Guide

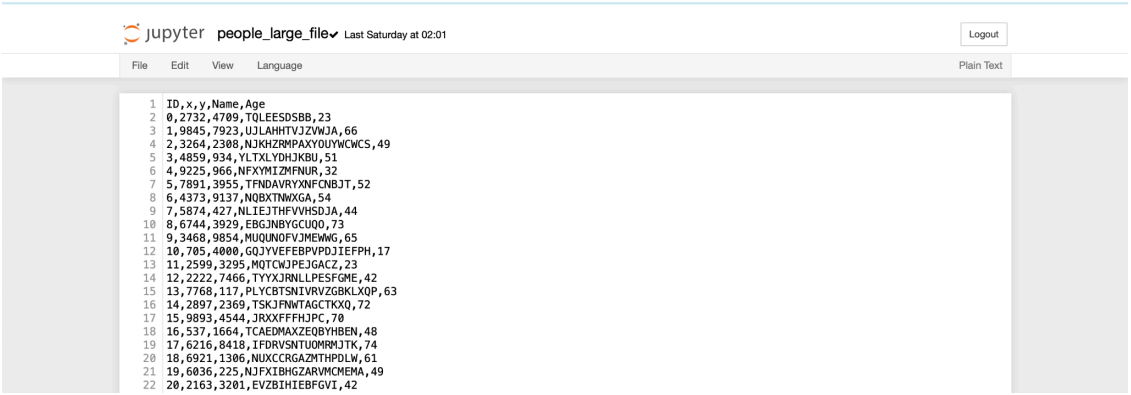
1) Data Creation Part 1:

- Run the InfectedData.ipynb file in a python environment like Jupyter Notebook and change the paths of people_large_file, infected_small_file, and people_some_infected_files. This will generate the csv files for the three datasets we need. Because the number of records was not specified, we set the people_large and people_some_infected to have a count of 10000 and the infected dataset to have a count of 100.

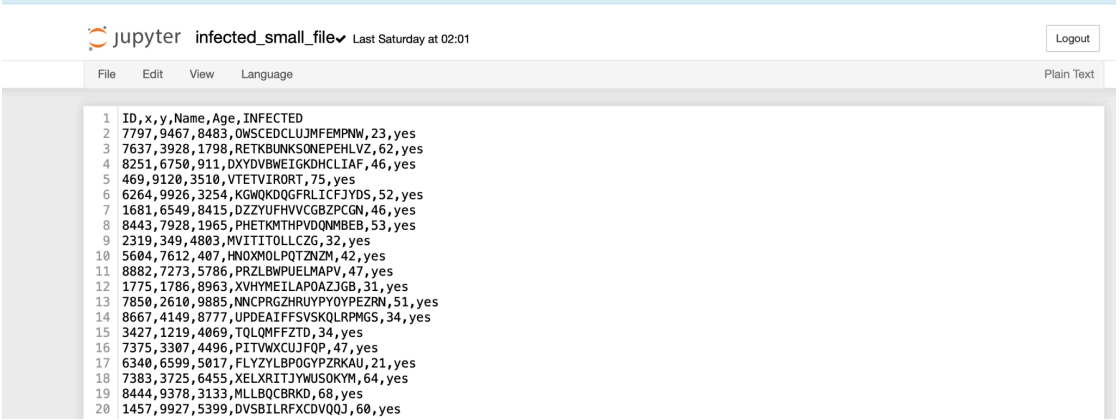
2) Data Creation Part 2:

- Run the PurchaseData.ipynb file in a python environment like Jupyter Notebook and change the addresses of customers_file and purchases_file. We are creating 5 million records, this may take some time to generate. We also elected to save the files outside of the IDE for this reason. We purposely made every single customer have exactly 100 purchases to make sure to fulfill the requirement that a customer should have 100 purchases on average.

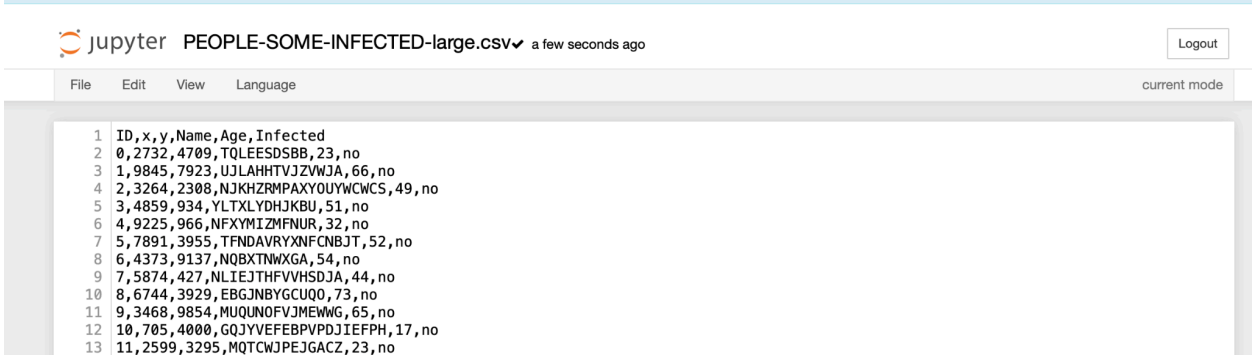
Sample of the people-LARGE file data:



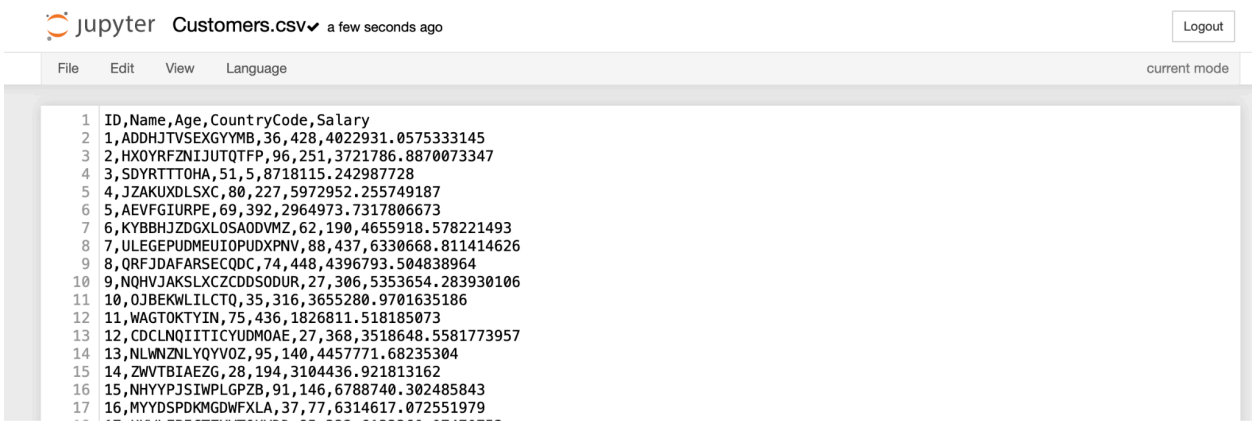
Sample of the infected-SMALL file data:



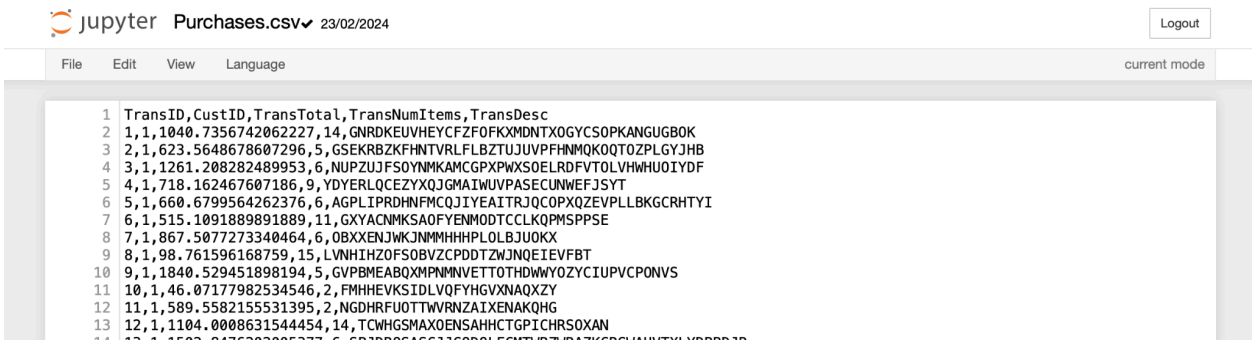
Sample of the people-some-infected-LARGE file data:



Sample for the Customers file data:



Sample for the Purchases file data:



Explanation of Solutions

1) Part 1 Spark RDD
    a) Query 1:
    We first opened up a new SparkConf and a new SparkContext.
    We load in people, which has format RDD[(String, Double, Double)] using sc.textFile('path') with the local path of the PEOPLE-LARGE.csv file. .filter

ignores the first line of the file, and then we split each line into it's id, x, and y which are a String, Double, and Double respectively. We do the same for infected, as it has the same format but we change the path to INFECTED-small.csv.
We then do a cartesian product of these two RDDs and apply a filter and map function on it. We first filter out any instances where the IDs are the same and we then check if the two people are less than 6 units away using the expression math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2)) <= 6. The map function is used to project only the IDs of the two people. The script ends by calling the .collect() on the closeContacts RDD, printing each line, and stopping the SparkContext

b) Query 2:
We do the same process as above by opening the SparkConf and SparkContext, loading and parsing the PEOPLE-large and INFECTED-small RDDs. We then only the IDs of the infected people by projecting only the ID of the infected RDD and getting only the distinct IDs. broadcastInfectedIDs is used to store the new RDD and then we create a nonInfectedPeople RDD which keeps the entries in people who are not in infectedIDs. We then do the same computation of closeContacts by doing the cartesian product and filtering on the same conditions, i.e. making sure the IDs are not the same and that the distance between the two people is <= 6 units. distinct() is used to keep only the distinct ids, and we end the script with the same process as question one with using .collect(), printing each line, and stopping the SparkContext

c) Query 3:
We first opened the SparkConf and the SparkContext like we had done in the previous queries
We then filtered out the first line and split the csv into the different parts. The infected RDD was created from the PEOPLE-SOME-INFECTED csv, where we separated the ones labeled yes and no for infected status as infected and notInfected RDDs. We broadcasted infected to store it. We also then created closeContacts from a cartesian product of the two RDDs under the condition that the IDs were not the same and the distance between the two people were <= 6 units. We then did a mapreduce implementation where we mapped by InfectedId, 1 and reduced to the sum of the 1s to get the count of each infectedID.

2) Part 2 Spark SQL

a) Query 1:
Our whole solution is built in a single main() function. We begin by using the "SparkSession()" function with the appropriate parameters such as the app's name, and the amount of cores we allow our program to use. Next up, we use the spark.read.format() function on the "Purchases.csv" file saved in our downloads while setting it up so that it reads csv files as our datasets are all csv files, and also making sure that it uses the "," as the delimiter, and making sure that the correct file path was used for the function to find the file. Once the file was loaded, we used the createOrReplaceTempView to give the loaded file an

alias to be able to run queries on it. Next, we used the .sql() function with the query "SELECT * FROM purchases WHERE TransTotal <= 600" to be able to only keep the records from the file where the TransTotal was less than or equal to 600. Once that was done, we used the .repartition() function to make sure that all of our results were stored in 1 single file in a folder called T1_temp within our IDE. However, we the used file objects and functions like listfiles(), find(), and renameTo() to make sure that the file “T1.csv” was saved somewhere like the downloads instead of the IDE due to the amount of data inside.

b) Query 2:
Our whole solution is built in a single main() function. We begin by using the “SparkSession()” function with the appropriate parameters such as the app’s name, and the amount of cores we allow our program to use. Next up, we use the spark.read.format() function on the “T1.csv” file obtained from the first query saved in our downloads while setting it up so that it reads csv files as our datasets are all csv files, and also making sure that it uses the “,” as the delimiter, and making sure that the correct file path was used for the function to find the file. Once the file was loaded, we used the createOrReplaceTempView to give the loaded file an alias to be able to run queries on it. Next, we used the .sql() function with the query "SELECT TransNumItems, " + "percentile_approx(TransTotal, 0.5) AS median, " + "MIN(TransTotal) AS min," + "MAX(TransTotal) AS max " + "FROM T1 " + "GROUP BY TransNumItems" group the Purchases in T1 by the Number of Items purchased, and calculate the median, min and max of total amount spent for purchases for each group.. Once that was done, we used the .show() functions to display the results back to the client side on the console.

c) Query 3:
Our whole solution is built in a single main() function. We begin by using the “SparkSession()” function with the appropriate parameters such as the app’s name, and the amount of cores we allow our program to use. Next up, we use the spark.read.format() function on the “T1.csv” file obtained from the first query, and the “Customers.csv” files saved in our downloads while setting it up so that it reads csv files as our datasets are all csv files, and also making sure that it uses the “,” as the delimiter, and making sure that the correct file path was used for the function to find the files. Once the files were loaded, we used the createOrReplaceTempView to give the loaded files an alias to be able to run queries on them. Next, we used the .sql() function with the query "SELECT ID, Age, " + "COUNT(*) AS TotalItems, " + "SUM(TransTotal) AS TotalSpent " + "FROM T1 " + "JOIN customers ON T1.CustID = customers.ID " + "WHERE Age BETWEEN 18 AND 25 " + "GROUP BY ID, Age" to be able to group the Purchases in T1 by customer ID only for young customers between 18 and 25 years of age, and for each group report the customer ID, their age, and total number of items that this person has purchased, and total amount spent by the customer. Once that was done, we used the .repartition() function to make sure that all of our results were stored in 1 single file in a folder called T3_temp within

our IDE. However, we the used file objects and functions like listfiles(), find(), and renameTo() to make sure that the file “T3.csv” was saved somewhere like the downloads instead of the IDE due to the amount of data inside.

3) Part 3 MLlib
    a) Task 1:
    Our whole solution is built in a single main() function. We begin by using the “SparkSession()” function with the appropriate parameters such as the app’s name, and the amount of cores we allow our program to use. Next up, we use the spark.read.format() function on the “Customers.csv” file obtained from the first query, and the “Purchases.csv” files saved in our downloads while setting it up so that it reads csv files as our datasets are all csv files, and also making sure that it uses the “,” as the delimiter, and making sure that the correct file path was used for the function to find the files. Once the files were loaded, we used the createOrReplaceTempView to give the loaded files an alias to be able to run queries on them. Next, Next, we used the .sql() function with the query "SELECT Age, " + "Salary, TransNumItems, " + "TransTotal FROM purchases " + "JOIN customers ON purchases.CustID = customers.ID" to get a dataset composed of customer ID, TransID, Age, Salary, TransNumItems and TransTotal, which we stored in a variable called "dataset". Once the dataset was made, we used the .randomSplit() to randomly split the data from the Dataset file into 80% of it being used for training, and 20% of it being used for testing.
    b) Task 2:
    Continuing from Task 1, as all the required features were numerical, we were able to use an assembler to select the age, salary, and TransNumItems from the split data as the features. We then used a LinearRegression() object to declare a linear regression while setting the input as the features, and the column for which to predict values as Transtotal. Along with this, we also use a GBTRegressor() object to declare a gradient-boosting tree regression while setting the input as the features, and the column for which to predict values as Transtotal. To make sure that we had a sufficient number of iterations to be able to potentially give an accurate prediction, we set the max number of iterations for both regression models as 25. We then used the .fit() and .transform() functions to train both models with the training data, and store the testing data to check the predictions against.
    c) Task 3:
    Continuing from Task 2, we then used evaluators for both of our models to check the accuracy of the predictions while using root-mean-square error (RMSE) as our metric of choice, and then used the println() functions to display the results back to the client. It should be noted that with 25 iterations, the code will take approximately 3-4 minutes to run before showing the results. As the RMSE for the linear regression turned out to be slightly less than the one for the gradient-boosting tree regressor, we can conclude that the linear regression ended up being the more effective of the 2 used for this task. Because the error was so large for both models, we can conclude that the Transtotal value does not

depend on the customer ID, TransID, Age, Salary, TransNumItems. However, it should be kept in mind that this is not necessarily fully accurate as the training and testing data for both of the models was randomly generated

Contribution Statement

We each completed the project separately and converged our ideas to submit the final version.

Resource Usage Statement

Axel

- The main resource I used for this assignment was https://spark.apache.org/ to learn all of the documentation that I needed for this project. Additional resources that also had good Spark code examples were https://sparkbyexamples.com/spark/how-to-create-a-sparksession-and-spark-context/ , https://medium.com/@thejasbabu/partial-partially-applied-functions-in-scala-a0d179e7df3, and https://www.scala-lang.org/api/2.12.1/scala/PartialFunction.html. I also used https://towardsdatascience.com/your-first-apache-spark-ml-model-d2bb82b599dd to get a better example of how to perform machine learning tasks using Scala. Additionally, I used ChatGPT to help me get the additional dependencies required for SparkSQL and MLlib, help me use dataframes to my advantage to build my programs that generated the data required for this project, and help me figure out how to properly build 1 file with all of the query from SparkSQL queries with the appropriate name, and in the appropriate directory on my computer.

Arjun

- I read ://spark.apache.org/docs/3.0.0/api/scala/org/apache/spark/index.html https://sparkbyexamples.com/ to understand more about the syntax, as I was new to Scala but the functional language paradigm is easy to understand, so there was not too much of a learning curve.
- I used ChatGPT to help fill any gaps in knowledge and understand more about the setup for these files.

Marc

- For the completion of this project assignment, I utilized the following resources to develop my own code to be later combined with my groupmates:

    1. Apache Spark Documentation: I extensively consulted the official Apache Spark documentation (https://spark.apache.org/docs/latest/) to understand Spark programming, Spark SQL, and MLlib functionalities.
    2. Scala Programming Documentation: I referred to the Scala programming language documentation (https://docs.scala-lang.org/) to grasp Scala syntax, features, and best practices used in Spark programming.
    3. Learning Spark Book: I utilized the book "Learning Spark" (O'Reilly) as a fundamental resource, providing insights into Spark RDDs, DataFrames, and Spark SQL.
    4. Online Educational Platforms: Additional learning resources from online educational platforms including those provided on canvas were used to gain a deeper understanding of specific Spark and Scala concepts.

5. Generative AI (ChatGPT): I employed Generative AI, specifically ChatGPT, during the MLlib section of the project, as it was the least familiar aspect for me. I used it to generate code snippets based on carefully crafted prompts.

- My strategy for validating the trustworthiness and validity of the generative AI solution involved:
  - Cross-Validation with Documentation: I cross-referenced generated code with official documentation to ensure alignment with established practices.
  - Iterative Prompt Refinement: I refined prompts iteratively based on output quality, guiding the model toward more accurate responses.
  - Peer Review: Code outputs underwent peer review, ensuring correctness and appropriateness.
  - Trial and Error: Multiple iterations and experimentation with prompts were conducted to achieve code outputs meeting project requirements.

Rishi

- Mostly used the resources on the assignment file and ChatGPT to better understand everything.