),
FileInfo(path='dbfs:/FileStore/tables/clinicaltrial_2021-1.zip', name='clinicaltrial_2021-1.zip', size=11508457, modificationTime=1680861265000),
FileInfo(path='dbfs:/FileStore/tables/clinicaltrial_2021.csv', name='clinicaltrial_2021.csv', size=50359696, modificationTime=1682166008000),
FileInfo(path='dbfs:/FileStore/tables/clinicaltrial_2021.zip', name='clinicaltrial_2021.zip', size=11508457, modificationTime=1679338098000),
```

```
FileInfo(path='dbfs:/FileStore/tables/pharma.zip', name='pharma.zip', size=109982, modificationTime=1679338037000),
```

Take make room for reusable codes that can be used to analyse any year, we declare a year variable, concatenate it to 'clinicaltrial_' to construct the names of the zip files in the (DBFS). The file is assigned to a variable named 'basefile'

```
year = "2021"

basefile = "clinicaltrial_" + year
```

We import the OS library and use the OS.environ dictionary to set 'basefile' as the environment variable and assign it to the previously constructed variable 'basefile'.

```
import os

os.environ['basefile'] = basefile
```

```
Command took 0.09 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/22/2023, 1:23:09 PM on Assignment2
```

This is to ensure that the basefile variable is available to any child process that is created with the current interface. With the above configuration, any file in the form of 'clinicaltrial_+ year' will be assigned as the basefile in this environment.

Unzipping the files

We need to unzip the zipped folders in order to extract the csv files. Because the dbutils tookit doesn't provide unzip command, we copy the zipped folder from **'/FileStore/tables/**' using **dbutils.fs.cp()** into a temporary folder **'file:/tmp/'**, unzip it and copy the extracted .csv file into a new directory in '/**FileStore/tables**/. This is done for both clinicaltrial and pharma data.

```
dbutils.fs.cp("/FileStore/tables/" + basefile + ".zip", "file:/tmp/")

dbutils.fs.cp("/FileStore/tables/pharma.zip", "file:/tmp/")
```

```
Out[9]: True
```

```
Command took 1.17 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:31:59 AM on
```

```
%sh
# unzipping the basefile into a temporary folder
unzip -d /tmp /tmp/$basefile.zip

# unzipping the pharma.zip
unzip -d /tmp /tmp/pharma.zip
```

```
Archive:  /tmp/clinicaltrial_2021.zip
  inflating: /tmp/clinicaltrial_2021.csv
Archive:  /tmp/pharma.zip
  inflating: /tmp/pharma.csv
```

```
Command took 0.53 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:33:25 AM on
```

```
#creating directory for the basefile in the Filestore/tables
dbutils.fs.mkdirs("/FileStore/tables/" + basefile)
#moving the file from the temporary folder to DBFS
dbutils.fs.mv("file:/tmp/" +basefile + ".csv" , "/FileStore/tables/" +basefile + ".csv" , True)

#creating the directory for the pharma unzipped file
dbutils.fs.mkdirs('FileStore/tables/pharma')
#moving the pharma.csv from the temporary folder to DBFS
dbutils.fs.mv('File:/tmp/pharma.csv', '/FileStore/tables/pharma', True)
```

```
Out[14]: True
```

```
Command took 2.48 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:33:30 AM on My Cluster
```

Reading the CSV files

For RDD,
The clinicaltrial file is read into an RDD variable called 'clinicaltrial_RDD' while the pharma is read into pharma_RDD.

```
#reading the file into RDD
clinicaltrial_RDD=sc.textFile('/FileStore/tables/' +basefile + '.csv')
#reading the pharma file into RDD
pharma_RDD = sc.textFile('/FileStore/tables/pharma')
```

```
Command took 0.18 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:33:33 AM on My Cluster
```

For Dataframe,
The csv files are read into a dataframe variable with the names 'clinicaltrial_df' and pharma_df for clinicaltrial and pharma data respectively.

```
clinicaltrial_df = spark.read.options(delimiter ="|").csv("/FileStore/tables/" + basefile +  ".csv/", header= "True", inferSchema=True)
pharma_df = spark.read.options(delimiter =",").csv("/FileStore/tables/pharma.csv", header= "True", inferSchema=True)
```

```
▸ (4) Spark Jobs
▸ clinicaltrial_df: pyspark.sql.dataframe.DataFrame = [Id: string, Sponsor: string ... 7 more fields]
▸ pharma_df: pyspark.sql.dataframe.DataFrame = [Company: string, Parent_Company: string ... 32 more fields]
Command took 12.68 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:53:03 AM on My Cluster
```

The **'options()**' method is used to set the delimiter to a pipe (|) character for clinicaltrial and to comma (',') for pharma data. The **'header = "True"'** is used to specify that the first row contains column names. **'inferSchema = True**' is used to specify that Spark should attempt to infer the schema of the Dataframe from the contents of the csv file.

For Spark SQL
We first created and registered a User Defined Function called 'year_variable' having an argument 'year' and returns the 'year' as result.

```python
%python
def year_variable(year):
    # perform complex calculation
    result = year
    return result
```

```
Command took 0.09 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/25/2023, 10:27:42 AM on My Cluster
```

```
Cmd 3
```

```python
%python
from pyspark.sql.functions import udf

spark.udf.register("year_variable", year_variable)
```

```
Out[120]: <function __main__.year_variable(year)>
```

```
Command took 0.20 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/25/2023, 10:27:42 AM on My Cluster
```

We created a view named **'year'** as **year_value** by selecting the created UDF **year_variable and** choosing the desired year of study such as 2019, 2020 or 2021

```
--Creating a temporary view by selecting the function
CREATE OR REPLACE TEMPORARY VIEW year AS
SELECT year_variable('2021') AS year_value
```

With **spark.sql()** and **collect()** method, we retrieve the **year_value** as a list of rows from the temporary view, year. Since the view has only one row, we use **[0][0]** to retrieve the first value of the first row. We the then concatenate the **year_value** in the python code and '**clinicaltrial_**' to construct the **basefile** string as before

```python
%python
# Retrieve year_value from the temporary view
year_value = spark.sql("SELECT * FROM year").collect()[0][0]

# Use year_value in your Python code
basefile = "clinicaltrial_" + year_value
```

```
▸ (1) Spark Jobs
Command took 0.29 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/25/2023, 10:27:42 AM on My Cluster
```

Next, we read the basefile and the pharma data into a dataframe variable named 'clinicaltrial' and pharma just like in dataframe above

```python
%python
clinicaltrial = spark.read.options(delimiter ="|").csv("/FileStore/tables/" + basefile +  ".csv/", header= "True", inferSchema=True)

pharma = spark.read.csv('/FileStore/tables/pharma', header=True, sep=',', inferSchema = True)
```

```
▸ (4) Spark Jobs
▸ clinicaltrial:  pyspark.sql.dataframe.DataFrame = [Id: string, Sponsor: string ... 7 more fields]
▸ pharma:  pyspark.sql.dataframe.DataFrame = [Company: string, Parent_Company: string ... 32 more fields]
Command took 4.24 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/25/2023, 10:27:42 AM on My Cluster
```

we create a temporary view of both data in SQl with the code below.

```
%python
#creating temporary views of the data
clinicaltrial.createOrReplaceTempView('clinicaltrial')

pharma.createOrReplaceTempView('pharma')
```

```
Command took 0.10 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/25/2023, 10:27:42 AM on My Cluster
```

DATA EXPLORATION

An exploration of both data shows that the the clinicaltrial_2021 has 387261 rows and 9 columns while the pharma data has 968 rows and 34 columns.

```
#count of number of columns in the dataset
first_row1 = clinicaltrial_RDD.first()

print('number of columns in clinicaltrial data:',len(first_row1.split("|")))
#count of number of rows (minus 1 is used to exclude the header column)
num_of_rows = clinicaltrial_RDD.count()-1
print('number of rows: ',num_of_rows)
```

```
▸ (2) Spark Jobs
number of columns in clinicaltrial data: 9
number of rows:  387261
Command took 3.00 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:46:23 AM on My Cluster
```

```
#count of number of columns in the dataset
first_row = pharma_RDD.first()

print('number of columns in pharma data:',len(first_row.split(",")))
#count of number of rows (minus 1 is used to exclude the header column)
num_of_rows = pharma_RDD.count()-1
print('number of rows: ',num_of_rows)
```

```
▸ (2) Spark Jobs
number of columns in clinicaltrial data: 34
number of rows:  968
Command took 0.60 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:46:24 AM on My Cluster
```

The exploration also revealed there is a missing value at the ID of 'NCT04381455'

```
#checking for missing values in the row
rows_with_missing_values = clinicaltrial_RDD_NH.filter(lambda row: "null" in row or "NaN" in row)
#show the row with missing value
rows_with_missing_values.collect()
```

```
▸ (1) Spark Jobs
Out[25]: ['NCT04381455|Studio Odontoiatrico Associato Dr. P. Cicchese e L. Canullo|Active, not recruiting|May 2020|Jun 2022|Observational|May 2020||']
Command took 2.11 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:46:24 AM on My Cluster
```

DATA PREPROCESSING

For RDD

We removed the header row on both datasets and assign them to '**clinicaltrial_RDD_NH**' and '**pharma_RDD_NH'** respectively.

```
#removing the header row for clinicaltrial data.
header = clinicaltrial_RDD.first()
clinicaltrial_RDD_NH =clinicaltrial_RDD.filter(lambda row: row !=header)

#removing the first row for pharma data
header2 = pharma_RDD.first()
pharma_RDD_NH = pharma_RDD.filter(lambda row: row != header2)
```

▸ (2) Spark Jobs

Command took 1.39 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 11:34:07 AM on My Cluster

For Dataframe:

Dataframes recognize named columns and assume the first rows as the header row which in this case is true and so, we do not need to perform any operation to remove the header row. The dataframes for clinicaltrial and pharma dataset are respective **clinicaltrial_df** and **pharma_df**

For Spark SQL

We create tables in the databricks default database from the already created temporary views for both datasets.

```
create or replace table default.clinicaltrial as select * from clinicaltrial
```

▸ (8) Spark Jobs

Query returned no results

Command took 13.40 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/25/2023, 10:27:42 AM on My Cluster

Cmd 13

```
create or replace table default.pharma as select * from pharma
```

▸ (8) Spark Jobs

Query returned no results

Command took 5.18 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/25/2023, 10:27:42 AM on My Cluster

Solutions to Questions

Question 1: The number of distinct studies in the dataset

Since the Id column contains unique data for each row, the number of distinct studies can be found from the count of the IDs.

RDD implementation

we split the RDD at the delimiter, select the ID column which is the first column [0] and count the column using count() and distinct () functions

```
#split the RDD at delimiter
split_rdd = clinicaltrial_RDD_NH.map(lambda line: line.split("|"))
#select the first column which is the ID column
ID_column_rdd = split_rdd.map(lambda x: x[0])
#count the distinct values
column_count = ID_column_rdd.distinct().count()
#print out the column count
print('number of distinct studies:', column_count)
```

▸ (1) Spark Jobs

number of distinct studies: 387261

Command took 4.26 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 11:59:48 AM on My Cluster

Dataframe implementation

Here, we also select the 'Id' column using the select function and apply the distinct and count

functions to them

```
num_studies = clinicaltrial_df.select("id").distinct().count()

print("Number of distinct studies:", num_studies)
```

▸ (3) Spark Jobs

```
Number of distinct studies: 387261
```

Command took 4.84 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 12:10:37 PM on My Cluster

Spark SQL implementation

We use the aggregate function 'count' and the distinct keyword to count the distinct IDs

```
------count of all the studies using ID column
select 'Total number of studies: ', count('id') as Count_of_Studies from default.clinicaltrial
```

▸ (6) Spark Jobs

Table +

	Total number of studies:	Count_of_Studies
1	Total number of studies:	387261

1 row | 2.99 seconds runtime

Command took 2.99 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 12:38:19 PM on My Cluster

Question 2:Listing all the types in the type column along with their frequencies

Here, we assume that the type column contains the different study types. We then count and group the different types.

RDD implemetation

We use the map() function to create a key value pair for the values in the sixth column with the value of 1 as the count. The **reduceByKey()** function is used to sum up the counts for each key (i.e type). The **sortBy()** is then used to sort the result in descending order. We iterate through the resulting counts_by_type RDD using the collect() function and print out the type and count(frequency) of each value.

```
counts_by_type  = clinicaltrial_RDD_NH.map(lambda s:(s.split("|")[5], 1)).reduceByKey(lambda a,b: a + b).sortBy(lambda x: x[1],
ascending=False)

#print the type counts and the corresponding numbers
for type_count in counts_by_type.collect():
    print(type_count[0],'=', type_count[1])
```

▸ (3) Spark Jobs

```
Interventional = 301472
Observational = 77540
Observational [Patient Registry] = 8180
Expanded Access = 69
```

Command took 3.18 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 11:34:07 AM on My Cluster

Dataframe implementation

We import all the necessary functions from pyspark.sql.functions. The groupBy function is used to group the dataframe according to the type column. Count() function then counts the number of occurences of each value. Finally, sort() and desc() functions are combined as sort(desc("count")) to sort the result in descending order based on the count. Show() is used to print out the result

```
from pyspark.sql.functions import desc, col

type_counts = clinicaltrial_df.groupBy("Type").count().sort(desc("count"))

type_counts.show()
```

(2) Spark Jobs

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

```
+--------------------+------+
|                Type| count|
+--------------------+------+
|      Interventional|301472|
|       Observational| 77540|
|Observational [Pa...|  8180|
|     Expanded Access|    69|
+--------------------+------+
```

Command took 6.64 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:53:03

SPARK SQL implementation

We use the aggragate function 'count' to count all the records from default.clinicaltrial, group them by "Type", and order them by count in descending order.

```
SELECT Type, COUNT(*) as count
FROM default.clinicaltrial
GROUP BY Type
ORDER BY count DESC
```

(2) Spark Jobs

Table +

	Type	count
1	Interventional	301472
2	Observational	77540
3	Observational [Patient Registry]	8180
4	Expanded Access	69

4 rows | 3.03 seconds runtime

Command took 3.03 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 12:38:19 PM on My Cluster

Discussion of Result: The sum of the types as seen from the analysis is **387261** which tallies with the result obtained in question 1 for the total number of studies. This shows the correctness of the result.

Quest 3: The top 5 conditions with their frequencies.

Column 7 of the clinicaltrial contains the conditions. Some rows contain more than one condition and so, such rows need to be exploded in order to capture the different conditions.

RDD implementation

We first create an RDD called condition_rdd by splitting the dataset by '|' and selecting the condition column and using flatMap() to explode the column.

We then count the frequency of each condition by creating a tuple of the condition and the value 1 for each record in the RDD and use reduceByKey to sum up the values for each condition. The result is sorted using sortBy() function and the empty strings are filtered out of it. The resulting RDD which contains tuples in the form of condition and count is assigned to conditions_count.

We use the take() function to take out the top 5 conditions, loop over them and print out their values.

```
# Split on '|', select the Conditions column and explode using flatmap
condition_rdd = clinicaltrial_RDD_NH.map(lambda x: x.split('|')[7]) \
                        .flatMap(lambda x: x.split(','))

# Count the frequency of each condition
conditions_count = condition_rdd.map(lambda x: (x.strip(), 1)) \
                                .reduceByKey(lambda x, y: x + y) \
                                .sortBy(lambda x: x[1], ascending=False) \
                                .filter(lambda x: x[0].strip())
#get the top 5 conditions
top5_condition = conditions_count.take(5)
#display the result

for condition_count  in top5_condition:

   # print(condition_count[0][0], '=', condition_count[1])

     print("{:<25}{}".format(condition_count[0], condition_count[1]))
```

▸ (3) Spark Jobs

```
Carcinoma                13389
Diabetes Mellitus        11080
Neoplasms                9371
Breast Neoplasms         8640
Syndrome                 8032
```

Command took 3.53 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 11:34:07 AM on My Cluster

Dataframe implementation

The explode function is applied to the condition column of the dataframe to create a new dataframe called splitted_ConditionsDF in which the conditions column is splitted by commas and stored in a new column called splitted_conditions.

The splitted_condition column is selected and the explode function is applied to it to create a new dataframe called exploded_ConditionDF having a new column named Exploded_Conditions.

The exploded_ConditionDF is counted with count() function, sorted in a descending order with the sort() ad desc() functions and grouped using groupBy function according to the values contained in the Exploded_Conditions column. Limit(5) is used to take the top 5 values and the result is displayed as a panda dataframe

```
#importing the necessary libraries
from pyspark.sql.functions import split, explode

 #Applying the split function on the condition column to create a new column called splitted_conditins
Splitted_conditionDF = clinicaltrial_df.select(split("Conditions", ",").alias("Splitted_Conditions"))

#applying the explode function on the splitted_coditionDF to create Exploded_conditionDf
Exploded_conditionDF = Splitted_conditionDF.select(explode("Splitted_Conditions").alias('Exploded_conditions'))

#counting the exploded condition column and grouping it according to the the values
conditions_count = Exploded_conditionDF.groupBy('Exploded_conditions').count().sort(desc('count')).limit(5)

#converting the resulting dataframe to pandas
conditions_count.toPandas()
```

▸ (2) Spark Jobs

▸ Splitted_conditionDF: pyspark.sql.dataframe.DataFrame = [Splitted_Conditions: array]

▸ Exploded_conditionDF: pyspark.sql.dataframe.DataFrame = [Exploded_conditions: string]

▸ conditions_count: pyspark.sql.dataframe.DataFrame = [Exploded_conditions: string, count: long]

	Exploded_conditions	count
0	Carcinoma	13389
1	Diabetes Mellitus	11080
2	Neoplasms	9371
3	Breast Neoplasms	8640
4	Syndrome	8032

Command took 4.66 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 3:15:23 PM on My Cluster

Spark SQL implementation

We first used a subquery to split the conditions table into separate conditions using the split function and explode the resulting array of conditions into separate rows using the explode function.

The subquery results to a temporary table where each row represents a single condition. We then use another query to select the condition column from the subquery table and apply the count function to count the number of occurrences of each condition in the table, grouping them using the

group by clause and ordering them using the **order by** clause in descending order using **desc** keyword.
Limit 5 is used to pick the top 5

```
SELECT condition, COUNT(*) AS Frequency
FROM (
    SELECT explode(split(Conditions, ',')) AS condition
    FROM default.clinicaltrial
    WHERE Conditions IS NOT NULL
)
GROUP BY condition
ORDER BY Frequency DESC
LIMIT 5;
```

▸ (2) Spark Jobs

Table +

	condition	Frequency
1	Carcinoma	13389
2	Diabetes Mellitus	11080
3	Neoplasms	9371
4	Breast Neoplasms	8640
5	Syndrome	8032

5 rows | 3.37 seconds runtime

Command took 3.37 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 12:38:

Question4: Ten most common sponsors that are not pharmaceutical companies.

The second column of the pharma dataset contains all the pharmaceutical companies and the second column of the clinicaltrial dataset contains all the sponsors. We can assume that the names that are in sponsors column but not in the list of pharmaceutical companies are the non pharmaceutical sponsors which can then be obtained by applying filtering technique.

RDD Implementation
First we remove the double quotes in the pharma company data, split the pharma_RDD_NH by comma delimiter and select the distinct values in the second column (Parent company) and assign the result to **pharma_RDD_NH2**. This represents the unique names of all the pharmaceutical companies. Secondly, we select only the sponsors column of the clinicaltrial_RDD_NH and assign it to **clinicaltrial_RDD_NH2**. We then create non_pharma_RDD which contains only non - pharmaceutical sponspors by filtering out pharma_RDD_NH2 from clinicaltrial_RDD_NH2 using the **subtract()** function.
Finally, we create a count_of_non_pharma RDD by applying map transformation to non_pharma_RDD to create key-value pairs where each key is non_pharma sponsor and the value is set to 1.the reduceByKey transformation groups the key-value pairs by key and adds up the values of each key to produce a count of non-pharma sponsors. The sortBy transformation sorts the key-value pairs in descending order based on the value of each pair. We apply the take() function to count_of_non_pharma to display the first 10 value pair

```
#removing the double quotation in RDD, splitting at comma delimiter and selecting the distinct values in the parent company column
pharma_RDD_NH2 =pharma_RDD_NH.map(lambda x: x.replace('"', '')).map(lambda s:(s.split(",")[1])).distinct()

#selecting only the sponsor column of clinicaltrial_RDD_NH
clinicaltrial_RDD_NH2 = clinicaltrial_RDD_NH.map(lambda x: x.split('|')[1])

#filtering out of non_pharma data by selecting only sponsors that dont appear in pharma dataset
non_pharma_RDD = clinicaltrial_RDD_NH2.subtract(pharma_RDD_NH2)

#counting, grouping and sorting the distinct non_pharma sponsors in descending order according to count
counts_of_non_pharma  = non_pharma_RDD.map(lambda x: (x, 1)).reduceByKey(lambda a, b: a + b).sortBy(lambda x: x[1], ascending=False)

#displaying the top 10
counts_of_non_pharma.take(10)
```

▸ (3) Spark Jobs

```
Out[55]: [('National Cancer Institute (NCI)', 3218),
 ('M.D. Anderson Cancer Center', 2414),
 ('Assistance Publique - Hôpitaux de Paris', 2369),
 ('Mayo Clinic', 2300),
 ('Merck Sharp & Dohme Corp.', 2243),
 ('Assiut University', 2154),
 ('Novartis Pharmaceuticals', 2088),
 ('Massachusetts General Hospital', 1971),
 ('Cairo University', 1928),
 ('Hoffmann-La Roche', 1828)]
```

Command took 5.83 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 11:34:07 AM on My Cluster

Dataframe implementation

First, we create a dataframe, parent_company_df by selecting only the Parent_Company column of pharma_df.

We create a list of pharma companies out of the parent_company_df by using the row(), distinct() and collect() functions and name it pharma_companies

Next, out of the pharma company list, we create a dataframe called non_pharma_df by filtering the clinicaltrial_df to include only rows where value in the sponsor column is not in the pharma_companies list. The **'~'** symbol has been used to negate **isin()** method such that only rows where the 'sponsor' column value is not in, are selected.

Finally, we count the number of rows using count(), group them according to the distinct sponsors using groupBy() method and sort them according to the value of the count in a descending order.

The top ten sponsors are selected using the limit method and the result is assigned to top_sponsors dataFrame and displayed using the show method.

```python
# Select the Parent Company column from the dataset
parent_company_df = pharma_df.select("Parent_Company")

# Create a list of pharmaceutical companies by collecting unique values from the Parent Company column
pharma_companies = [row['Parent_Company'] for row in parent_company_df.distinct().collect()]

# Filter the dataset to include only non-pharmaceutical companies
non_pharma_df = clinicaltrial_df.filter(~clinicaltrial_df['Sponsor'].isin(pharma_companies))

# Group by sponsor and count the number of clinical trials each sponsor has sponsored
sponsor_counts = non_pharma_df.groupBy('Sponsor').count().orderBy('count', ascending=False)

# Select the top 10 sponsors with the highest number of sponsored clinical trials
top_sponsors = sponsor_counts.limit(10)

top_sponsors.show()
```

▸ (4) Spark Jobs

▸ parent_company_df: pyspark.sql.dataframe.DataFrame = [Parent_Company: string]
▸ non_pharma_df: pyspark.sql.dataframe.DataFrame = [Id: string, Sponsor: string ... 7 more fields]
▸ sponsor_counts: pyspark.sql.dataframe.DataFrame = [Sponsor: string, count: long]
▸ top_sponsors: pyspark.sql.dataframe.DataFrame = [Sponsor: string, count: long]

```
+--------------------+-----+
|             Sponsor|count|
+--------------------+-----+
|National Cancer I...| 3218|
|M.D. Anderson Can...| 2414|
|Assistance Publiq...| 2369|
|         Mayo Clinic| 2300|
|Merck Sharp & Doh...| 2243|
|   Assiut University| 2154|
|Novartis Pharmace...| 2088|
|Massachusetts Gen...| 1971|
|    Cairo University| 1928|
|   Hoffmann-La Roche| 1828|
+--------------------+-----+
```

Command took 7.52 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:53:03 AM on My Cluster

Spark SQL Implementatation

We select the spnsor, count of all the sponsor from the clinicaltrial table aliased as **a** and then left join with the pharma data aliased **b**. The join is done on the sponsors column of clinicaltrial and the parent_company column of pharma data. Because it is **left join**, everything under sponsor in the clinical data will be selected while only the items under parent_company in pharma data that are also in clinicaltrial will be selected in pharma. The rest places are set to null. We then use the **where clause** to select when parent company is null and sponsor not null. Group them by sponsor and order them by the count_of_sponsors in descending order. 'Limit 10' is used to pick the top 10.

```sql
SELECT a.Sponsor, COUNT(*) AS Count_of_Sponsor
FROM default.clinicaltrial a
LEFT JOIN default.pharma b ON a.Sponsor = b.Parent_Company
WHERE b.Parent_Company IS NULL AND a.Sponsor IS NOT NULL
GROUP BY a.Sponsor
ORDER BY Count_of_Sponsor DESC
LIMIT 10
```

▸ (3) Spark Jobs

Table +

	Sponsor	Count_of_Sponsor
1	National Cancer Institute (NCI)	3218
2	M.D. Anderson Cancer Center	2414
3	Assistance Publique - Hôpitaux de Paris	2369
4	Mayo Clinic	2300
5	Merck Sharp & Dohme Corp.	2243
6	Assiut University	2154
7	Novartis Pharmaceuticals	2088

10 rows | 3.52 seconds runtime

Command took 3.52 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 12:38:19 PM on My Cluster

Question 5: Visualisation of completed studies each month in a given year

The count of each month in a given year will give the number of completed studies that month.

RDD implementation

we first import the datetime function which is needed to create convert string to date format and the calender function that will be used to order the months. From the clinicaltrial_RDD_NH. We filter out the studies that have status column as 'completed' and in which the completed column ends with the year variable initially chosen.We assign the result to Completed_Studies_in_year.

We use the map() and the datetime.strptime functions to extract the month and year from completion date and store the month as key-value pair with the value with each value as 1. We then use the reduceByKey() function to sum the values for each key (month) and sort the result by the month key using the sortByKey() function. We use the map() function again to convert the month to its name using calender.month_name function.

```
#import the datetime function for converting to date and calender for sorting the months
from datetime import datetime
import calendar

#extract the studies with status 'completed' and in which completion has year = '2021'
Completed_Studies_in_year = clinicaltrial_RDD_NH.filter(lambda x: x.split("|")[2] == "Completed" and x.split("|")[4].endswith(year)

#extracting the month and year from the completion date and mapping them to (month, 1)
Completed_Studies_in_year = Completed_Studies_in_year.map(lambda x: (datetime.strptime(x.split("|")[4], '%b %Y').month,1))

#sort the result by month key
Completed_Studies_in_year = Completed_Studies_in_year.reduceByKey(lambda x,y: x + y).sortByKey()

#map the month number to its name
Completed_Studies_in_year = Completed_Studies_in_year.map(lambda x: (calendar.month_name[x[0]],x[1]))
```

▸ (2) Spark Jobs

Command took 2.83 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 11:34:07 AM on My Cluster

Cmd 25

```
Completed_Studies_in_year.take(12)
```

▸ (2) Spark Jobs

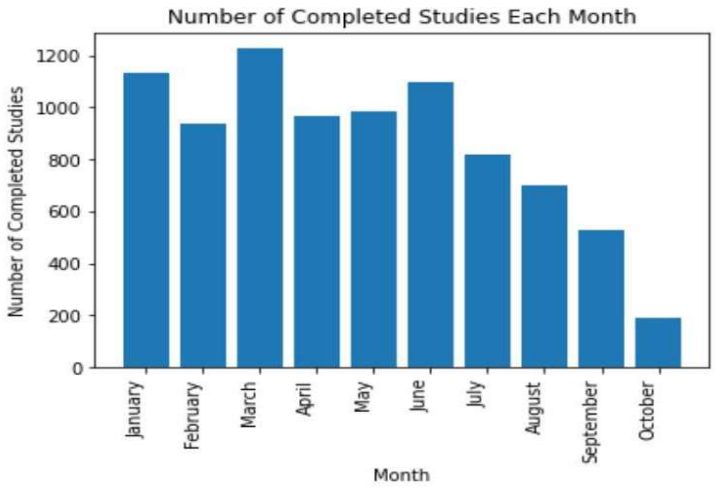
```
Out[57]: [('January', 1131),
 ('February', 934),
 ('March', 1227),
 ('April', 967),
 ('May', 984),
 ('June', 1094),
 ('July', 819),
 ('August', 700),
 ('September', 528),
 ('October', 187)]
```

Command took 0.29 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 11:34:07 AM on My Cluster

The code and a visualisation in matplotlib of the completed_studies_in_year is shown below

```
#creating a barchart visualisation with completed_studies
import matplotlib.pyplot as plt

# The data
dataset = Completed_Studies_in_year.take(12)

# Extract the month names and counts into separate lists
months = [item[0] for item in dataset]
counts = [item[1] for item in dataset]

# Create a bar chart
fig, ax = plt.subplots()
ax.bar(months, counts)

# Set the chart title and axis labels
ax.set_title('Number of Completed Studies Each Month')
ax.set_xlabel('Month')
ax.set_ylabel('Number of Completed Studies')

# Rotate the x-axis labels to make them easier to read
plt.xticks(rotation=90, ha='right')

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

▸ (2) Spark Jobs



Dataframe Implemetation

We first import col, to_date,date_format functions from the pyspark.sql.functions. we then select the status and completion columns and applied filter to select only rows where the status is ''completed' and completion column contains the year variable initially selected. We use the to_date function to convert the completion column to date type column and use the count() function to count the completion column and group them according to the month and also sort them according to time months.

We created a new column called months having only the month name and dropped the completion table which is having month and year name.

The final result was reordered to make the month come first and was converted toPandas(), displayed and plotted.

```
from pyspark.sql.functions import col, to_date, date_format
#selecting the status and completion columns, filter them based on status=completed and completion year containing the year being con
completed_studies = clinicaltrial_df.select('Status', 'Completion')\
    .filter(col('Status') == "Completed")\
    .filter(col('Completion').contains(year))

#converting the completion column to date format
completed_studies = completed_studies.withColumn("Completion", to_date(completed_studies["Completion"], "MMM yyyy"))

#counting the rows and grouping and sorting according to the completion month
completed_studies = completed_studies.groupBy(col('Completion')).count().sort('Completion')

#change the column name from complateion to Months and show only the months
completed_studies = completed_studies.withColumn("Months", date_format(completed_studies["Completion"], "MMM")).drop("Completion")

#make the month column come before count
completed_studies = completed_studies.select("Months", *[col for col in completed_studies.columns if col != "Months"])

completed_studies.toPandas()
```

▸ (4) Spark Jobs

▸ completed_studies: pyspark.sql.dataframe.DataFrame = [Months: string, count: long]

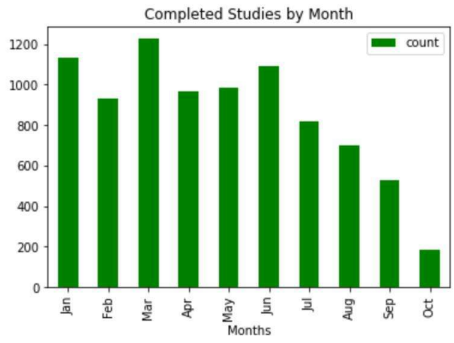
	Months	count
0	Jan	1131
1	Feb	934
2	Mar	1227
3	Apr	967
4	May	984
5	Jun	1094
6	Jul	819
7	Aug	700
8	Sep	528
9	Oct	187

Command took 5.47 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:53:03 AM on My Cluster

```
completed_studies.toPandas().plot(kind="bar", x="Months", y="count", title="Completed Studies by Month", color ='green')
```

▸ (4) Spark Jobs

```
Out[11]: <AxesSubplot:title={'center':'Completed Studies by Month'}, xlabel='Months'>
```



Command took 4.55 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 10:53:03 AM on My Cluster

Spark SQL Implementation

Here, we first created a subquery by selecting the completion dates from the clinicaltrial table, filtering only the completed studies and only for the years specified in the 'year' variable by using the **like** operator and concatenating '%' (select * from year), '%'. The subquery also converts the 'completion ' column to a date format using the cast and to_date function. The outer query groups the result by month using the date_format function.

The result is finally ordered by the month in chronological order using case statement.

```
 CREATE OR REPLACE TEMPORARY VIEW completed_studies_by_month AS
SELECT DATE_FORMAT(completion_date, "MMM") AS Month, COUNT(*) AS Number_of_Completed_Studies
FROM
  (SELECT id, completion, CAST(TO_DATE(completion, 'MMM yyyy') AS DATE) AS completion_date
   FROM default.clinicaltrial
   WHERE completion LIKE CONCAT('%',(select * from year), '%') AND Status ='Completed')
GROUP BY Month
order by case Month
when 'Jan' then 1
when 'Feb' then 2
when 'Mar' then 3
when 'Apr'  then 4
when 'May' then 5
when 'Jun' then 6
when 'Jul' then 7
when 'Aug' then 8
when 'Sep' then 9
when 'Oct'  then 10
when 'Nov' then 11
when 'Dec' then 12
end
```

OK

Command took 0.84 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 5:15:36 PM on My Cluster

To view the result, we select from completed_studies_by_month

```
select * from completed_studies_by_month
```

▸ (4) Spark Jobs

Table +

	Month	Number_of_Completed_Studies
1	Jan	1131
2	Feb	934
3	Mar	1227
4	Apr	967
5	May	984
6	Jun	1094
7	Jul	819

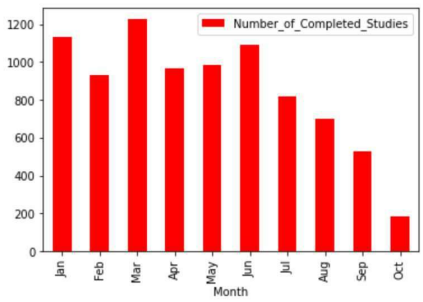
10 rows | 6.00 seconds runtime

Command took 6.00 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 5:15:39 PM on My Cluster

We can further convert the view to a dataframe and then to pandas and plot using matplotlib

```
#converting the dataframe to a panda and ploting a bar chart using matplotlib
import pandas as pd

import matplotlib.pyplot as plt
df_pandas = df_completed_studies.toPandas()
df_pandas.plot(kind="bar", x="Month", y="Number_of_Completed_Studies", color = 'red')
plt.show()
```

▸ (4) Spark Jobs



Command took 5.53 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 12:38:19 PM on My Cluster

FURTHER ANALYSIS

1. Finding the top 10 sponsors with completed studies Using RDD

The top ten sponsors who have completed studies can further be found by filtering the clinicaltrial dataset for only 'Completed' studies.

We first select the status column and filter it to include only those studies in which the status is 'completed' and assign it to an RDD called completed_studies. We extract the sponsor column by splitting the pipe delimited string and selecting the second column which is the sponsor column.

Further, we count the number of completed studies for each sponsor by creating a new RDD that uses the sponsor name as the key and a value of 1 and then reducing by key to add up the values for each sponsor. The result is sorted by count of completed studies.

```
# To find the sponsors with the highest completed studies
# we first filter out all the completed studies

Completed_Studies = clinicaltrial_RDD_NH.filter(lambda x: x.split("|")[2] == "Completed")

#selecting only the sponsor
Sponsors_with_completed_studies = Completed_Studies.map(lambda x: x.split('|')[1])

#counting, grouping and sorting sponspors with completed studies in descending order according to count
count_of_sponsors_with_completed_studies  = Sponsors_with_completed_studies.map(lambda x: (x, 1)).reduceByKey(lambda a, b: a + b).sortBy(lambda x: x[1], ascending=False)
#selecting the top 10 and displaying them
count_of_sponsors_with_completed_studies.take(10)
for sponsor, count in count_of_sponsors_with_completed_studies.take(10):
    print("{:<40}{}".format(sponsor, count))
```

▸ (4) Spark Jobs

```
GlaxoSmithKline                         2929
AstraZeneca                             2125
Pfizer                                  1949
National Cancer Institute (NCI)         1897
Boehringer Ingelheim                    1712
Merck Sharp & Dohme Corp.               1693
Novartis Pharmaceuticals                1411
Hoffmann-La Roche                       1379
Eli Lilly and Company                   1355
Bayer                                   1214
```

Command took 3.31 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/26/2023, 11:34:07 AM on My Cluster

2. Creation of a User Defined Function to Count distinct items in each column

We can create a user defined function that can be used to easily extract and count all the distinct elements in each column. It is a User Defined Function because it can be applied to any column and will just require changing the column name.
First, we import the necessary functions from pyspark.sql.functions

```
from pyspark.sql.functions import desc, col
from pyspark.sql.types import IntegerType
```

Secondly, we define the UDF to extract the count from a row object such as the columns we have in the dataset.

```
# Define the UDF to extract the count from a Row object
def extract_count(row):
    if isinstance(row, int):
        return row
    else:
        return row['count']
```

‘extract_count’ is the name of the function and it takes an input row and extract the value of the count of the column from the row. we use the ‘if’ statement and ‘isinstance’ function to return the value as it is if it is an integer (which means that the count is already extracted)
Next is to register the extract_count function with pyspark as extract_count_udf using the ‘udf’ function. The IntegerType() is used to show that the UDF will return the row as integer (countable)

```
# Register the UDF with Spark
extract_count_udf = udf(extract_count, IntegerType())
```

We can now use the UDF to compute any column count as follows.

```
# Compute the COLUMN counts
COLUMN_counts = (clinicaltrial_df
                 .groupBy('COLUMN')
                 .count()
                 .withColumn('count', extract_count_udf(col('count')))
                 .orderBy(desc('count'))
                )

COLUMN_counts.show()
```

groupBy() function is used to group the column according to the elements in the column, and the count() method is called to count the number of rows in each group. withColumn method is used to replace the ‘count’ column with the result obtained when the extract_count_udf is applied to the ‘count’ column. Finally, the orderBy method is called to sort the DataFrame in descending order of the ‘count’ column.
All we have to be doing is substitute any COLUMN in the above code with any column name in our dataframe and it will display the result of the count of all the distinct elements in the column.

We can use it to find all the Status and their frequency as below

```
# Compute the Status counts
Status_counts = (clinicaltrial_df
                 .groupBy('Status')
                 .count()
                 .withColumn('count', extract_count_udf(col('count')))
                 .orderBy(desc('count'))
                )

Status_counts.show()
```

▸ (2) Spark Jobs

▸ Status_counts: pyspark.sql.dataframe.DataFrame = [Status: string, count: integer]

```
+--------------------+------+
|              Status| count|
+--------------------+------+
|           Completed|209749|
|          Recruiting| 60950|
|      Unknown status| 44608|
|          Terminated| 22285|
|Active, not recru...| 17848|
|  Not yet recruiting| 16499|
|           Withdrawn|  9973|
|Enrolling by invi...|  3682|
|           Suspended|  1598|
| No longer available|    39|
|Approved for mark...|    24|
|           Available|     5|
|Temporarily not a...|     1|
+--------------------+------+
```

Command took 4.03 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 7:23:38 AM on My Cluster

Computing for sponsor's count

```
# Compute the Sponsor counts
Sponsor_counts = (clinicaltrial_df
                  .groupBy('Sponsor')
                  .count()
                  .withColumn('count', extract_count_udf(col('count')))
                  .orderBy(desc('count'))
                 )

Sponsor_counts.toPandas()
```

▸ (4) Spark Jobs

▸ Sponsor_counts: pyspark.sql.dataframe.DataFrame = [Sponsor: string, count: integer]

	Sponsor	count
0	GlaxoSmithKline	3378
1	National Cancer Institute (NCI)	3218
2	AstraZeneca	2691
3	Pfizer	2645
4	M.D. Anderson Cancer Center	2414
...	...	...
34437	Hôpital Militaire De Rabat	1
34438	Bayfront Health St Petersburg	1
34439	H.K.E.S's S.Nijalingappa Institute of Dental S...	1
34440	Organisation for Rural Community Development, ...	1
34441	Institut de l'Atherothrombose	1

34442 rows × 2 columns

Command took 4.78 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 3:49:

3. Creation Of Special Views In Spark SQL

We can create a special view that allows us to see the length of time it took to complete a particular study.

In this case, we create a view called 'Study_Period' that contains the study ID, Start and complete dates (in string format), start and complete dates(converted to date format) and the difference between the start and complete date in months aliased as months_diff. The CAST function is used to convert the start and completion columns from string to date.

```
--create a view of all completed studies within a period
CREATE OR REPLACE TEMP VIEW Study_Period AS

SELECT Id, Start, Completion,

       CAST(TO_DATE(Start, 'MMM yyyy') AS DATE) AS start_date,

       CAST(TO_DATE(Completion, 'MMM yyyy') AS DATE) AS completion_date,

       DATEDIFF(MONTH, CAST(TO_DATE(Start, 'MMM yyyy') AS DATE), CAST(TO_DATE(Completion, 'MMM yyyy') AS DATE)) AS months_diff

FROM default.clinicaltrial
```

OK

Command took 0.69 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 5:51:19 AM on My Cluster

We can use the select to see the view.

```
select * from Study_period
```

(1) Spark Jobs

Table +

	Id	Start	Completion	start_date	completion_date	months_diff
1	NCT02758028	Aug 2005	Nov 2021	2005-08-01	2021-11-01	195
2	NCT02751957	Jul 2016	Jul 2020	2016-07-01	2020-07-01	48
3	NCT02758483	Mar 2017	Jan 2018	2017-03-01	2018-01-01	10
4	NCT02759848	Jan 2012	Dec 2014	2012-01-01	2014-12-01	35
5	NCT02758860	Jun 2016	Sep 2020	2016-06-01	2020-09-01	51
6	NCT02757209	Apr 2016	Jan 2018	2016-04-01	2018-01-01	21
7	NCT02752438	May 2016	Jul 2017	2016-05-01	2017-07-01	14

Finally, we can use the **WHERE** clause to select or to count all the studies completed within a particular time frame like within 12 months

View,

```
select * from study_period where months_diff <= 12
```

(1) Spark Jobs

Table +

	Id	Start	Completion	start_date	completion_date	months_diff
1	NCT04468815	Jul 2020	Oct 2020	2020-07-01	2020-10-01	3
2	NCT04462822	Aug 2020	Dec 2020	2020-08-01	2020-12-01	4
3	NCT04464603	Oct 2021	Feb 2022	2021-10-01	2022-02-01	4
4	NCT04464954	Nov 2020	Apr 2021	2020-11-01	2021-04-01	5
5	NCT04466501	Jun 2019	Sep 2019	2019-06-01	2019-09-01	3
6	NCT04465877	Jun 2020	Feb 2021	2020-06-01	2021-02-01	8
7	NCT04467008	Jul 2020	Aug 2020	2020-07-01	2020-08-01	1

10,000 rows | Truncated data | 1.62 seconds runtime

```
Command took 1.62 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 5:49:25 AM on My Cluster
```

Count,

```
--counting all the studies completed in 12 months or less
select count(*) from study_period where months_diff <= 12
```

(2) Spark Jobs

Table +

	count(1)
1	95018

1 row | 3.55 seconds runtime

```
Command took 3.55 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk
```

4. TRANSFORMATION AND ANALYSIS WITH POWER BI

The dataframe, clinicaltrial_df can be refined and saved into PowerBI from where it can be further analysed.

First, we replace all commas ',' in any column with ' OR '. This is because Spark SQL cannot save

a table with "," in it. We do this with regular expression as follows:

```
from pyspark.sql import functions as F

# Replace commas with " OR " in all columns of the DataFrame
clinicaltrial_df_BI = clinicaltrial_df.select([F.regexp_replace(col, ',', ' OR ').alias(col) for col in clinicaltrial_df.columns])

clinicaltrial_df_BI.show()
```

▸ (1) Spark Jobs

▸ clinicaltrial_df_BI: pyspark.sql.dataframe.DataFrame = [Id: string, Sponsor: string ... 7 more fields]

```
+-----------+--------------------+--------------------+--------+----------+--------------------+----------+--------------------+--------------------+
|         Id|             Sponsor|              Status|   Start|Completion|                Type|Submission|          Conditions|       Interventions|
+-----------+--------------------+--------------------+--------+----------+--------------------+----------+--------------------+--------------------+
|NCT02758028|The University of...|          Recruiting|Aug 2005|  Nov 2021|      Interventional|  Apr 2016|                null|                null|
|NCT02751957|     Duke University|           Completed|Jul 2016|  Jul 2020|      Interventional|  Apr 2016|Autistic Disorder...|                null|
|NCT02758483|Universidade Fede...|           Completed|Mar 2017|  Jan 2018|      Interventional|  Apr 2016|   Diabetes Mellitus|                null|
|NCT02759848|Istanbul Medeniye...|           Completed|Jan 2012|  Dec 2014|       Observational|  May 2016|Tuberculosis OR L...|                null|
|NCT02758860|University of Rom...|Active OR  not re...|Jun 2016|  Sep 2020|Observational [Pa...|  Apr 2016|Diverticular Dise...|                null|
|NCT02757209|Consorzio Futuro ...|           Completed|Apr 2016|  Jan 2018|      Interventional|  Apr 2016|              Asthma|Fluticasone OR Xh...|
|NCT02752438|   Ankara University|      Unknown status|May 2016|  Jul 2017|Observational [Pa...|  Apr 2016|     Hypoventilation|                null|
```

The new dataframe clinicaltrial_df_BI is then written into a csv file called clinicaltrial_BI.csv as follows.

```
# Saving the newly created dataframe as a csv file name clinicaltrial_BI
clinicaltrial_df_BI.write.csv("/FileStore/tables/clinicaltrial_BI.csv", header=True, mode="overwrite")
```

▸ (1) Spark Jobs

Command took 10.32 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 8:07:22 AM on My Cluster

Next, we create a spark sql delta table , name it clinicaltrial_BI and saved the CSV file into it.

```
%sql
--- creating a table named clinicaltrial_BI and writing the file into it
DROP TABLE IF EXISTS clinicaltrial_BI;
-- create the table using the CSV file
CREATE TABLE clinicaltrial_BI
USING csv
OPTIONS (
  path "dbfs:/FileStore/tables/clinicaltrial_BI.csv",
  header "true",
  schema "inferSchema"
);
```

▸ (1) Spark Jobs

▸ _sqldf: pyspark.sql.dataframe.DataFrame

OK

Command took 1.78 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 8:08:11 AM on My Cluster

The table is seen inside the FileStore default database.

```
%sql
show tables
```

▸ _sqldf: pyspark.sql.dataframe.DataFrame = [database: string, tableName: string ... 1 more field]

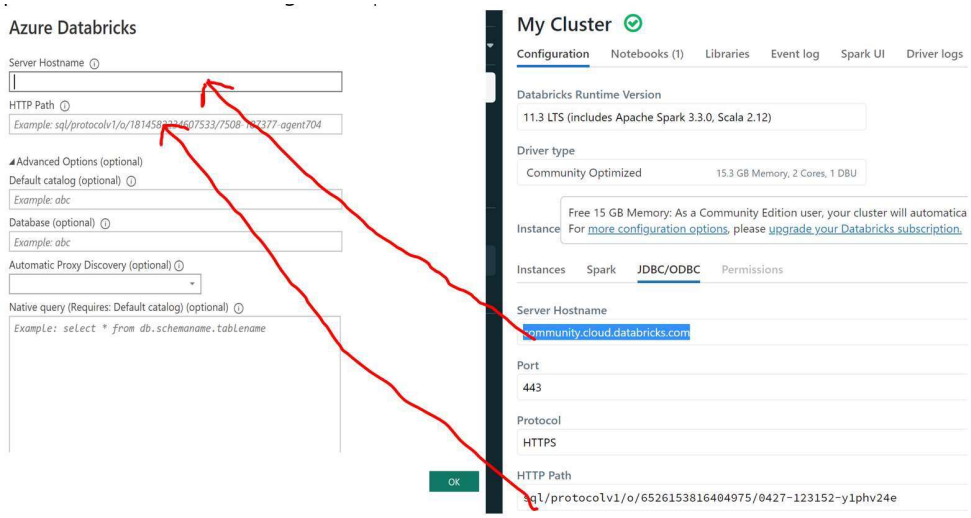
Table +

	database	tableName	isTemporary
1	default	clinicaltrial	false
2	default	clinicaltrial_bi	false
3	default	pharma	false

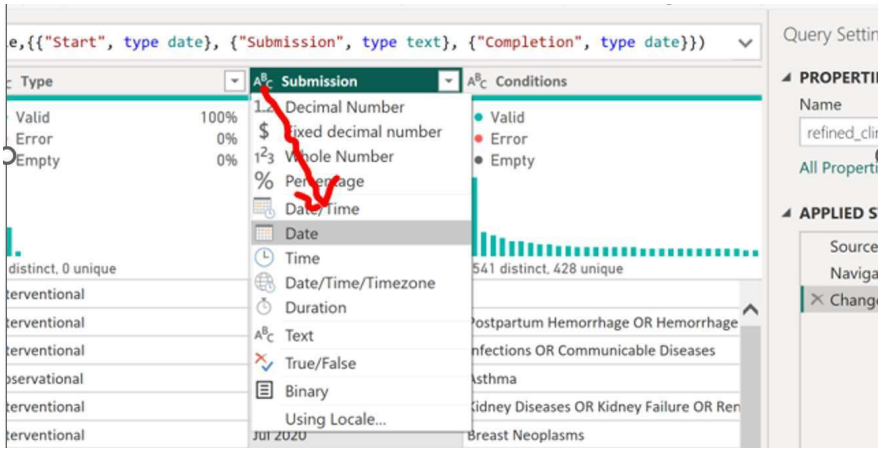
3 rows | 0.32 seconds runtime

Command took 0.32 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 8:10:51 AM on My Cluster

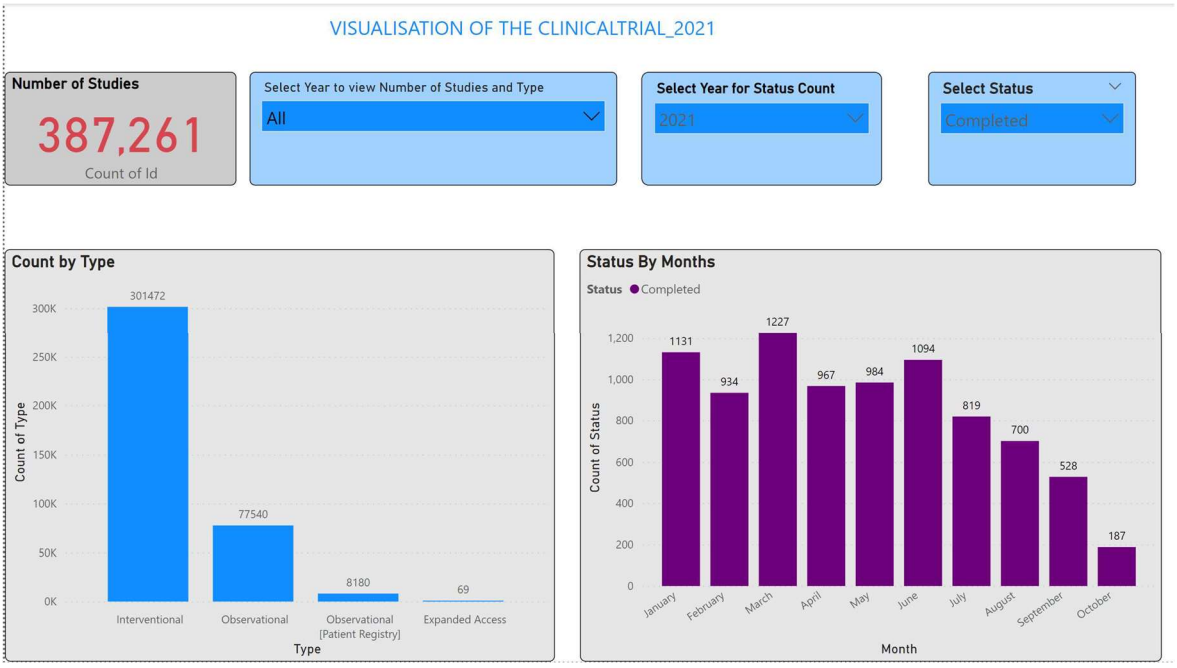
In the Power BI interface, we point to the Get data tool and from the drop down we select the 'more' option and type in databricks. This brought out the Azure Databricks interface where we input the details of the running cluster as shown below.



We follow the rest prompts to get the clinicaltrial_BI into the power BI interface as a table.
In the power BI interface, the clinicaltrial_BI was preprocessed by transforming the Start, Completion and Submission columns into date datatype as shown below

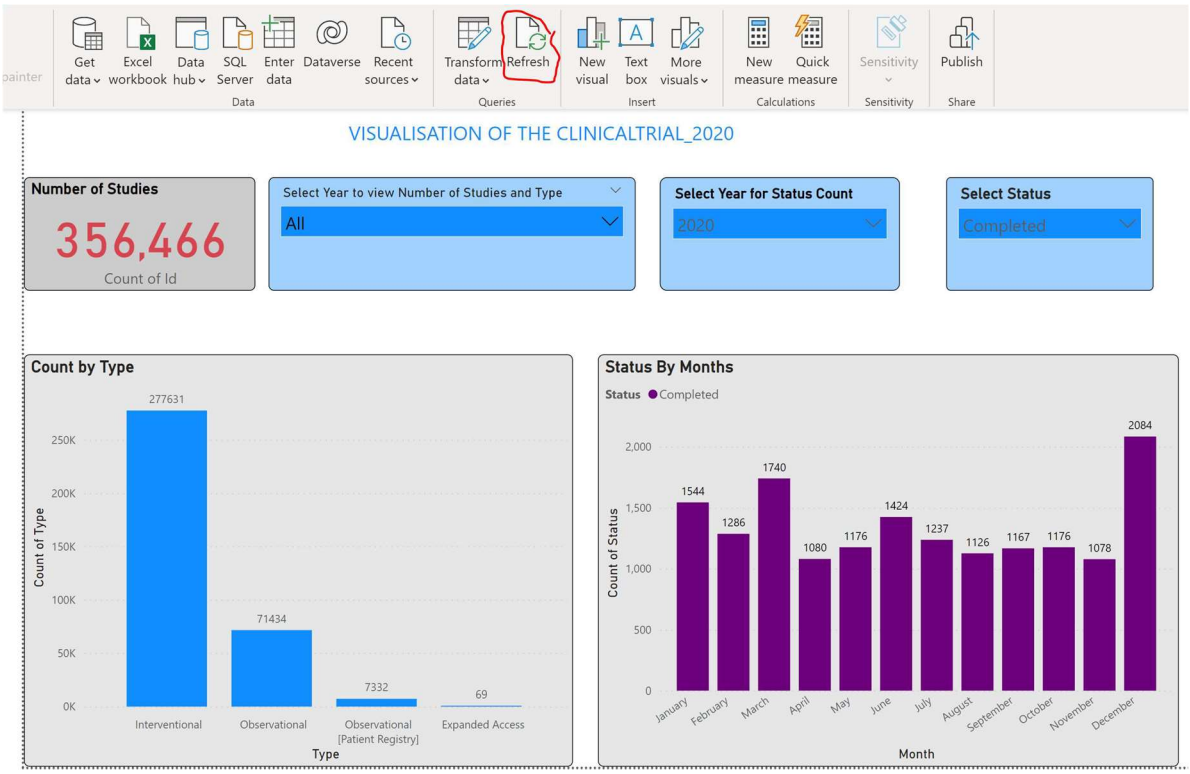


After the transformation, the table was loaded into the Power BI interface to create an interactive dashboard
2021

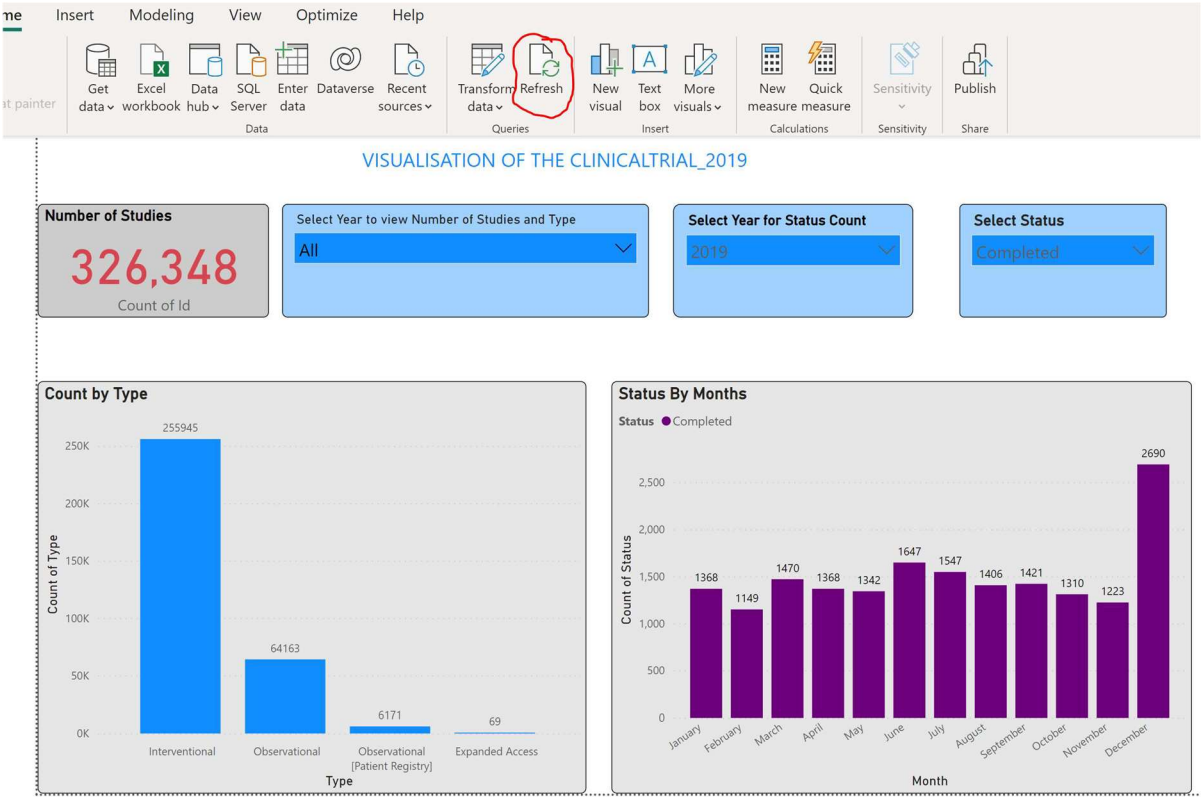


By changing the year variable in the databricks run of the clinicaltrial dataset, we can easily regenerate another visualization for the chosen year by clicking the Refresh button in the powerBI toolbar
Below are the visualizations for 2020 and 2019

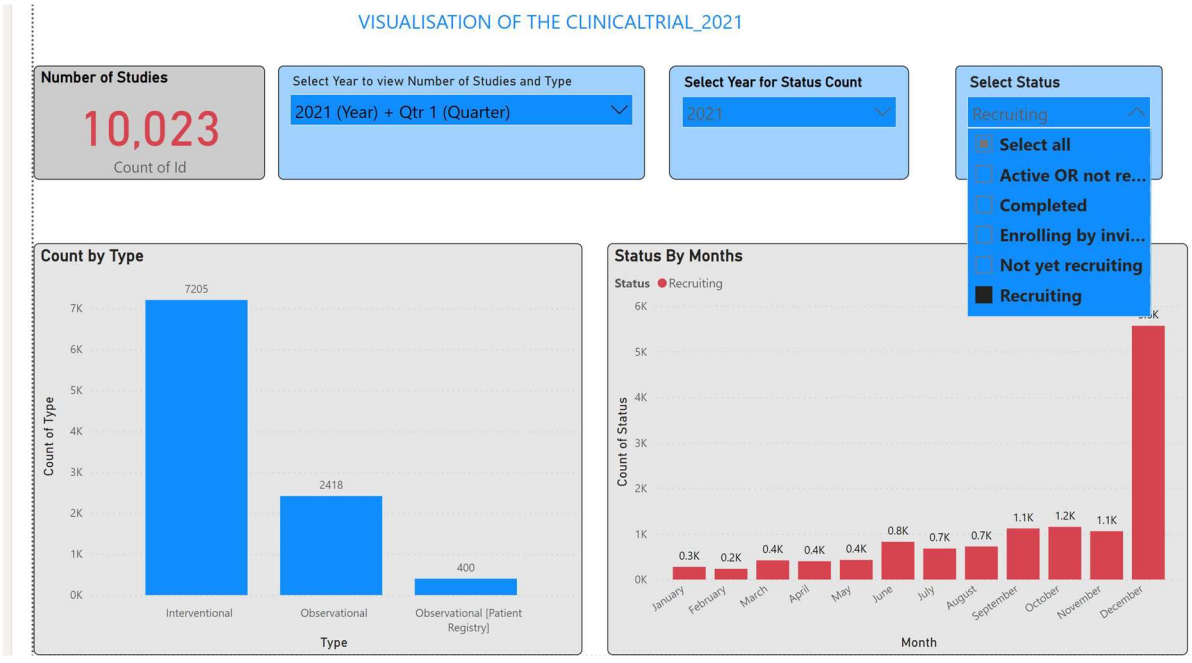
2020



2019



With the visualization dashboard, it is possible to get more insights into the clinicaltrial dataset. For Instance, we can get the number of studies having status as 'Recruiting' for any particular month and for any particular year by selecting Recruiting in the '**Select Status**' Slicer and select the year say 2021 in the '**Select year for Status Count'** slicer.
Also, we can easily get the count of studies for any particular year, quarter, month or even day by selecting the period from '**Select Year to view No of Studies and Type**' slicer.



The visualization above shows

1. The number of studies for each month that has status as ‘Recruiting’

2. The number of studies that were carried quartre 1 of 2021 and the count by type of those studies.

Conclusion: The datasets have been analysed using different analytical tools. The result obtained for each analysis are the same. The results are also logical conclusions from the datasets as they all conform to the datasets. There are no outliers or incoherent results.

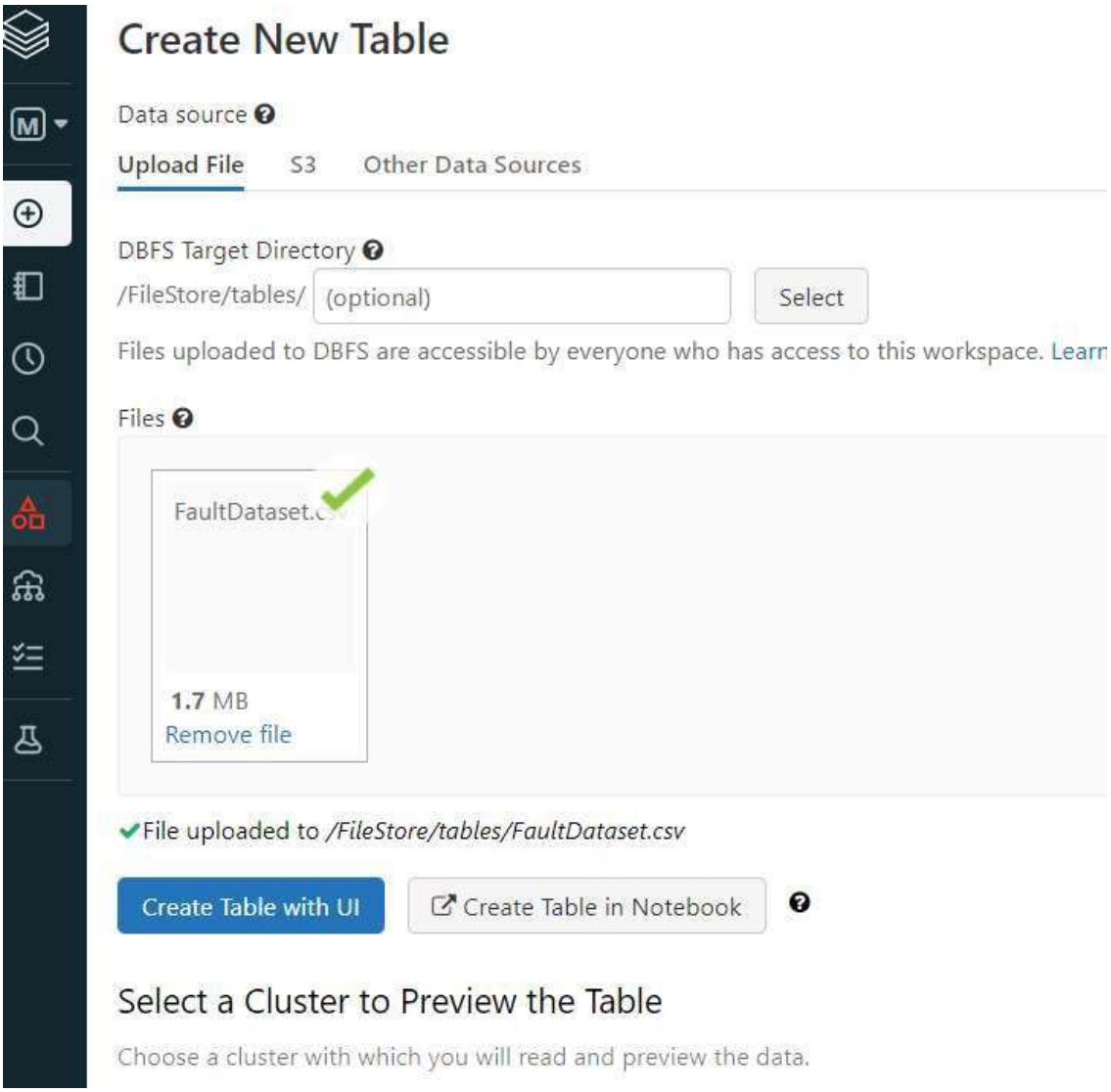
TASK 2

INTRODUCTION

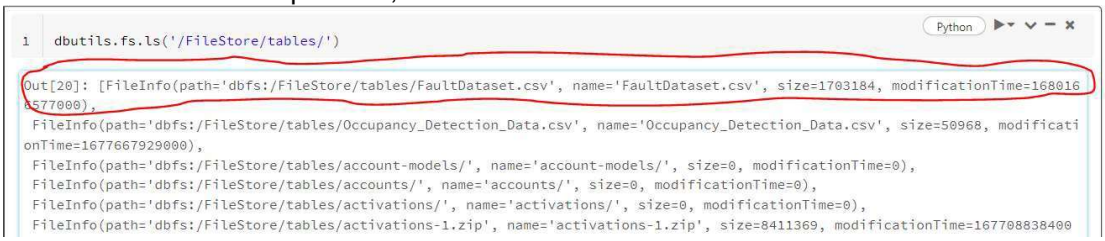
The task requires carrying a classification procedures on the provided Faultdataset.csv using machine learning tools contained in the databricks environment

SETUP

The analysis is performed in databricks community edition using the Machine Learning ML runtime version 12.1ML. The first task is to import the csv dataset into the databricks community edition workspace using the upload interface.



Once the dataset was uploaded, it was found to exist at /FileStore/tables/FaultDataset.csv.

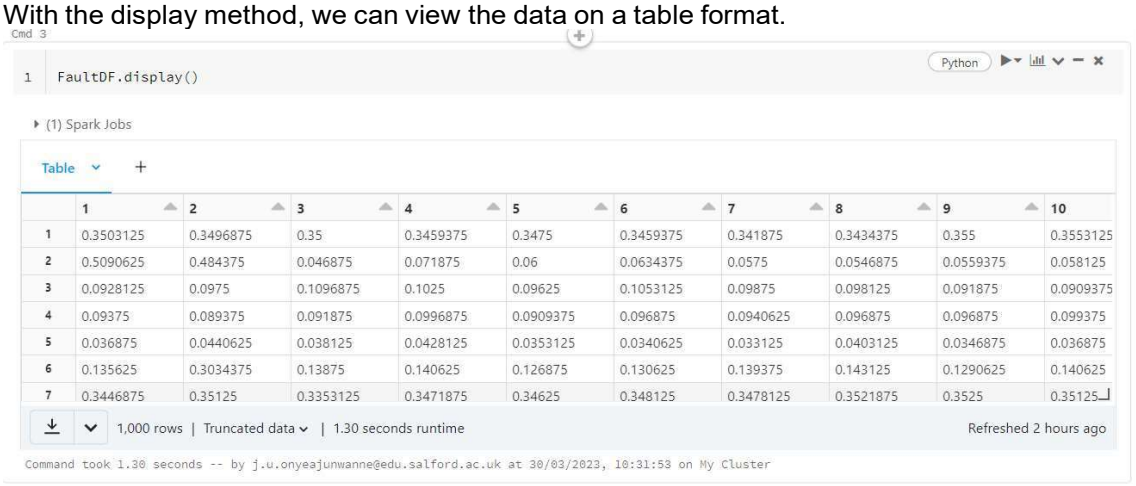


It is important we import the MLflow library as this is what is needed to enable us do machine learning analysis. We have to also enable **autologging** to allow automatic logging of machine learning metadata and artifacts without requiring manual code changes which will later help in identifying the best performing hyperparameters and configurations.

Data Exploration and Preprocessing

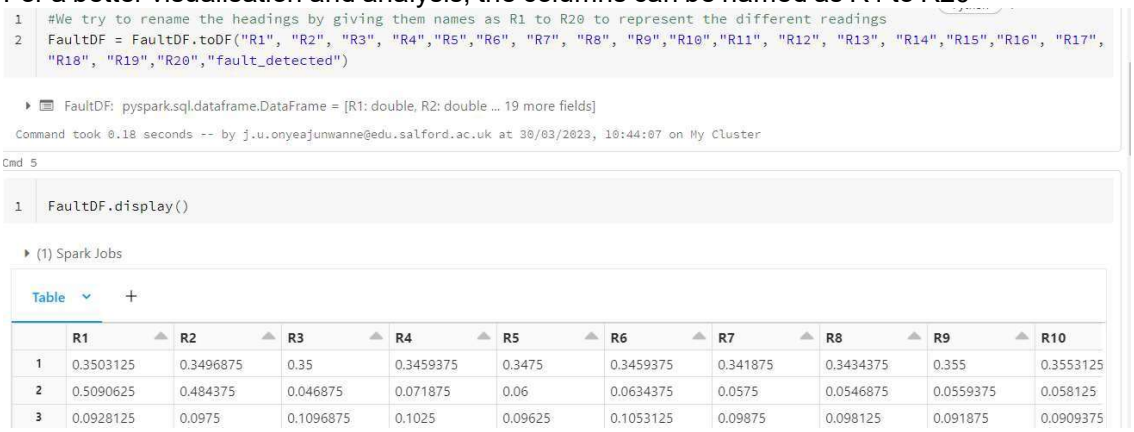
At this stage, we load the dataset into spark dataframe. The header is set to true to ensure that the first row is recognised as the header. The inferSchema is set to true to ensure that PySpark will automatically try to infer the schema of the input data by examining the data in the input file.

With the display method, we can view the data on a table format.



	1	2	3	4	5	6	7	8	9	10
1	0.3503125	0.3496875	0.35	0.3459375	0.3475	0.3459375	0.341875	0.3434375	0.355	0.3553125
2	0.5090625	0.484375	0.046875	0.071875	0.06	0.0634375	0.0575	0.0546875	0.0559375	0.058125
3	0.0928125	0.0975	0.1096875	0.1025	0.09625	0.1053125	0.09875	0.098125	0.091875	0.0909375
4	0.09375	0.089375	0.091875	0.0996875	0.0909375	0.096875	0.0940625	0.096875	0.096875	0.099375
5	0.036875	0.0440625	0.038125	0.0428125	0.0353125	0.0340625	0.033125	0.0403125	0.0346875	0.036875
6	0.135625	0.3034375	0.13875	0.140625	0.126875	0.130625	0.139375	0.143125	0.1290625	0.140625
7	0.3446875	0.35125	0.3353125	0.3471875	0.34625	0.348125	0.3478125	0.3521875	0.3525	0.35125

For a better visualisation and analysis, the columns can be named as R1 to R20



	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
1	0.3503125	0.3496875	0.35	0.3459375	0.3475	0.3459375	0.341875	0.3434375	0.355	0.3553125
2	0.5090625	0.484375	0.046875	0.071875	0.06	0.0634375	0.0575	0.0546875	0.0559375	0.058125
3	0.0928125	0.0975	0.1096875	0.1025	0.09625	0.1053125	0.09875	0.098125	0.091875	0.0909375

A temporary view of that was created and spark SQL is used to query the data for further exploration.

First, we check the average of each column (reading) that resulted in fault detection or no-fault detection.

Cmd 6

```
FaultDF.createOrReplaceTempView('FaultData')
```

Command took 0.20 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 30/03/2023, 10:54:44 on My Cluster

Cmd 7

```
%sql
select fault_detected, AVG(R1) as R1,AVG(R2) as R2,AVG(R3) as R3,AVG(R4) as R4,AVG(R5) as R5,AVG(R6) as R6,AVG(R7) as
R7,AVG(R8) as R8,AVG(R9) as R9,AVG(R10) as R10,AVG(R11) as R11,AVG(R12) as R12,AVG(R13) as R13,AVG(R14) as R14,AVG(R15) as
R15,
AVG(R16) as R16,AVG(R17) as R17,AVG(R18) as R18,AVG(R19) as R19,AVG(R20) as R20
from FaultData
group by fault_detected
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [fault_detected: integer, R1: double ... 19 more fields]

Table +

	fault_detected	R1	R2	R3	R4	R5	R6
1	1	0.5345139636246223	0.5351908227507541	0.5354065997632375	0.5359103933491173	0.5363514986009466	0.53627300
2	0	0.14873264636246228	0.15007149967714162	0.1488361628282389	0.1483677491390444	0.1493353825871714	0.14938286

A count of each column was done. From the result, there is equal number of positive and negative test results i.e 50% of the test data resulted to fault being detected while 50% also resulted to no fault detection.

```
%sql
select fault_detected, AVG(R1) as R1,AVG(R2) as R2,AVG(R3) as R3,AVG(R4) as R4,AVG(R5) as R5,AVG(R6) as R6,AVG(R7) as R7,AVG(R8) as R8,AVG(R9) as R9,AVG(R10) as
R10,AVG(R11) as R11,AVG(R12) as R12,AVG(R13) as R13,AVG(R14) as R14,AVG(R15) as R15,
AVG(R16) as R16,AVG(R17) as R17,AVG(R18) as R18,AVG(R19) as R19,AVG(R20) as R20
from FaultData
group by fault_detected
```

(2) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [fault_detected: integer, R1: double ... 19 more fields]

Table +

	fault_detected	R1	R2	R3	R4	R5	R6	R7
1	1	0.5345139636246223	0.5351908227507541	0.5354065997632375	0.5359103933491173	0.5363514986009466	0.5362730036590614	0.5369834938656909
2	0	0.14873264636246228	0.15007149967714162	0.1488361628282389	0.1483677491390444	0.1493353825871714	0.14938286967283676	0.1484459077701255

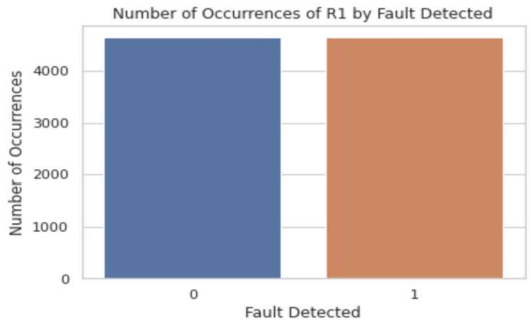
2 rows | 4.70 seconds runtime Refreshed 14 days ago

Command took 4.70 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 3/30/2023, 11:05:42 AM on My Cluster

Visualization Using Barplot

Since all the readings are same, we can visualize one of the readings, R1

```
#Visualising using barplot
import seaborn as sns
import matplotlib.pyplot as plt

# Create a bar plot using the Seaborn library
sns.set(style="whitegrid")
ax = sns.barplot(x="fault_detected", y="R1", data=df)

# Set the plot title and axis labels
ax.set_title("Number of Occurrences of R1 by Fault Detected")
ax.set_xlabel("Fault Detected")
ax.set_ylabel("Number of Occurrences")

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



Command took 0.25 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/28/2023, 7:36

Checking for Missing Values.

We can check for missing values in any of the columns using the code below.

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum
null_counts = FaultDF.agg(*[sum(col(c).isNull().cast("int")).alias(c) for c in FaultDF.columns])
print("Null counts:")
null_counts.show()
```

▸ (2) Spark Jobs

▸ null_counts: pyspark.sql.dataframe.DataFrame = [R1: long, R2: long ... 21 more fields]

```
Null counts:
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+--------------+--------+-----+
| R1| R2| R3| R4| R5| R6| R7| R8| R9|R10|R11|R12|R13|R14|R15|R16|R17|R18|R19|R20|fault_detected|features|label|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+--------------+--------+-----+
|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|  0|             0|       0|    0|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+--------------+--------+-----+
```

Command took 2.07 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 30/03/2023, 14:04:39 on My Cluster

From the result, there is no missing value.

Feature Transformation

Next the RFormula transformer is used to transform the features into a vector format. Since we are using all the features in the dataset, we use the class label:

fault_detected ~.

```python
#we convert the features to a vector
from pyspark.ml.feature import RFormula
preprocess = RFormula(formula = "fault_detected ~ .")
FaultDF = preprocess.fit(FaultDF).transform(FaultDF)
FaultDF.show(5)
```

Show result

▾ (1) MLflow run

Logged 1 run to an experiment in MLflow. Learn more

Next, the data set is split into a training dataset and a test dataset. This is so that we can train our classification model on the **training data** and then evaluate its performance on the **test data**. This allows us to see how well the model is able to generalize to new, unseen data. In this, 70% of the data is used for training while 30% is used for testing. seed rate of 100 is used. The seed is used to initialize a random number generator, and to ensure that the split is consistent across runs and to make our results more reproducible

To do this, we use the code below

```python
#splitting the data into training and test datasets
(trainingDF, testDF) = FaultDF.randomSplit([0.7,0.3], seed =100)
```

▸ trainingDF: pyspark.sql.dataframe.DataFrame = [R1: double, R2: double ... 21 more fields]

▸ testDF: pyspark.sql.dataframe.DataFrame = [R1: double, R2: double ... 21 more fields]

Command took 0.14 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 30/03/2023, 12:10:15 on My Cluster

TRAINING THE MODEL

Decision tree algorithm is used for classification. The first step here is to instantiate an instance of the algorithm and then specify the features and label as contained in the data frame. Thereafter the fit() method is used to fit the model to our data.

```python
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(labelCol="label", featuresCol = "features")
model =dt.fit(trainingDF)
```

Cancel ••• Running command...

▸ (14) Spark Jobs

▾ (1) MLflow run

Logged 1 run to an experiment in MLflow. Learn more

EVALUATION OF THE MODEL

There are different metrics for evaluating a classification model. In this model we consider two.

1. Accuracy: This is the percentage of correctly classified instances out of all instances
2. Confusion Matrix: a table that shows the number of true positives, false positives, true negatives, and false negatives for a given classification threshold.

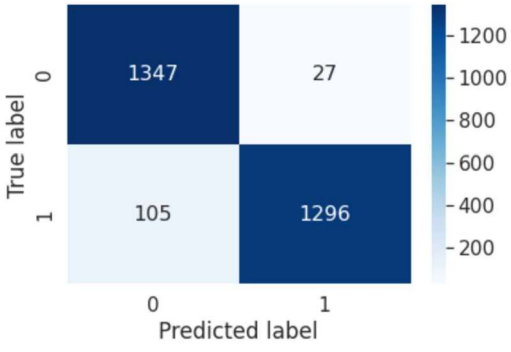
We are going to use accuracy metrics to analyse the performance of this classification model. This will be able to tell us the percentage of the samples that were correctly predicted. We instantiate the evaluator, use the evaluate method to generate the metric and finally use the print method to return the value of accuracy as shown below.

```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
evaluator = MulticlassClassificationEvaluator(labelCol="label", predictionCol = "prediction", metricName="accuracy")

accuracy = evaluator.evaluate(predictions)

print("Accuracy =%g " %(accuracy))

▸ (1) Spark Jobs
Accuracy =0.952432
Command took 1.88 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 30/03/2023, 15:21:27 on My Cluster
```

We can see from this that the accuracy of the model is 0.952432 which means that the 95.24% of the predictions made by the model are correct.

Confusion Matrix

```
#Confusion Matrix
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
# Convert the predicted labels and actual labels to numpy arrays
y_pred = predictions.select('prediction').rdd.map(lambda x: x[0]).collect()
y_true = predictions.select('label').rdd.map(lambda x: x[0]).collect()
# Create the confusion matrix
cm = confusion_matrix(y_true, y_pred)
# Plot the confusion matrix
sns.set(font_scale=1.4)
sns.heatmap(cm, annot=True, annot_kws={"size": 16}, cmap='Blues', fmt='g')
plt.xlabel('Predicted label')
plt.ylabel('True label')
plt.show()

▸ (2) Spark Jobs
```



Using Other Classification Algorithms:

1. **Random Forest Classifier:** individual trees in random forest produce a class of prediction and the class with most votes become the model's prediction.

The code below can be used to realize the random forest algorithm in databricks environment.

We first import and initialise the classifier.

```
#Using Random Forest Classifier
#import the Classifier
from pyspark.ml.classification import RandomForestClassifier

# Initialize the Random Forest Classifier with default parameters
rf = RandomForestClassifier(labelCol="label", featuresCol="features")
```

Command took 0.15 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/13/2023, 11:00:19 PM on My Cluster

The model is then fit to the training data.

```
# Fit the model to the training data
model2 = rf.fit(trainingDF)

# Use the model to make predictions on the test data
predictions2 = model2.transform(testDF)
```

▸ (16) Spark Jobs

▾ (1) MLflow run

Logged 1 run to an experiment in MLflow. Learn more

▸ predictions2: pyspark.sql.dataframe.DataFrame = [R1: double, R2: double ... 24 more fields]

```
2023/04/13 22:00:13 INFO mlflow.utils.autologging_utils: Created MLflow autologging run with ID '1714e360142a411bbbdcd1afb8b9a79a', which will track hyperpara
meters, performance metrics, model artifacts, and lineage information for the current pyspark.ml workflow
2023/04/13 22:00:23 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, which can be time-con
suming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
```

Command took 59.01 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/13/2023, 11:00:13 PM on My Cluster

Checking for the accuracy

```
accuracy = evaluator.evaluate(predictions2)

print("Accuracy =%g " %(accuracy))
```

▸ (1) Spark Jobs

```
Accuracy =0.967928
```

Command took 1.11 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/13/2023, 10:44:05 PM on My Cluster

2. **Logistic Regression Classifier:** In logistic regression, the sum of the input features are computed and used to calculate the logistic of the result.
   As usual, we begin by importing the Logistic Regression Classifier

```
# importing the logistic classifier
from pyspark.ml.classification import LogisticRegression
```

Then, the logistic classifier is initialized with default parameters and it is fit to training dataset to create a prediction model. The Model is finally used to make prediction on the test data.

```
# Initializing the Logistic Regression Classifier with default parameters
lr = LogisticRegression(labelCol="label", featuresCol="features")

# Fitting the model to the training data and using the model to make prediction
model3 = lr.fit(trainingDF)
predictions3 = model3.transform(testDF)
```

▸ (26) Spark Jobs

▾ (1) MLflow run

Logged 1 run to an experiment in MLflow. Learn more

▸ predictions3: pyspark.sql.dataframe.DataFrame = [R1: double, R2: double ... 24 more fields]

```
2023/04/13 21:41:17 INFO mlflow.utils.autologging_utils: Created MLflow autologging run with ID 'a567149df19f47dc8004d3fec42c1e90', which will track hyperpara
meters, performance metrics, model artifacts, and lineage information for the current pyspark.ml workflow
2023/04/13 21:41:26 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repository, which can be time-con
suming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
```

Command took 55.39 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/13/2023, 10:41:17 PM on My Cluster

The accuracy of the prediction model is computed below

```
accuracy = evaluator.evaluate(predictions3)

print("Accuracy =%g " %(accuracy))
```

▸ (1) Spark Jobs

```
Accuracy =0.808649
```

Command took 0.94 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/13/2023, 10:45:09 PM on My Cluster

3. **The Support Vector Machine Algorithm:** in this type of algorithm, the input data is transformed into a high-dimensional space using linear or polynomial kernels. The following codes are used to perform classification using SVM

```
# for support vector Machines
from pyspark.ml.classification import LinearSVC
svm = LinearSVC(labelCol="label", featuresCol="features")
# train the model
model4 = svm.fit(trainingDF)
predictions4 = model4.transform(testDF)
```

▸ (99) Spark Jobs

▾ (1) MLflow run

Logged 1 run to an experiment in MLflow. Learn more

▸ predictions4: pyspark.sql.dataframe.DataFrame = [R1: double, R2: double ... 23 more fields]

```
2023/04/27 19:37:27 INFO mlflow.utils.autologging_utils: Created MLflow autologging run with ID
s, model artifacts, and lineage information for the current pyspark.ml workflow
2023/04/27 19:37:50 INFO mlflow.spark: Inferring pip requirements by reloading the logged model
itly specify the conda_env or pip_requirements when calling log_model().
```

Command took 1.28 minutes -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 8:34:02 PM on machine learning

The accuracy is computed below

```
accuracy = evaluator.evaluate(predictions4)

print("Accuracy =%g " %(accuracy))
```

▸ (1) Spark Jobs

```
Accuracy =0.805766
```

Command took 1.57 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 8:34:02 PM on machine learning

Summary of Results with Default Parameters

Algorithm	Accuracy Achieved
Decision Tree	**0.952432**
Random Forest	**0.969728**
Logistic Regression	**0.808644**
Support Vector machine (SVP)	**0.8057**

From the result, the **Random Forest** algorithm has given the best accuracy with the default hyperparameters. The Logistic Regression algorithm gave the least accuracy and it does not use the stated parameters.

HYPERPARAMETERS

Hyperparameters are the parameters that control the behavior of the training algorithm and influence the performance of the model. They are usually set before training the model and they remain constant during training and are usually determined by the model designer. There are different hyperparameters for different algorithms. Examples of hyperparameters for decision tree and random forest algorithms are impurities, gini, maxdepth. For logistic regression, we have regParam, fitIntercept, elastic NetParam etc. For Support Vector Machine algorithm, we have max_iter, threshold, regParam etc. To achieve a good result in building a classification model, it is important to choose appropriate hyperparameters combination. Azure Databricks allows for the combination of different hyperparameters until the best result is obtained

Hyperparameter Tuning for Decision Tree Algorithm

The following hyperparameters are used in building the model in Decision Tree and Random forest Algorithms

1. maxDepth: This is used in decision tree algorithm to determine the maximum number of depth of the tree. It simply specifies the maximum number of levels that a decision tree can have.
2. Maxbins: this is used to specify the maximum number of bins to employ when continuos numerical features are being converted into categorical features.
3. Impurity: this is used to determine which attribute to split on at each branch node.

The idea here is to combine different hyperparameter values to see what the accuracy will be.

Accuracy	Impurity	Maxbins	Maxdepths
0.952432	gini	32	5
?	gini, entropy	32, 64, 128	5, 7, 9

To begin, we instantiate the paraGridBuilder and then add the value combination which we want to try

```
Cmd 15

from pyspark.ml.tuning import ParamGridBuilder
#  Create parameter grid
parameters=ParamGridBuilder()\
.addGrid(dt.impurity,['gini','entropy'])\
.addGrid(dt.maxDepth, [5,7,9])\
.addGrid(dt.maxBins, [32,64,128])\
.build()

Command took 0.12 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 30/03/2023, 16:47:47 on My Cluster
```

We instantiate the TrainValidatorSplit object. This time we use 75% for training the model unlike the 70% we used for the first model.

```
Cmd 16

from pyspark.ml.tuning import TrainValidationSplit
#Defining train validation Split
tvs = TrainValidationSplit()\
.setSeed(100)\
.setTrainRatio(0.75)\
.setEstimatorParamMaps(parameters)\
.setEstimator(dt)\
.setEvaluator(evaluator)

Command took 0.10 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 30/03/2023, 16:50:15 on My Cluster
```

Here we now fit the model to the training data. This is the process of **hyperparameter tuning** whereby the model will first split the data and then test the combination of the different parameters. there will be 19 runs altogether. One run for data splitting and 2x3x3 =18 runs for the combination of the different parameters.

```
Cmd 17
1  gridsearchModel =tvs.fit(trainingDF)

▸ (59) Spark Jobs
▾ (19) MLflow runs
    Logged 19 runs to an experiment in MLflow. Learn more

2023/03/30 16:05:39 INFO mlflow.utils.autologging_utils: Created MLflow autologging run with ID '806a3480ab654e24b41ee5614fc13bb
f', which will track hyperparameters, performance metrics, model artifacts, and lineage information for the current pyspark.ml wo
rkflow
2023/03/30 16:06:47 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repo
sitory, which can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().
2023/03/30 16:07:45 INFO mlflow.spark: Inferring pip requirements by reloading the logged model from the databricks artifact repo
sitory, which can be time-consuming. To speed up, explicitly specify the conda_env or pip_requirements when calling log_model().

Command took 2.94 minutes -- by j.u.onyeajunwanne@edu.salford.ac.uk at 30/03/2023, 17:05:38 on My Cluster

Cmd 18
```

We can find the best model using the code below.

```
Cmd 18
1  #Selecting the best model and identifying the parameters
2
3  bestModel =gridsearchModel.bestModel
4  print('Parameters for best Model: ')
5  print('MaxDepth Parameter: %g' %bestModel.getMaxDepth())
6  print('Impurity Parameter: %s' %bestModel.getImpurity())
7  print('MaxBins Parameter: %g' %bestModel.getMaxBins())

Parameters for best Model:
MaxDepth Parameter: 7
Impurity Parameter: entropy
MaxBins Parameter: 128

Command took 0.08 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 30/03/2023, 17:24:00 on My Cluster

Cmd 19
```

Tracking the experiment with MLflow, we can see the different models generated for the decision tree algorithm hyperparameter tuning.

Experiments ›

Assignement_Task2 Provide Feedback

Experiment ID: 1243323632912797 Artifact Location: dbfs:/databricks/mlflow-tracking/1243323632912797

› Description Edit

Table view | Chart view | Sort: accuracy | Columns | Refresh

Time created: All time | State: Active

	Run Name	Created	Source	Models	Metrics: accuracy	Metrics: accuracy_testDF	Parameters: impurity	Parameters: maxBins	Parameters: maxDepth
	stately-frog-688	14 minutes ago	Assigne...	-	0.97		entropy	128	7
	brawny-panda-315	14 minutes ago	Assigne...	-	0.969		gini	64	9
	delicate-gnu-219	14 minutes ago	Assigne...	-	0.968		gini	32	9
	merciful-skink-164	14 minutes ago	Assigne...	-	0.968		entropy	64	7
	intelligent-owl-778	14 minutes ago	Assigne...	-	0.966		gini	32	7
	skittish-stork-82	14 minutes ago	Assigne...	-	0.966		entropy	64	9
	rebellious-skink-851	14 minutes ago	Assigne...	-	0.966		entropy	128	9
	amusing-moth-530	14 minutes ago	Assigne...	-	0.964		gini	64	7
	burly-hawk-531	14 minutes ago	Assigne...	-	0.963		entropy	32	9
	secretive-gull-892	14 minutes ago	Assigne...	-	0.962		gini	64	5
	skillful-dog-911	14 minutes ago	Assigne...	-	0.959		entropy	128	5
	big-bear-955	14 minutes ago	Assigne...	-	0.958		gini	128	9

From the above, it is seen that the combination of entropy as the impurity, maxbinBins of 128 and maxDepth of 7, has produced the best accuracy of 0.97.

We can use the new model to make predictions in mlflow as show below

```
import mlflow
logged_model = 'runs:/806a3480ab654e24b41ee5614fc13bbf/best_model'

# Load model
loaded_model = mlflow.spark.load_model(logged_model)

# Perform inference via model.transform()
loaded_predictions=loaded_model.transform(FaultDF)
loaded_predictions.show()
```

```
(7) Spark Jobs
loaded_predictions: pyspark.sql.dataframe.DataFrame = [R1: double, R2: double ... 24 more fields]
```

```
2023/04/28 07:14:07 INFO mlflow.spark: 'runs:/806a3480ab654e24b41ee5614fc13bbf/best_model' resolved as 'dbfs:/databricks/mlflow-tracking/1243323632912797/806a3480ab654e24b41ee5614fc13bb
f/artifacts/best_model'
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+----
-----+---------+--------------+--------------------+-----+-------------+--------------------+----------+
|       R1|       R2|       R3|       R4|       R5|       R6|       R7|       R8|       R9|      R10|      R11|      R12|      R13|      R14|      R15|      R16|      R17|      R18|
R19|      R20|fault_detected|            features|label|rawPrediction|         probability|prediction|
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+----
-----+---------+--------------+--------------------+-----+-------------+--------------------+----------+
|0.3503125|0.3496875|     0.35|0.3459375|   0.3475|0.3459375| 0.341875|0.3434375|    0.355|0.3553125|0.3459375|   0.3525|   0.3575|0.3590625|  0.35875|0.3484375|0.3590625|     0.35|0.35
59375|0.3490625|             0|[0.3503125,0.3496...|  0.0| [2054.0,0.0]|           [1.0,0.0]|       0.0|
|0.5090625| 0.484375| 0.046875| 0.071875|     0.06|0.0634375|   0.0575|0.0546875|0.0559375| 0.058125|0.0628125| 0.065625|0.0640625|0.0634375|0.0534375| 0.084375|0.0615625|  0.05375| 0.0
76875| 0.056875|             0|[0.5090625,0.4843...|  0.0| [23.0,200.0]|[0.10313901345291...|       1.0|
|0.0928125|   0.0975|0.1096875|   0.1025|  0.09625|0.1053125|  0.09875| 0.098125| 0.091875|0.0909375|  0.09875| 0.103125|      0.1|0.1034375|0.1015625|0.0978125|0.0990625|  0.10375| 0.0
98125|0.1040625|             0|[0.0928125,0.0975...|  0.0| [2054.0,0.0]|           [1.0,0.0]|       0.0|
|  0.09375| 0.089375| 0.091875|0.0996875|0.0909375| 0.096875|0.0940625| 0.096875| 0.096875| 0.099375| 0.099375|0.0959375|0.0959375|0.0940625|  0.09125|0.0996875|  0.09375|0.0934375|0.09
71875| 0.094375|             0|[0.09375,0.089375...|  0.0| [2054.0,0.0]|           [1.0,0.0]|       0.0|
| 0.036875|0.0440625| 0.038125|0.0428125|0.0353125|0.0340625| 0.033125|0.0403125|0.0346875| 0.036875| 0.035625|  0.03625|0.0409375| 0.039375|    0.035| 0.040625|0.0384375| 0.036875|
0.04|0.0371875|             0|[0.036875,0.04406...|  0.0| [933.0,14.0]|[0.98521647307286...|       0.0|
| 0.135625|0.3034375|  0.13875| 0.140625| 0.126875| 0.130625| 0.139375| 0.143125|0.1290625| 0.140625|0.1340625|0.1396875|0.1384375|0.1453125|0.1453125|0.1496875|0.1440625|0.1359375|0.14
53125|  0.14625|             0|[0.135625,0.30343...|  0.0| [2054.0,0.0]|           [1.0,0.0]|       0.0|
|0.3446875|  0.35125|0.3353125|0.3471875|  0.34625| 0.348125|0.3478125|0.3521875|   0.3525|  0.35125|0.3571875| 0.360625|0.3640625|  0.36625|0.3640625|0.3634375|   0.3475|  0.35375|
```

Command took 6.41 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/28/2023, 8:07:20 AM on My Cluster

Hyperparameter Tuning for other Algorithms

1. Random Forest Algorithm: the same hyperparameter were used as with Decision tree.

Instantiating the grid parameter

```
from pyspark.ml.tuning import ParamGridBuilder
#  Create parameter grid
parameters=ParamGridBuilder()\
.addGrid(rf.impurity,['gini','entropy'])\
.addGrid(rf.maxDepth, [5,7,9])\
.addGrid(rf.maxBins, [32,64,128])\
.build()
```

Command took 0.04 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 9:34:23 PM o

Instantiating the train validator

```
from pyspark.ml.tuning import TrainValidationSplit
#Defining train validation Split
tvs_rf = TrainValidationSplit()\
.setSeed(100)\
.setTrainRatio(0.75)\
.setEstimatorParamMaps(parameters)\
.setEstimator(rf)\
.setEvaluator(evaluator)
```

Command took 0.10 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023

Fitting the model to the training data

```
gridsearchModel40 = tvs_rf.fit(trainingDF)
```

▸ (61) Spark Jobs

▾ (19) MLflow runs

Logged 19 runs to an experiment in MLflow. Learn more

```
2023/04/27 20:38:33 INFO mlflow.utils.autologging_utils: Created MLflow auto
s, model artifacts, and lineage information for the current pyspark.ml workf
2023/04/27 20:39:55 INFO mlflow.spark: Inferring pip requirements by reloadi
itly specify the conda_env or pip_requirements when calling log_model().
2023/04/27 20:41:08 INFO mlflow.spark: Inferring pip requirements by reloadi
itly specify the conda_env or pip_requirements when calling log_model().
```

Command took 3.60 minutes -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 9:38:33

Finding the best result

```
#Selecting the best model and identifying the parameters
bestModel_rf =gridsearchModel40.bestModel
print('Parameters for best Model: ')
print('MaxDepth Parameter: %g' %bestModel_rf.getMaxDepth())
print('Impurity Parameter: %s' %bestModel_rf.getImpurity())
print('MaxBins Parameter: %g' %bestModel_rf.getMaxBins())
```

```
Parameters for best Model:
MaxDepth Parameter: 9
Impurity Parameter: entropy
MaxBins Parameter: 128
```

Command took 0.07 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023,

Outputting the new accuracy

```
New_accuracy_rf = evaluator.evaluate(bestModel_rf.transform(testDF))

print("New_Accuracy_rf =%g " %(New_accuracy_rf))
```

▸ (1) Spark Jobs

```
New_Accuracy_rf =0.972613
```

Command took 0.76 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 10:02:1

2. **Hyperparameter Tuning for Logistic Regression**: The hyperparameters considered are regParam, fitIntercept, elastic NetParam.

We start Instantiating the grid parameters

```
#instantiating the grid parameters
param_grid_lr = (ParamGridBuilder()
             .addGrid(LogisticRegression.regParam, [0.01, 0.1])
             .addGrid(LogisticRegression.elasticNetParam, [0.0, 0.5])
              .addGrid(LogisticRegression.fitIntercept, [False, True])
             .build())
```

Command took 0.14 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 10:38:47 PM on machine learning

Instantiating the train validator

```
from pyspark.ml.tuning import TrainValidationSplit
#Defining TrainValidationSplit
tvs_lr = TrainValidationSplit()\
.setSeed(100)\
.setTrainRatio(0.75)\
.setEstimatorParamMaps(param_grid_lr)\
.setEstimator(lr)\
.setEvaluator(evaluator)
```

```
Command took 0.12 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 10:3
```

Fitting the new model to the training data

```
# train model using grid search
gridsearchModel23 = tvs_lr.fit(trainingDF)
```

```
▸ (96) Spark Jobs
▾ (1) MLflow run
    Logged 1 run to an experiment in MLflow. Learn more
2023/04/27 21:39:11 INFO mlflow.utils.autologging_utils: Created MLflow autologgi
s, model artifacts, and lineage information for the current pyspark.ml workflow
2023/04/27 21:39:34 WARNING mlflow.utils.autologging_utils: Encountered unexpecte
timator, but found a param undefined__regParam
Command took 23.30 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 10:39:11 PM o
```

Finding the best model

```
bestModel_lr = gridsearchModel23.bestModel
print("Parameters for the Best Model:")
print("regParam Parameter: %g" % bestModel_lr.getOrDefault('regParam'))
print("fitIntercept Parameter: %s" % bestModel_lr.getOrDefault('fitIntercept'))
print("elasticNetParam Parameter: %g" % bestModel_lr.getOrDefault('elasticNetParam'))
```

```
Parameters for the Best Model:
regParam Parameter: 0
fitIntercept Parameter: True
elasticNetParam Parameter: 0
Command took 0.06 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 10:42:28 PM on machine learning
```

Printing the new accuracy

```
New_accuracy_lr = evaluator.evaluate(bestModel_lr.transform(testDF))
 print("New_Accuracy_lr =%g " %(New_accuracy_lr))
```

```
▸ (1) Spark Jobs
New_Accuracy_lr =0.808649
Command took 2.04 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 10:42:33 P
```

3. **Hyperparameter Tuning for SVM.**: the hyperparameters considered are max_iter, threshold, regParam

Instantiating the grid parameters:

```
#Instantiating the grid parameter
paramGrid_svm = ParamGridBuilder() \
    .addGrid(svm.regParam, [0.1, 0.01]) \
    .addGrid(svm.maxIter, [10, 100]) \
    .addGrid(svm.threshold, [0.0, 0.5]) \
    .build()
```

```
Command took 0.05 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023,
```

Defining the train validator and fitting the model to the training data

```
from pyspark.ml.tuning import TrainValidationSplit
#Defining TrainValidationSplit
tvs_svm = TrainValidationSplit()\
.setSeed(100)\
.setTrainRatio(0.75)\
.setEstimatorParamMaps(paramGrid_svm)\
.setEstimator(svm)\
.setEvaluator(evaluator)
# train model using grid search
gridsearchModel92 = tvs_svm.fit(trainingDF)
```

▸ (97) Spark Jobs

▾ (9) MLflow runs

Logged 9 runs to an experiment in MLflow. Learn more

```
2023/04/27 21:04:16 INFO mlflow.utils.autologging_utils: Created MLflow autologg
s, model artifacts, and lineage information for the current pyspark.ml workflow
2023/04/27 21:05:57 INFO mlflow.spark: Inferring pip requirements by reloading t
itly specify the conda_env or pip_requirements when calling log_model().
2023/04/27 21:07:06 INFO mlflow.spark: Inferring pip requirements by reloading t
itly specify the conda_env or pip_requirements when calling log_model().
```

Command took 3.90 minutes -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 10:04:16 PM

Finding the best combination and printing out the resulting accuracy

```
# Get the best model
bestModel_svm = gridsearchModel92.bestModel
# Print the parameters of the best model
print("Parameters for the Best Model:")
print("RegParam Parameter: %g" % bestModel_svm.getOrDefault('regParam'))
print("MaxIter Parameter: %g" % bestModel_svm.getOrDefault('maxIter'))
print("Tol Parameter: %g" % bestModel_svm.getOrDefault('tol'))
```

```
Parameters for the Best Model:
RegParam Parameter: 0.1
MaxIter Parameter: 10
Tol Parameter: 1e-06
```

Command took 0.05 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 9:21

Printing out the resulting accuracy

```
New_accuracy_svp = evaluator.evaluate(bestModel_svm.transform(testDF))
 print("New_Accuracy_svp =%g " %(New_accuracy_svp))
```

▸ (1) Spark Jobs

```
New_Accuracy_svp =0.851171
```

Command took 1.84 seconds -- by j.u.onyeajunwanne@edu.salford.ac.uk at 4/27/2023, 10:16:3

Summary of Results

Algorithm Type	Initial Accuract	Accuracy with Hyperparameter Tuning	Hyperparameters	% improvement
Decision Tree	0.952432	0.97	Impurity-entropy, maxDepth-7, maxBins-128	1.844541133
Random Forest	0.969728	0.9721613	impurity-entropy, maxDepth-9, maxBins-128	0.250926033

Logistic Regression	**0.808644**	**0.808644**	**regParam- 0, fitIntercept-True, elasticNetParam-0**	**0**
SVP	**0.8057**	**0.8512**	**regParam- 0.1, maxIter-10, Tol Param-0.000006**	**5.647263249**

From the table, it can be seen that there is an improvement in the accuracy of the model after hyperparameter tuning except for logistic regression. The Logistic Regression is generally not a good algorithm to analyze the type of dataset as it works best with linear data.

Decision Tree and Random forest models have given the highest accuracy.

However, the best accuracy is obtained with Random Forest Model by setting entropy as the impurity, maxDepth of 7 and maxBins of 128

Conclusion: The faultDataset has been classified using Decision Tree and other algorithms. The accuracy of the Decision Tree algorithm increased with hyperparameter tuning by 1.84% (from 0.952432 to 0.97) and the best hyperparameters are entropy as impurity, maxDepth of 7 and maxBins of 128.

By testing other algorithms on the dataset, it was discovered that that Random Forest algorithm can give an accuracy of 97.2% with hyperparameter combination of entropy, maxDepth of 9 and maxBins of 128.