**CA675 - Assignment 1 - Spam Detection System**

Cloud Assignment 1

Name- Ananya Das

Student number- 23262592

Email id - [ananya.das5@mail.dcu.ie](mailto:ananya.das5@mail.dcu.ie)

**Task 3: Clean and process the data using pyspark**

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, regexp\_replace, length, when, count, udf

# Create a SparkSession

spark = SparkSession.builder.appName("DataCleaning").getOrCreate()

# Read the CSV file from Google Cloud Storage

df = spark.read.csv("gs://spam-detection-bucket/spam\_email\_dataset - spam\_email\_dataset.csv", header=True)

# Perform data cleaning steps

# 1. Email

# Remove leading and trailing whitespace

df = df.withColumn("Email", regexp\_replace("Email", "^\\s+|\\s+$", ""))

# Convert email to lowercase

df = df.withColumn("Email", col("Email"))

# Remove invalid characters

df = df.withColumn("Email", regexp\_replace("Email", "[^a-zA-Z0-9\\.\_@-]", ""))

# 2. Subject

# Remove leading and trailing whitespace

df = df.withColumn("Subject", regexp\_replace("Subject", "^\\s+|\\s+$", ""))

# Convert subject to lowercase

df = df.withColumn("Subject", col("Subject"))

# Remove HTML tags

df = df.withColumn("Subject", regexp\_replace("Subject", "<[^>]\*>", ""))

# Remove punctuation

df = df.withColumn("Subject", regexp\_replace("Subject", "[^a-zA-Z0-9\\s]", ""))

# 3. Sender

# Remove leading and trailing whitespace

df = df.withColumn("Sender", regexp\_replace("Sender", "^\\s+|\\s+$", ""))

# Convert sender to lowercase

df = df.withColumn("Sender", col("Sender"))

# Remove invalid characters

df = df.withColumn("Sender", regexp\_replace("Sender", "[^a-zA-Z0-9\\.\_@-]", ""))

# 4. Recipient

# Remove leading and trailing whitespace

df = df.withColumn("Recipient", regexp\_replace("Recipient", "^\\s+|\\s+$", ""))

# Convert recipient to lowercase

df = df.withColumn("Recipient", col("Recipient"))

# Remove invalid characters

df = df.withColumn("Recipient", regexp\_replace("Recipient", "[^a-zA-Z0-9\\.\_@-]", ""))

# 5. Date

# Validate date format(dd/mm/yyyy)

df = df.where("Date LIKE '\_\_/\_\_/\_\_\_\_'")

# 6. Time

# Validate time format (HH:MM)

df = df.where("Time LIKE '\_\_:\_\_'")

# 7. Attachments

# Convert attachments to lowercase

df = df.withColumn("Attachments", col("Attachments"))

# Remove invalid characters

df = df.withColumn("Attachments", regexp\_replace("Attachments", "[^a-zA-Z0-9\\.\_\\s]", ""))

# 8. Link Count

# Convert link count to integer

df = df.withColumn("Link Count", col("Link Count").cast("int"))

# Handle missing values

df = df.where(col("Link Count") >= 0)

# 9. Word Count

# Convert word count to integer

df = df.withColumn("Word Count", col("Word Count").cast("int"))

# Handle missing values

df = df.where(col("Word Count") >= 0)

# 10. Uppercase Count

# Convert uppercase count to integer

df = df.withColumn("Uppercase Count", col("Uppercase Count").cast("int"))

# Handle missing values

df = df.where(col("Word Count") >= 0)

# 11. Exclamation Count

# Convert exclamation count to integer

df = df.withColumn("Exclamation Count", col("Exclamation Count").cast("int"))

# Handle missing values

df = df.where(col("Exclamation Count") >= 0)

# 12. Question Count

# Convert question count to integer

df = df.withColumn("Question Count", col("Question Count").cast("int"))

# Handle missing values

df = df.where(col("Question Count") >= 0)

# 13. Dollar Count

# Convert dollar count to integer

df = df.withColumn("Dollar Count", col("Dollar Count").cast("int"))

# Handle missing values

df = df.where(col("Dollar Count") >= 0)

# 14. Punctuation Count

# Convert punctuation count to integer

df = df.withColumn("Punctuation Count", col("Punctuation Count").cast("int"))

# Handle missing values

df = df.where(col("Punctuation Count") >= 0)

# 15. HTML Tags Count

# Convert HTML tags count to integer

df = df.withColumn("HTML Tags Count", col("HTML Tags Count").cast("int"))

# Handle missing values

df = df.where(col("HTML Tags Count") >= 0)

# 16. SpamIndicator

# Convert spam indicator to boolean

df = df.withColumn("SpamIndicator", col("SpamIndicator").cast("boolean"))

# Write the cleaned data to Google Cloud Storage

df.write.csv("gs://spam-detection-bucket/cleaned-data.csv", header=True)

# Stop the SparkSession

spark.stop()

Let's break down the steps further:

1. SparkSession Creation:

- A `SparkSession` is the entry point for any Spark functionality. It provides a way to interact with Spark and allows the application to use Spark's distributed computing capabilities.

2. Read Data:

- The `spark.read.csv` method is used to read a CSV file into a Spark DataFrame (`df`). The data is assumed to have a header, which is specified with the `header=True` parameter.

3. Data Cleaning:

- Email, Subject, Sender, Recipient:

- Leading and trailing whitespaces are removed from these text fields.

- The text is converted to lowercase to ensure uniformity.

- Any invalid characters (specified by the regular expression) are removed.

- Date and Time:

- The `where` clause filters the data to only include rows where the date and time formats match the specified patterns.

- Attachments, Link Count, Word Count, Uppercase Count, Exclamation Count, Question Count, Dollar Count, Punctuation Count, HTML Tags Count:

- Numeric columns are cast to the appropriate data type (integer or boolean).

- Rows with negative values are filtered out (assuming these values should be non-negative).

- SpamIndicator:

- The `SpamIndicator` column is cast to a boolean type.

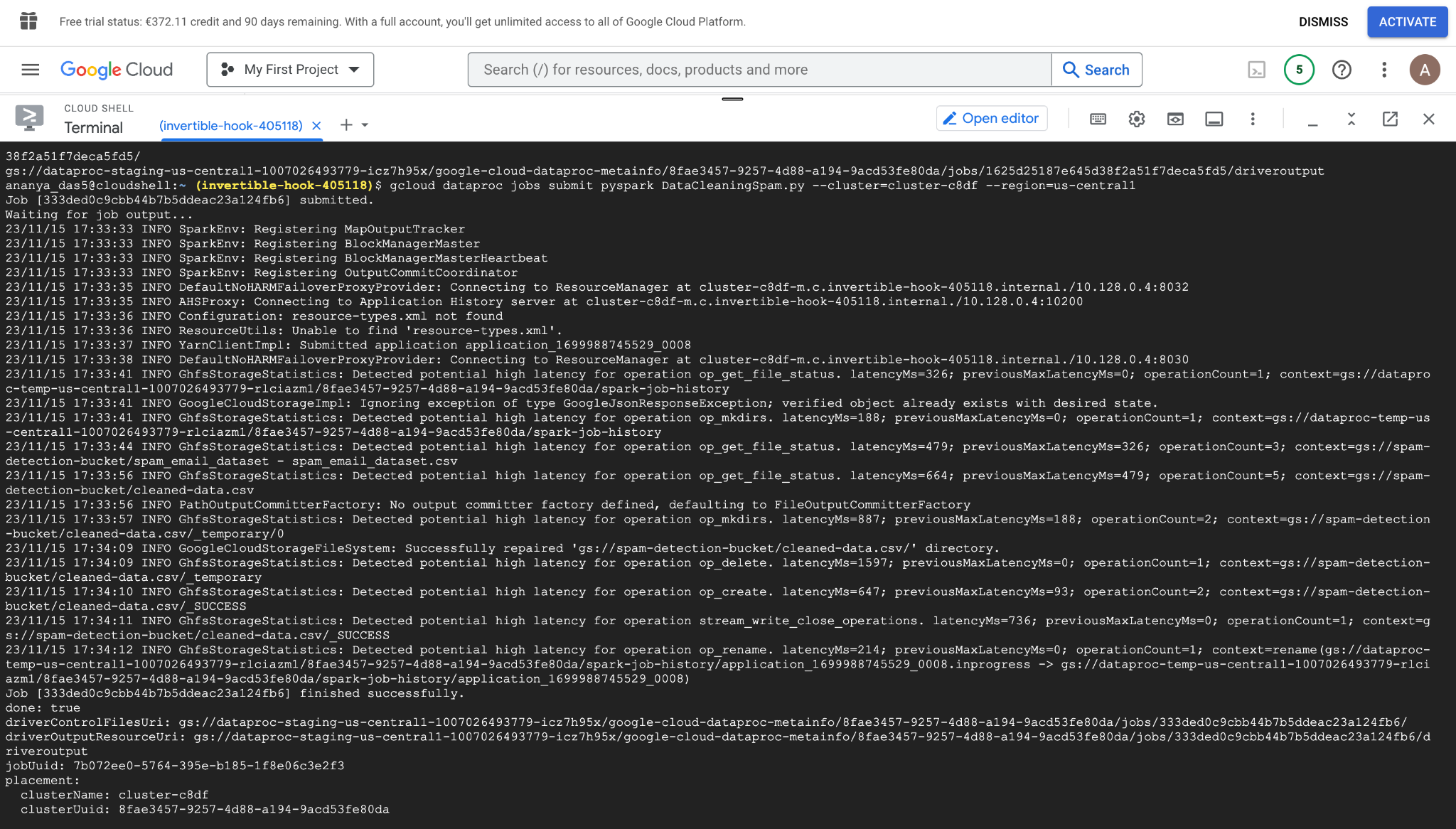
4. Write Cleaned Data:

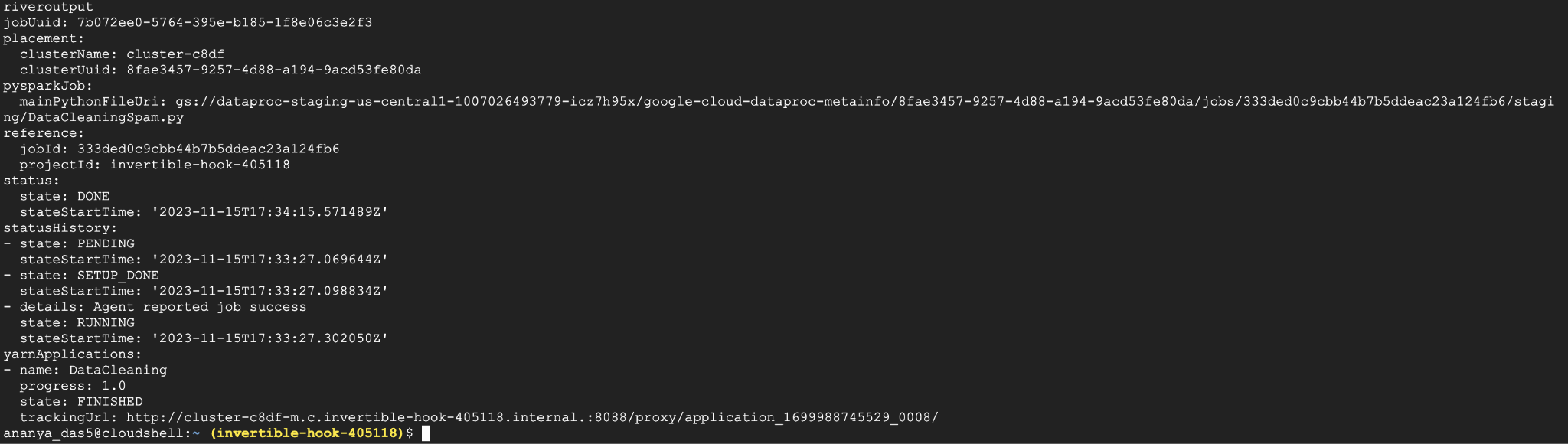
- The cleaned DataFrame (`df`) is written to a CSV file in Google Cloud Storage. The `header=True` parameter specifies that the CSV file should include a header with column names.

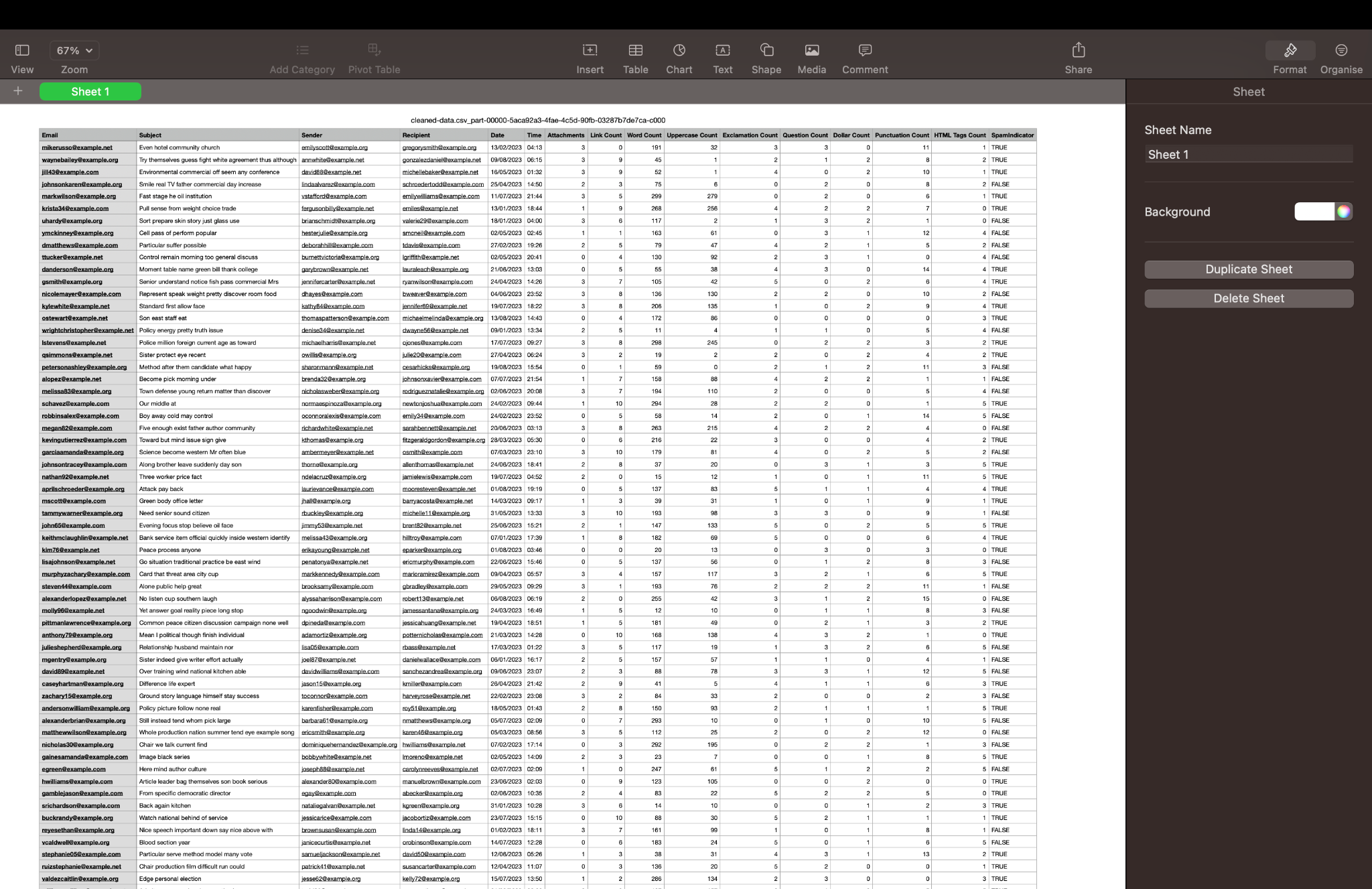
5. Stop SparkSession:

- The `spark.stop()` method is called to gracefully stop the SparkSession and release resources.

This script demonstrates a typical data cleaning pipeline using PySpark for distributed data processing. The cleaning steps are designed based on assumptions about the data and the desired output format. Adjustments should be made according to the specific characteristics of your dataset.







**Task 4: Top 10 Spam and Top 10 Ham mails**

from pyspark.sql import SparkSession

from pyspark.sql.functions import col

# Create a SparkSession

spark = SparkSession.builder.appName("EmailAnalysis").getOrCreate()

# Specify the path to your dataset in Google Drive

file\_path = 'gs://spam-detection-bucket/cleaned-data.csv/part-00000-5aca92a3-4fae-4c5d-90fb-03287b7de7ca-c000.csv'

# Load your dataset into a PySpark DataFrame

df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Identify the top 10 ham and spam emails based on the 'SpamIndicator' column

top\_10\_ham = df.filter(col('SpamIndicator') == 1).limit(10)

top\_10\_spam = df.filter(col('SpamIndicator') == 0).limit(10)

# Write the cleaned data to Google Cloud Storage

top\_10\_spam.write.csv("gs://spam-detection-bucket/Top10Spam.csv", header=True)

top\_10\_ham.write.csv("gs://spam-detection-bucket/Top10Ham.csv", header=True)

# Stop the SparkSession

spark.stop()

Let's delve into each step in more detail:

1. SparkSession Creation:

- A `SparkSession` is created using `SparkSession.builder.appName("EmailAnalysis").getOrCreate()`. This is the starting point for using Spark functionality in the application. The `appName` method sets the name of the application to "EmailAnalysis."

2. Specify Dataset Path:

- `file\_path` is a variable that holds the path to the cleaned dataset in Google Cloud Storage. It's essential to provide the correct path to access the data.

3. Load Dataset:

- The `spark.read.csv` method reads the CSV file from the specified path into a PySpark DataFrame named `df`.

- The `header=True` parameter indicates that the first row of the CSV file contains column names.

- The `inferSchema=True` parameter lets Spark infer the schema of the DataFrame based on the data.

4. Identify Top 10 Ham and Top 10 Spam Emails:

- Two DataFrames, `top\_10\_ham` and `top\_10\_spam`, are created by filtering the original DataFrame (`df`) based on the values in the 'SpamIndicator' column.

- The `col` function from `pyspark.sql.functions` is used to refer to DataFrame columns. `col('SpamIndicator') == 1` filters for ham (where 'SpamIndicator' is 1), and `col('SpamIndicator') == 0` filters for spam.

- The `limit(10)` method ensures that only the top 10 rows are selected from each DataFrame.

5. Write Results to Google Cloud Storage:

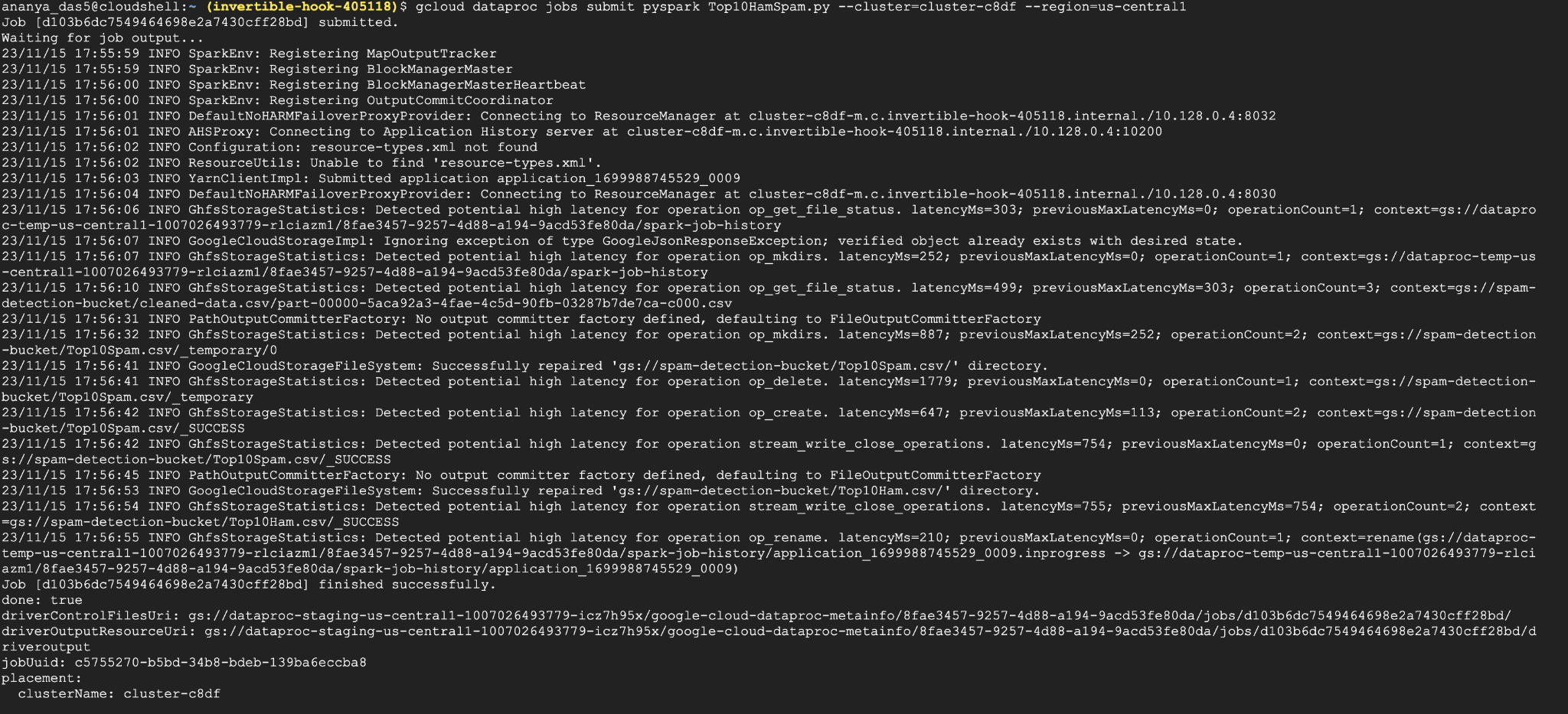
- The `write.csv` method is used to write the `top\_10\_spam` and `top\_10\_ham` DataFrames to separate CSV files in Google Cloud Storage.

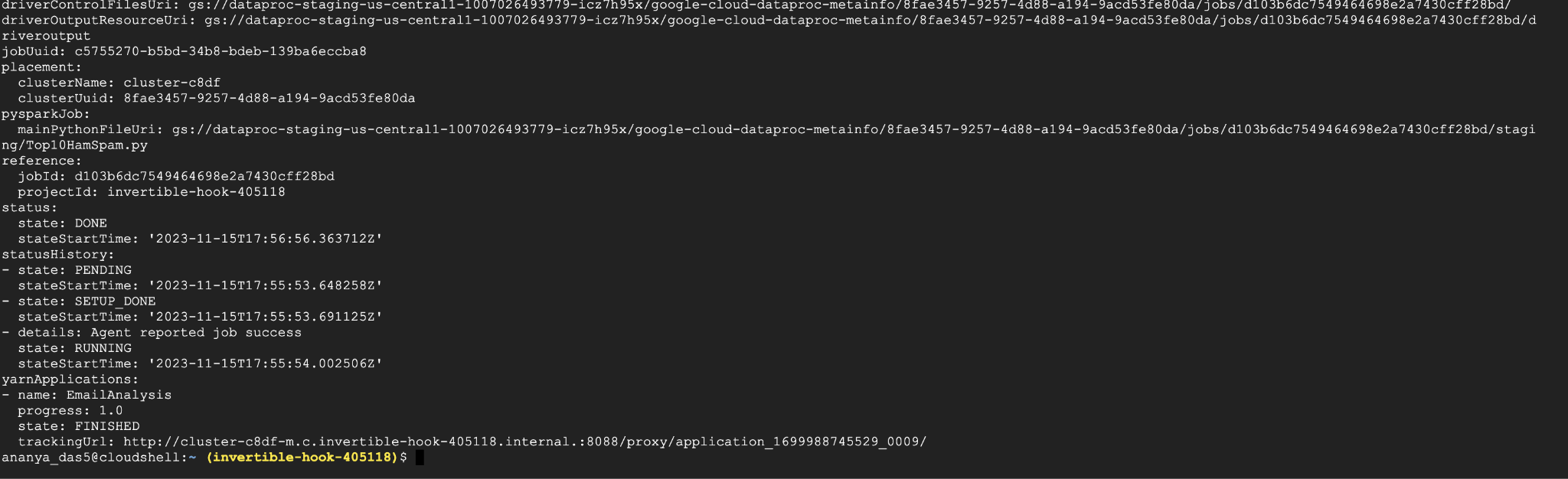
- The files are named "Top10Spam.csv" and "Top10Ham.csv," and the `header=True` parameter includes the column names in the CSV files.

6. Stop SparkSession:

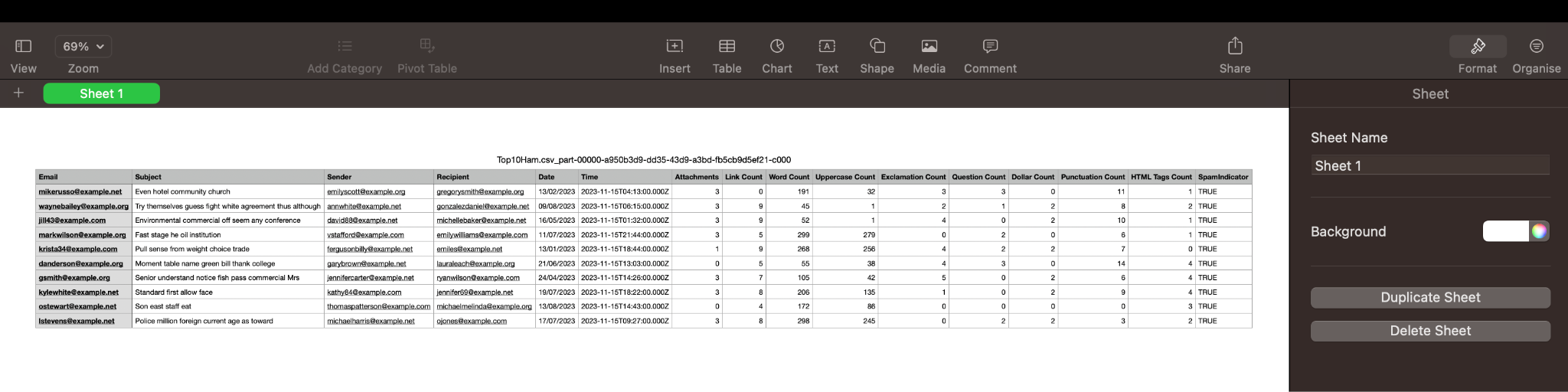
- The `spark.stop()` method is called to gracefully stop the SparkSession and release resources. This is important to avoid resource leaks.

This script is a part of an email analysis pipeline, demonstrating how to use Spark for data processing and analysis tasks. Adjustments can be made based on the specific requirements and characteristics of your dataset.

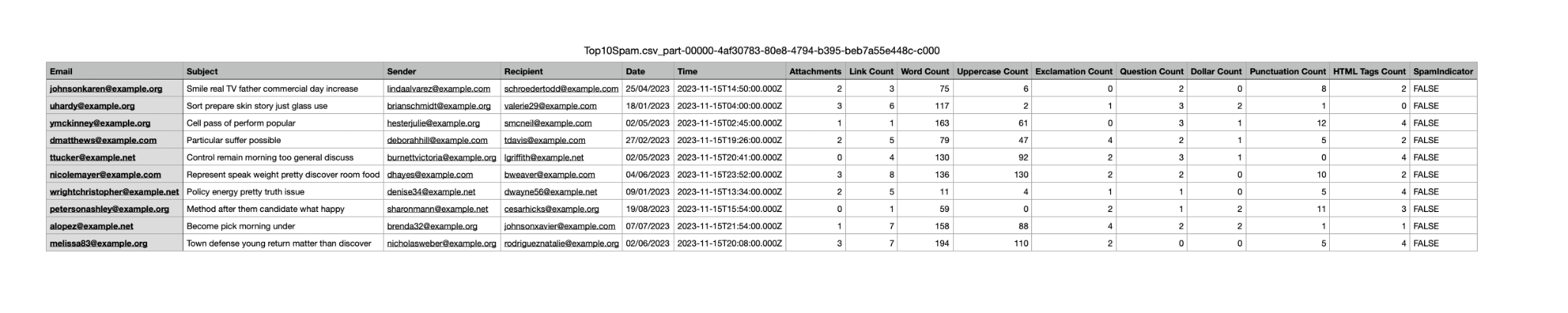




Top 10 ham



Top 10 Spam



**Task 5: TF-IDF using MapReduce**

from pyspark.sql import SparkSession

from pyspark.ml.feature import HashingTF, IDF, Tokenizer

from pyspark.sql.functions import col, udf

from pyspark.sql.types import StringType, ArrayType

# Create a Spark session

spark = SparkSession.builder.appName("TFIDFExample").getOrCreate()

# Load top 10 ham and spam DataFrames from separate CSV files

ham\_df = spark.read.csv("gs://spam-detection-bucket/Top10Ham.csv/part-00000-a950b3d9-dd35-43d9-a3bd-fb5cb9d5ef21-c000.csv", header=True, inferSchema=True) # Update the file path

spam\_df = spark.read.csv("gs://spam-detection-bucket/Top10Spam.csv/part-00000-4af30783-80e8-4794-b395-beb7a55e448c-c000.csv", header=True, inferSchema=True) # Update the file path

# Tokenize the emails for ham

ham\_tokenizer = Tokenizer(inputCol="Email", outputCol="words")

ham\_words\_data = ham\_tokenizer.transform(ham\_df)

# Tokenize the emails for spam

spam\_tokenizer = Tokenizer(inputCol="Email", outputCol="words")

spam\_words\_data = spam\_tokenizer.transform(spam\_df)

# Hashing Term Frequency (TF) for ham

ham\_hashing\_tf = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=100)

ham\_tf\_data = ham\_hashing\_tf.transform(ham\_words\_data)

# Hashing Term Frequency (TF) for spam

spam\_hashing\_tf = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=100)

spam\_tf\_data = spam\_hashing\_tf.transform(spam\_words\_data)

# Inverse Document Frequency (IDF) for ham

ham\_idf = IDF(inputCol="rawFeatures", outputCol="features")

ham\_idf\_model = ham\_idf.fit(ham\_tf\_data)

ham\_tfidf\_data = ham\_idf\_model.transform(ham\_tf\_data)

# Inverse Document Frequency (IDF) for spam

spam\_idf = IDF(inputCol="rawFeatures", outputCol="features")

spam\_idf\_model = spam\_idf.fit(spam\_tf\_data)

spam\_tfidf\_data = spam\_idf\_model.transform(spam\_tf\_data)

flatten\_udf\_spam = udf(lambda values: [float(value) for value in values] if values is not None else [], ArrayType(StringType()))

spam\_tfidf\_data = spam\_tfidf\_data.withColumn("features\_flat", flatten\_udf\_spam(col("features")))

flatten\_udf\_ham = udf(lambda values: [float(value) for value in values] if values is not None else [], ArrayType(StringType()))

ham\_tfidf\_data = ham\_tfidf\_data.withColumn("features\_flat", flatten\_udf\_ham(col("features")))

# Assuming df is your DataFrame with the 'words' column of type array<string>

# Convert the array of words to a single string

array\_to\_string\_udf = udf(lambda arr: ' '.join(arr), StringType())

flatten\_udf\_spam = udf(lambda values: ' '.join(map(str, values)), StringType())

# Fix for spam\_tfidf\_data

spam\_tfidf\_data = spam\_tfidf\_data.withColumn("words\_str", array\_to\_string\_udf(col("words")))

spam\_tfidf\_data = spam\_tfidf\_data.withColumn("features\_flat", flatten\_udf\_spam(col("features")))

spam\_tfidf\_data.select("words\_str", "features\_flat").write.csv("gs://spam-detection-bucket/TFIDFSpam", header=True, mode="overwrite")

# Fix for ham\_tfidf\_data

ham\_tfidf\_data = ham\_tfidf\_data.withColumn("words\_str", array\_to\_string\_udf(col("words")))

ham\_tfidf\_data = ham\_tfidf\_data.withColumn("features\_flat", flatten\_udf\_spam(col("features")))

ham\_tfidf\_data.select("words\_str", "features\_flat").write.csv("gs://spam-detection-bucket/TFIDFHam", header=True, mode="overwrite")

# Stop the Spark session

spark.stop()

! Let's delve into each step in more detail:

1. Create a Spark Session:

- Explanation Initiates a Spark session named "TFIDFExample." A Spark session is the entry point for working with Spark functionalities, and it provides a unified way to interact with Spark.

2. Load Top 10 Ham and Spam DataFrames:

- Explanation: Reads CSV files from Google Cloud Storage into PySpark DataFrames for further processing. This step is crucial for accessing and analysing the dataset.

3. Tokenize Emails for Ham and Spam:

- Explanation Utilises the `Tokenizer` class to split the 'Email' column into a list of words for both ham and spam DataFrames. Tokenization is the process of breaking down text into individual units, typically words or phrases.

4. Hashing Term Frequency (TF) for Ham and Spam:

- Explanation Applies the HashingTF algorithm to convert tokenized words into raw features (Term Frequency) for both ham and spam DataFrames. HashingTF is a method for turning a collection of text documents into numerical feature vectors.

5. Inverse Document Frequency (IDF) for Ham and Spam:

- Explanation Applies the IDF algorithm to raw features obtained from HashingTF to calculate the TF-IDF for both ham and spam DataFrames. IDF is used to measure the importance of a word in a document relative to a collection of documents.

6. Flatten Features for Ham and Spam:

- Explanation Defines a User-Defined Function (UDF) to flatten TF-IDF features from arrays to a list of float values for both ham and spam DataFrames. This step ensures that the TF-IDF features are in a format suitable for further analysis.

7. Convert Array of Words to Single String:

- Explanation Defines a UDF to convert the array of words to a single string for both ham and spam DataFrames. This transformation simplifies the representation of words for downstream processing.

8. Write TF-IDF Results to Google Cloud Storage:

- Explanation: Writes TF-IDF results for ham and spam DataFrames to separate CSV files in Google Cloud Storage (`TFIDFHam` and `TFIDFSpam`). Storing results in GCS makes them accessible and shareable.

9. Stop the Spark Session:

- Explanation Stops the SparkSession to release resources. This is a standard practice to ensure efficient resource utilisation.

This process involves a series of transformations and computations to derive TF-IDF features from the input data and store the results for further analysis.

