CS119 QUIZ 7 - Darcy Corson

PART I - RETAIL DATA ANALYSIS

1. [5 points] Loading the data into a Dataframe and removing junk records. How many records were removed by doing so?

   The code below was used to download, load, and clean the dataset of interest using PySpark.

```
# DOWNLOAD THE DATASET

import requests

url = "https://storage.googleapis.com/singhj-tufts-cs119/online-retail-II.csv"
response = requests.get(url)

# Open the file in binary write mode and write the contents of the response
with open("online-retail-II.csv", "wb") as file:
    file.write(response.content)

print("File downloaded successfully.")

# LOAD DATA INTO DATAFRAME

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, sum as _sum, max as _max, datediff, current_date, to_date,
desc, rank, date_add, lit
from pyspark.sql.window import Window

spark = SparkSession.builder.appName("RFM Analysis").getOrCreate()

# Load the dataset
df = spark.read.csv("online-retail-II.csv", header=True, inferSchema=True)

# Show the first few rows to confirm it's loaded correctly
df.show(5)

# CLEAN THE DATA

# Remove records with null Customer ID
df_clean = df.filter(df["Customer ID"].isNotNull())

# Convert InvoiceDate to a proper date format
from pyspark.sql.functions import to_date
df_clean = df_clean.withColumn("InvoiceDate", to_date(df_clean["InvoiceDate"], "MM/dd/yyyy"))

# Count and report the number of records removed
original_count = df.count()
clean_count = df_clean.count()
records_removed = original_count - clean_count
print(f"Records removed: {records_removed}")

# Report the number of clean records left
number_of_records = df_clean.count()
print(f"Number of records after cleaning: {number_of_records}")
```

   First, the dataset is downloaded from the specified URL using the requests library; the URL points to a CSV file named "online-retail-II.csv". The requests.get() function is used to retrieve the dataset, and the response content is saved to a local file named "online-retail-II.csv" in binary write mode. This step ensures that the dataset is locally available for processing.

Next, the dataset is loaded into a PySpark DataFrame using the spark.read.csv() method. This method reads the CSV file into a DataFrame, with options set to infer the schema and consider the first row as the header containing column names. The first few rows of the dataset are printed to allow for visual confirmation that it has loaded correctly.

```
+-------+---------+--------------------+--------+-------------------+-----+-----------+--------------+
|Invoice|StockCode|         Description|Quantity|        InvoiceDate|Price|Customer ID|       Country|
+-------+---------+--------------------+--------+-------------------+-----+-----------+--------------+
| 489434|    85048|15CM CHRISTMAS GL...|      12|2009-12-01 07:45:00| 6.95|    13085.0|United Kingdom|
| 489434|   79323P|  PINK CHERRY LIGHTS|      12|2009-12-01 07:45:00| 6.75|    13085.0|United Kingdom|
| 489434|   79323W| WHITE CHERRY LIGHTS|      12|2009-12-01 07:45:00| 6.75|    13085.0|United Kingdom|
| 489434|    22041|"RECORD FRAME 7""...|      48|2009-12-01 07:45:00|  2.1|    13085.0|United Kingdom|
| 489434|    21232|STRAWBERRY CERAMI...|      24|2009-12-01 07:45:00| 1.25|    13085.0|United Kingdom|
+-------+---------+--------------------+--------+-------------------+-----+-----------+--------------+
only showing top 5 rows
```

After the dataset is loaded, the data is cleaned. Specifically, null values in the "Customer ID" column are identified and filtered out using the df.filter() function. This results in a cleaned DataFrame without any records containing null values in the "Customer ID" column. The number of records removed due to null values in the "Customer ID" column is calculated and printed. Also, the total number of clean records remaining after filtering is determined and printed.

```
Records removed: 243007
Number of records after cleaning: 824364
```

2. [3 points] Calculation of monetary value, changing the column name to be monetary.

The code below calculates the monetary value of each customer's transactions based on the cleaned dataset (df_clean).

```
# CALCULATE MONETARY VALUE

monetary = df_clean.withColumn("TotalPrice", col("Quantity") * col("Price")) \
             .groupBy("Customer ID") \
             .agg(_sum("TotalPrice").alias("Monetary"))

# Display values
monetary.show()
```

First, a new column named "TotalPrice" in the DataFrame df_clean is created, and the column is populated by the result of multiplying the "Quantity" and "Price" columns, which represent the total price of each transaction. This operation is performed using the withColumn() function in PySpark, which adds a new column to the DataFrame based on the specified computation.

Then, the DataFrame is grouped by the "Customer ID" column using the groupBy() function to aggregate the total transaction amounts for each customer. The agg() function is then applied to compute the sum of the "TotalPrice" column for each customer group, with the result aliased as "Monetary". This aggregation operation consolidates all transaction amounts associated with each customer, providing a single monetary value representing their total spending.

Finally, the calculated monetary values are displayed using the show() function, which presents a summary of the DataFrame showing the "Customer ID" and their corresponding "Monetary" values. The first few rows of the dataset are printed for visual inspection.

```
+-----------+--------------------+
|Customer ID|            Monetary|
+-----------+--------------------+
|    17884.0|   3028.889999999997|
|    14285.0|  3158.6400000000003|
|    16822.0|  144.83999999999997|
|    16596.0|   579.6300000000001|
|    17072.0|              282.05|
|    12671.0|   2622.481000000001|
|    16981.0|            -4620.86|
|    14452.0|              665.59|
|    12737.0|              3710.5|
|    15893.0|  305.28000000000003|
|    14094.0|              334.27|
|    14269.0|  261.68000000000006|
|    12467.0|-2.1316282728030...|
|    16916.0|              1123.4|
|    13607.0|  1060.6099999999997|
|    14024.0|              645.74|
|    13094.0|             2214.66|
|    17633.0|  1974.8899999999996|
|    15846.0|  107.01000000000002|
|    16656.0|  16307.720000000008|
+-----------+--------------------+
only showing top 20 rows
```

The code below determines monetary scores for customers based on their total spending.

```
# CALCULATE SCORES FOR MONETARY

# Count the total number of customers (one row per customer)
total_customers = monetary.count()

# Calculate the customer counts for top 15%, top 30%, and top 60%
top_15_count = int(total_customers * 0.15)
top_30_count = int(total_customers * 0.30)
top_60_count = int(total_customers * 0.60)

# Rank customers by their total monetary value
windowSpec = Window.orderBy(desc("Monetary"))
monetary_ranked = monetary.withColumn("Rank", rank().over(windowSpec))

# Assign scores based on the specified thresholds
monetary_scored = monetary_ranked.withColumn("M_Score",
                         when(col("Rank") <= top_15_count, 1)
                         .when((col("Rank") > top_15_count) & (col("Rank") <= top_30_count), 2)
                         .when((col("Rank") > top_30_count) & (col("Rank") <= top_60_count), 3)
                         .otherwise(4))

# Display the Monetary values along with their assigned scores
monetary_scored.show()
```

First, it counts the total number of customers by counting the rows in the "monetary" DataFrame, assuming one row per customer. Then, it computes the customer counts corresponding to the top 15%, 30%, and 60% of customers. These counts are used to segment customers into different monetary score categories.

The code ranks customers based on their total monetary value. Customers are assigned ranks based on their total spending. Customers ranked within the top 15% receive a monetary score of 1, those ranked between the top 15% and top 30% receive a score of 2, and those between the top 30% and top 60% receive a score of 3. Customers outside these ranges are assigned a score of 4. A few monetary values along with their assigned scores are displayed for visual inspection.

```
+-----------+------------------+----+-------+
|Customer ID|          Monetary|Rank|M_Score|
+-----------+------------------+----+-------+
|    18102.0| 598215.2200000001|   1|      1|
|    14646.0| 523342.0700000004|   2|      1|
|    14156.0|296564.69000000024|   3|      1|
|    14911.0|   270248.529999999|   4|      1|
|    17450.0|233579.39000000013|   5|      1|
|    13694.0|190825.52000000016|   6|      1|
|    17511.0| 171885.9799999999|   7|      1|
|    12415.0|143269.28999999986|   8|      1|
|    16684.0|141502.24999999988|   9|      1|
|    15061.0|          136391.48|  10|      1|
|    15311.0|113513.06999999999|  11|      1|
|    13089.0|113214.19000000024|  12|      1|
|    17949.0| 98895.59000000003|  13|      1|
|    16029.0|           91800.91|  14|      1|
|    14298.0| 90489.31000000001|  15|      1|
|    15769.0|           84269.38|  16|      1|
|    13798.0| 73573.46999999991|  17|      1|
|    15838.0| 73404.10999999999|  18|      1|
|    12931.0|           71299.67|  19|      1|
|    17841.0| 69516.19000000061|  20|      1|
+-----------+------------------+----+-------+
only showing top 20 rows
```

3. [3 points] Calculation of frequency, changing the column name to be frequency.

The code below determines the frequency of transactions for each customer based on a cleaned dataset (df_clean).

```
# CALCULATE FREQUENCY

frequency = df_clean.groupBy("Customer ID").count().withColumnRenamed("count", "Frequency")

# Display values
frequency.show()
```

First, the DataFrame df_clean is grouped by the "Customer ID" column using the groupBy() function. This grouping operation organizes the dataset such that all transactions associated with each unique customer are grouped together.

Then, the count() function is applied to each group to calculate the number of transactions made by each customer, determining their transaction frequency. The result is a DataFrame that contains two columns: "Customer ID" and "count", where the "count" column represents the frequency of transactions for each customer. To provide clarity, the "count" column is renamed to "Frequency" using the withColumnRenamed() function.

Finally, a few of the calculated transaction frequencies are displayed using the show() function.

```
+-----------+---------+
|Customer ID|Frequency|
+-----------+---------+
|    17884.0|      489|
|    14285.0|       62|
|    16822.0|       13|
|    16596.0|       30|
|    17072.0|       22|
|    12671.0|       45|
|    16981.0|        1|
|    14452.0|      132|
|    12737.0|        2|
|    15893.0|        1|
|    14094.0|       48|
|    14269.0|       20|
|    12467.0|       18|
|    16916.0|      264|
|    13607.0|      123|
|    14024.0|       34|
|    13094.0|       38|
|    17633.0|      103|
|    15846.0|       29|
|    16656.0|      157|
+-----------+---------+
only showing top 20 rows
```

This code below calculates frequency scores for customers based on their transaction frequency. First, it determines the total number of customers by counting the rows in the "frequency" DataFrame. Then, it computes thresholds for the top 15%, 30%, and 60% of customers. These thresholds are used to segment customers into different frequency score categories.

```
# CALCULATE SCORES FOR FREQUENCY

# Calculate the number of customers (or total rows in 'frequency' DataFrame)
total_customers = frequency.count()

# Calculate thresholds for top 15%, 30%, and 60%
top_15_threshold = int(total_customers * 0.15)
top_30_threshold = int(total_customers * 0.30)
top_60_threshold = int(total_customers * 0.60)

# Rank customers based on Frequency
windowSpec = Window.orderBy(desc("Frequency"))
frequency = frequency.withColumn("Rank", rank().over(windowSpec))

# Assign Frequency scores based on calculated thresholds
frequency = frequency.withColumn("F_Score",
                    when(col("Rank") <= top_15_threshold, 1)
                    .when((col("Rank") > top_15_threshold) & (col("Rank") <= top_30_threshold), 2)
                    .when((col("Rank") > top_30_threshold) & (col("Rank") <= top_60_threshold), 3)
                    .otherwise(4))

# Display the adjusted Frequency scores
print (total_customers)
```

```
frequency.show()
```

Customers are assigned ranks based on the frequency of their transactions, and frequency scores are assigned based on customers' calculated ranks and predefined thresholds. Customers ranked within the top 15% receive a frequency score of 1, those ranked between the top 15% and top 30% receive a score of 2, and those between the top 30% and top 60% receive a score of 3. Customers outside these ranges are assigned a score of 4. Some adjusted frequency scores are displayed along with the total number of customers for visual inspection.

```
+-----------+---------+----+-------+
|Customer ID|Frequency|Rank|F_Score|
+-----------+---------+----+-------+
|    17841.0|    13097|   1|      1|
|    14911.0|    11613|   2|      1|
|    12748.0|     7307|   3|      1|
|    14606.0|     6709|   4|      1|
|    14096.0|     5128|   5|      1|
|    15311.0|     4717|   6|      1|
|    14156.0|     4130|   7|      1|
|    14646.0|     3890|   8|      1|
|    13089.0|     3438|   9|      1|
|    16549.0|     3255|  10|      1|
|    14298.0|     2868|  11|      1|
|    14527.0|     2837|  12|      1|
|    17850.0|     2827|  13|      1|
|    15039.0|     2810|  14|      1|
|    15005.0|     2548|  15|      1|
|    13081.0|     2430|  16|      1|
|    17511.0|     2134|  17|      1|
|    13263.0|     1920|  18|      1|
|    16782.0|     1900|  19|      1|
|    14159.0|     1885|  20|      1|
+-----------+---------+----+-------+
only showing top 20 rows
```

4. [5 points] Calculation of recency values, changing the name of the column accordingly.

The code below is responsible for calculating the recency of customer transactions.

```
# CALCULATE RECENCY

# Find the most recent invoice date in the dataset
most_recent_invoice_date =
df_clean.agg(_max("InvoiceDate").alias("MostRecent")).collect()[0]["MostRecent"]

# Set the fixed_date as one day after the most recent invoice date
fixed_date = date_add(lit(most_recent_invoice_date), 1)

# Calculate Recency
recency = df_clean.groupBy("Customer ID") \
            .agg(_max("InvoiceDate").alias("LastInvoiceDate")) \
            .withColumn("Recency", datediff(fixed_date, "LastInvoiceDate"))

# Display values to verify
recency.show()
```

First, it identifies the most recent invoice date in the dataset by using the agg function with _max to find the maximum date in the "InvoiceDate" column. This maximum date is stored in the variable most_recent_invoice_date. Then, it sets a fixed_date variable by adding one day to the most recent invoice date using the date_add function.

The code computes the recency for each customer by grouping the DataFrame df_clean by "Customer ID" and aggregating the maximum invoice date for each customer. It then calculates the difference in days between the fixed_date and the "LastInvoiceDate" using the datediff function. The resulting column is named "Recency". Finally, the recency.show() command displays the first few rows of the DataFrame for visual inspection.

```
+-----------+---------------+-------+
|Customer ID|LastInvoiceDate|Recency|
+-----------+---------------+-------+
|    17884.0|     2011-12-06|      4|
|    14285.0|     2011-11-18|     22|
|    16822.0|     2010-02-14|    664|
|    16596.0|     2011-11-24|     16|
|    17072.0|     2010-03-24|    626|
|    12671.0|     2010-04-12|    607|
|    16981.0|     2010-06-17|    541|
|    14452.0|     2011-11-29|     11|
|    12737.0|     2010-07-29|    499|
|    15893.0|     2010-08-15|    482|
|    14094.0|     2010-09-26|    440|
|    14269.0|     2010-09-30|    436|
|    12467.0|     2010-11-18|    387|
|    16916.0|     2011-11-16|     24|
|    13607.0|     2011-10-30|     41|
|    14024.0|     2011-08-10|    122|
|    13094.0|     2011-11-18|     22|
|    17633.0|     2011-11-08|     32|
|    15846.0|     2010-11-19|    386|
|    16656.0|     2011-11-17|     23|
+-----------+---------------+-------+
```

This code below computes the recency scores for each customer based on their transaction history.

```
# CALCULATE SCORES FOR RECENCY

# Define the date format for recency score calculation
date_format = "yyyy-MM-dd"

# Calculate Recency scores directly
recency_scores = recency.withColumn("R_Score",
                        when(col("LastInvoiceDate") > to_date(lit("2011-09-08"), date_format), 1)
                        .when((col("LastInvoiceDate") > to_date(lit("2011-06-07"), date_format)) &
                            (col("LastInvoiceDate") <= to_date(lit("2011-09-08"), date_format)), 2)
                        .when((col("LastInvoiceDate") > to_date(lit("2010-12-05"), date_format)) &
                            (col("LastInvoiceDate") <= to_date(lit("2011-06-07"), date_format)), 3)
                        .otherwise(4)
                        .alias("R_Score")
                      )

# Display recency scores
recency_scores.show()
```

First, it calculates the recency scores directly using a set of conditions applied to the "LastInvoiceDate" column. For each customer, the code evaluates the date of their last invoice against predefined date thresholds. Customers with the most recent invoices after September 8, 2011, are assigned a recency score of 1. Those with invoices dated between June 7, 2011, and September 8, 2011, receive a score of 2. For invoices dated between December 5, 2010, and June 7, 2011, the score is set to 3. Any customer with an invoice dated before December 5, 2010, is assigned a recency score of 4. The resulting recency scores are stored in a DataFrame column named "R_Score," which is displayed using the recency_scores.show() command for visual inspection.

```
+-----------+---------------+-------+-------+
|Customer ID|LastInvoiceDate|Recency|R_Score|
+-----------+---------------+-------+-------+
|    17884.0|     2011-12-06|      4|      1|
|    14285.0|     2011-11-18|     22|      1|
|    16822.0|     2010-02-14|    664|      4|
|    16596.0|     2011-11-24|     16|      1|
|    17072.0|     2010-03-24|    626|      4|
|    12671.0|     2010-04-12|    607|      4|
|    16981.0|     2010-06-17|    541|      4|
|    14452.0|     2011-11-29|     11|      1|
|    12737.0|     2010-07-29|    499|      4|
|    15893.0|     2010-08-15|    482|      4|
|    14094.0|     2010-09-26|    440|      4|
|    14269.0|     2010-09-30|    436|      4|
|    12467.0|     2010-11-18|    387|      4|
|    16916.0|     2011-11-16|     24|      1|
|    13607.0|     2011-10-30|     41|      1|
|    14024.0|     2011-08-10|    122|      2|
|    13094.0|     2011-11-18|     22|      1|
|    17633.0|     2011-11-08|     32|      1|
|    15846.0|     2010-11-19|    386|      4|
|    16656.0|     2011-11-17|     23|      1|
+-----------+---------------+-------+-------+
only showing top 20 rows
```

5. [4 points] Find the number of customers in each of the 6 categories in the table above.

The code below combines the RFM (Recency, Frequency, Monetary) scores into a single DataFrame called rfm_scores.

```
# JOIN RFM SCORES INTO SINGLE DATAFRAME

rfm_scores = monetary.join(frequency, "Customer ID") \
             .join(recency_scores.select("Customer ID", "R_Score"), "Customer ID") \
             .join(monetary_scored.select("Customer ID", "M_Score"), "Customer ID") \
             .select("Customer ID", "R_Score", "F_Score", "M_Score")

# Display values
rfm_scores.show()
```

The code performs a series of inner joins on the DataFrames containing the individual scores for recency, frequency, and monetary aspects. Initially, the monetary DataFrame is joined with the frequency DataFrame on the "Customer ID" column. Then, the resulting DataFrame from the previous join operation is joined with the recency_scores DataFrame, selecting only the "Customer ID" and "R_Score" columns. Then, the DataFrame is further joined with the monetary_scored DataFrame, again selecting only the "Customer ID" and "M_Score" columns. Finally, the resultant DataFrame is selected to include only the "Customer ID", "R_Score", "F_Score", and "M_Score" columns, forming the complete set of RFM scores. The .show() method is used to display some of the resulting DataFrame for visual inspection.

```
+-----------+-------+-------+-------+
|Customer ID|R_Score|F_Score|M_Score|
+-----------+-------+-------+-------+
|    17841.0|      1|      1|      1|
|    14911.0|      1|      1|      1|
|    12748.0|      1|      1|      1|
|    14606.0|      1|      1|      1|
|    14096.0|      1|      1|      1|
|    15311.0|      1|      1|      1|
|    14156.0|      1|      1|      1|
|    14646.0|      1|      1|      1|
|    13089.0|      1|      1|      1|
|    16549.0|      1|      1|      1|
|    14298.0|      1|      1|      1|
|    14527.0|      1|      1|      1|
|    17850.0|      3|      1|      1|
|    15039.0|      1|      1|      1|
|    15005.0|      1|      1|      1|
|    13081.0|      1|      1|      1|
|    17511.0|      1|      1|      1|
|    13263.0|      1|      1|      1|
|    16782.0|      1|      1|      1|
|    14159.0|      1|      1|      1|
+-----------+-------+-------+-------+
only showing top 20 rows
```

The code below classifies customers into segments based on their RFM (Recency, Frequency, Monetary) scores.

```
# CLASSIFY CUSTOMERS
# Assume that categorizations aren't intended to be mutually exclusive, allowing customers to belong to
multiple segments based on their RFM scores.

# Define a UDF to create an array of segments for each customer based on RFM scores
def determine_segments(r, f, m):
    segments = []
    if r == 1 and f == 1 and m == 1:
        segments.append("Best Customers")
    if f == 1:
        segments.append("Loyal Customers")
    if m == 1:
        segments.append("Big Spenders")
    if r == 3 and f == 1 and m == 1:
        segments.append("Almost Lost")
    if r == 4 and f == 1 and m == 1:
        segments.append("Lost Customers")
    if r == 4 and f == 4 and m == 4:
        segments.append("Lost Cheap Customers")
    return segments

concat_segments_udf = udf(determine_segments, ArrayType(StringType()))

# Apply the UDF to the DataFrame to create a new 'Segments' column
rfm_scores = rfm_scores.withColumn("Segments", concat_segments_udf("R_Score", "F_Score",
"M_Score"))

# Show the DataFrame to verify the new 'Segments' column
rfm_scores.show(truncate=False)
```

The code defines a user-defined function (UDF) called determine_segments, which takes the RFM scores (R_Score, F_Score, M_Score) as input and assigns customers to various segments based on certain conditions. These conditions include whether a customer is a "Best Customer," "Loyal Customer," "Big Spender," "Almost Lost," "Lost Customer," or "Lost Cheap Customer," depending on their RFM scores.

Once the UDF is defined, it is registered with Spark and applied to the DataFrame containing the RFM scores, rfm_scores, to create a new column called "Segments." This column stores an array of segment labels for each customer based on their RFM scores. Finally, some of the DataFrame is displayed to verify the addition of the "Segments" column.

Note that it is assumed that categorizations aren't intended to be mutually exclusive, which allows for customers to belong to multiple segments at the same time based on their RFM scores. Also, note that some customers will belong to no segments.

```
+-----------+-------+-------+-------+--------------------------------------------+
|Customer ID|R_Score|F_Score|M_Score|Segments                                    |
+-----------+-------+-------+-------+--------------------------------------------+
|17841.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|14911.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|12748.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|14606.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|14096.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|15311.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|14156.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|14646.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|13089.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|16549.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|14298.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|14527.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|17850.0    |3      |1      |1      |[Loyal Customers, Big Spenders, Almost Lost]   |
|15039.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|15005.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|13081.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|17511.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|13263.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|16782.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
|14159.0    |1      |1      |1      |[Best Customers, Loyal Customers, Big Spenders]|
+-----------+-------+-------+-------+--------------------------------------------+
only showing top 20 rows
```

The code below breaks down the segments assigned to each customer into individual rows, allowing for a more granular analysis of customer distribution across segments.

```
# PROVIDE CUSTOMER COUNTS

from pyspark.sql.functions import explode

# Explode the 'Segments' array into individual rows for each segment per customer
segments_exploded = rfm_scores.select(
    col("Customer ID"),
    explode(col("Segments")).alias("Segment")
)

# Count the number of customers in each segment
category_counts = segments_exploded.groupBy("Segment").count()

# Show the count of customers in each category
category_counts.show()
```

First, it uses the explode function to transform the array of segments in the DataFrame rfm_scores into separate rows for each segment associated with a customer. This process generates a new DataFrame called segments_exploded, where each row contains a single segment associated with a specific customer.

Next, the code groups the exploded segments by segment name using the groupBy function and calculates the count of customers within each segment. This aggregation is stored in a DataFrame named category_counts, where each row represents a segment and its corresponding count of customers.

Finally, the code displays the count of customers in each segment by using the show method on the category_counts DataFrame.

```
+--------------------+-----+
|             Segment|count|
+--------------------+-----+
|Lost Cheap Customers|  962|
|     Loyal Customers|  895|
|      Best Customers|  570|
|         Almost Lost|   20|
|        Big Spenders|  891|
|      Lost Customers|    6|
+--------------------+-----+
```

6. [5 points] How would you recommend that the loyal customers, (RFM = X1X), be further segmented? Please justify your answer.

   To further segment the loyal customers (RFM = X1X), it would be interesting to delve deeper into that group's behavior and characteristics to identify subgroups with distinct preferences and needs. Subsequently, the segmentation strategies described below might help businesses gain deeper understanding of their loyal customer base and tailor their marketing efforts and customer experiences more effectively to drive satisfaction, loyalty, and long-term value.

   First, the purchase patterns of loyal customers could be analyzed to identify specific product categories or types that they prefer. For example, some loyal customers might primarily purchase high-value items, while others might focus on frequent purchases of everyday essentials. Segmenting loyal customers based on their preferred product categories could help tailor marketing strategies and product offerings more effectively.

   Also, the lifetime value of loyal customers could also be assessed by considering not only their current spending but their potential future value. Customers with high lifetime value may warrant special treatment or exclusive offers to nurture their loyalty over the long term. Segmenting loyal customers based on their predicted lifetime value could help to prioritize resources and tailor retention strategies accordingly.

   Further, demographic or psychographic factors that may influence the behavior of loyal customers (e.g. age, gender, income level, lifestyle) could be evaluated. Segmenting loyal customers based on these factors could uncover valuable insights into their motivations and preferences, facilitating more targeted marketing campaigns and personalized customer experiences.

.
Finally, the way that loyal customers interact with the brand/store across different channels, including offline and online touchpoints, could also be examined. Segmenting loyal customers based on their channel preferences could help optimize omnichannel strategies and deliver seamless experiences across all touchpoints.

PART II - SHORT STORIES ANALYSIS FRAMEWORK

a. [2 points] Clean the text and remove stopwords.

This code below cleans and removes stopwords from (and tokenizes) the text of various stories by Edgar Allan Poe.

```
# CLEAN TEXTS AND REMOVE STOPWORDS

import requests
import re
import nltk
nltk.download('punkt')

# Function to clean text and remove stopwords
def clean_and_tokenize(text, stopwords):
    text = text.replace('-', ' ').replace('—', ' ')
    text = re.sub(r'[^\w\s]', '', text)
    words = nltk.word_tokenize(text)
    words = [word.lower() for word in words if word.lower() not in stopwords and word.isalpha()]
    return words

# Get list of stopwords
stopwords_url =
"https://gist.githubusercontent.com/rg089/35e00abf8941d72d419224cfd5b5925d/raw/12d899b70156fd0041f
a9778d657330b024b959c/stopwords.txt"
stopwords_list = requests.get(stopwords_url).content.decode('utf-8').splitlines()
stopwords = set(stopwords_list)

# Story titles
story_titles = [
    "A_DESCENT_INTO_THE_MAELSTROM",
    "BERENICE",
    "ELEONORA",
    "LANDORS_COTTAGE",
    "MESMERIC_REVELATION",
    "SILENCE-A_FABLE",
    "THE_ASSIGNATION",
    "THE_BLACK_CAT",
    "THE_CASK_OF_AMONTILLADO",
    "THE_DOMAIN_OF_ARNHEIM",
    "THE_FACTS_IN_THE_CASE_OF_M._VALDEMAR",
    "THE_FALL_OF_THE_HOUSE_OF_USHER",
    "THE_IMP_OF_THE_PERVERSE",
    "THE_ISLAND_OF_THE_FAY",
    "THE_MASQUE_OF_THE_RED_DEATH",
    "THE_PIT_AND_THE_PENDULUM",
    "THE_PREMATURE_BURIAL",
    "THE_PURLOINED_LETTER",
    "THE_THOUSAND-AND-SECOND_TALE_OF_SCHEHERAZADE",
```

CS119 QUIZ 7 - Darcy Corson

```
    "VON_KEMPELEN_AND_HIS_DISCOVERY",
    "WILLIAM_WILSON"
]

# Base URL for the text files in the GitHub repository
base_url = "https://raw.githubusercontent.com/singhj/big-data-repo/main/text-proc/poe-stories/"

# Process each story
for title in story_titles:
    file_url = base_url + title
    response = requests.get(file_url)
    if response.status_code == 200:
        cleaned_words = clean_and_tokenize(response.text, stopwords)
        cleaned_text = ' '.join(cleaned_words)
        print(f"First 100 words from '{title}': {' '.join(cleaned_text.split()[:100])}")
    else:
        print(f"Failed to get the story for '{title}'. HTTP Status Code: {response.status_code}")
```

First, it defines a function, clean_and_tokenize, which replaces hyphens and em dashes with spaces to prevent merging words together when punctuation is removed. The function also strips the text of all non-word characters except for spaces using a regular expression. Further, it tokenizes the cleaned text into individual words with the NLTK library, converts them to lowercase, filters out any stopwords and non-alphabetic tokens, and returns the list of cleaned tokens. Finally, it prints the first 100 cleaned, non-stopword words from each story for visual inspection. Note that the screenshot below has been cropped.

```
First 100 words from 'A_DESCENT_INTO_THE_MAELSTROM': ways god nature providence ways models frame commensurate vastness profundity unsearchableness works
First 100 words from 'BERENICE': dicebant mihi sodales sepulchrum amicae visitarem curas meas aliquar tulum fore levatas misery manifold wretchedness earth
First 100 words from 'ELEONORA': conservatione formæ specificæ salva anima race vigor fancy ardor passion men called mad question settled madness loftiest
First 100 words from 'LANDORS_COTTAGE': pendant domain arnheim pedestrian trip summer river counties york day declined embarrassed road pursuing land
First 100 words from 'MESMERIC_REVELATION': doubt envelop mesmerism startling universally admitted doubt mere doubters profession unprofitable disreputable
First 100 words from 'SILENCE-A_FABLE': mountain pinnacles slumber valleys crags caves listen demon hand head region speak dreary region libya borders river
First 100 words from 'THE_ASSIGNATION': stay fail meet thee hollow vale death wife henry king bishop ill fated mysterious man bewildered brilliancy thine
First 100 words from 'THE_BLACK_CAT': wild homely narrative pen expect solicit belief mad expect case senses reject evidence mad surely dream morrow die day
First 100 words from 'THE_CASK_OF_AMONTILLADO': injuries fortunato borne ventured insult vowed revenge nature soul suppose utterance threat avenged point
First 100 words from 'THE_DOMAIN_OF_ARNHEIM': garden lady fair cut lay slumbered delight open skies eyes shut azure fields heaven sembled large round set
First 100 words from 'THE_FACTS_IN_THE_CASE_OF_M._VALDEMAR': pretend matter extraordinary case valdemar excited discussion miracle circumstances desire
First 100 words from 'THE_FALL_OF_THE_HOUSE_OF_USHER': son cœur luth suspendu sitôt quon touche résonne dull dark soundless day autumn year clouds hung
First 100 words from 'THE_IMP_OF_THE_PERVERSE': consideration faculties impulses prima mobilia human soul phrenologists failed room propensity existing
First 100 words from 'THE_ISLAND_OF_THE_FAY': nullus enim locus sine genio musique marmontel contes moraux translations insisted calling moral tales mockery
First 100 words from 'THE_MASQUE_OF_THE_RED_DEATH': red death long devastated country pestilence fatal hideous blood avatar seal redness horror blood sharp
First 100 words from 'THE_PIT_AND_THE_PENDULUM': impia tortorum longos hic turba furores sanguinis innocui satiata aluit sospite nunc patria fracto nunc
First 100 words from 'THE_PREMATURE_BURIAL': themes absorbing horrible purposes legitimate fiction mere romanticist eschew offend disgust propriety handled
First 100 words from 'THE_PURLOINED_LETTER': purloined letter nil sapientiæ odiosius acumine nimio paris dark gusty evening autumn enjoying twofold luxury
First 100 words from 'THE_THOUSAND-AND-SECOND_TALE_OF_SCHEHERAZADE': truth stranger fiction occasion oriental investigations consult tellmenow isitsöornot
First 100 words from 'VON_KEMPELEN_AND_HIS_DISCOVERY': minute elaborate paper arago summary sillimans journal detailed statement published lieutenant maury
First 100 words from 'WILLIAM_WILSON': conscience grim spectre path william wilson fair lying sullied real appellation object scorn horror detestation race
```

b. [5 points] Use NLTK to decompose the first story (A_DESCENT_INTO…) into sentences & sentences into tokens.

c. [5 points] Tag all remaining words in the story as parts of speech using the Penn POS Tags. Create and print a dictionary with the Penn POS Tags as keys and a list of words as the values.

This script below processes "A Descent into the Maelström" by Edgar Allan Poe, available in a GitHub repository. It employs the Natural Language Toolkit (NLTK) in Python for natural language processing tasks.

```
# TAG AND CLASSIFY CLEANED WORDS; BUILD DICTIONARY

from nltk.tokenize import sent_tokenize, word_tokenize
from collections import defaultdict
```

```
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Get the story text
title = "A_DESCENT_INTO_THE_MAELSTROM"
base_url = "https://raw.githubusercontent.com/singhj/big-data-repo/main/text-proc/poe-stories/"
file_url = base_url + title
response = requests.get(file_url)

if response.status_code != 200:
    print(f"Failed to fetch the story for '{title}'. HTTP Status Code: {response.status_code}")
else:

    stopwords_url =
"https://gist.githubusercontent.com/rg089/35e00abf8941d72d419224cfd5b5925d/raw/12d899b70156fd0041fa9778d657330b024b959c/stopwords.txt"
    stopwords_list = requests.get(stopwords_url).content.decode('utf-8').splitlines()
    stopwords = set(stopwords_list)

    cleaned_tokens = clean_and_tokenize(response.text, stopwords)

    all_tagged = nltk.pos_tag(cleaned_tokens)

    # Create a dictionary
    pos_dict = defaultdict(list)
    for word, tag in all_tagged:
        pos_dict[tag].append(word)

    # Print dictionary
    for pos_tag, words in pos_dict.items():
        print(f"{pos_tag}: {sorted(set(words))}")
```

First, the code cleans and tokenizes the story's text. The clean_and_tokenize function is responsible for removing punctuation, making the text lowercase, removing stopwords, and splitting the text into individual tokens or words. After cleaning and tokenization, the script tags each token with its corresponding part of speech using the NLTK's pos_tag function. Then, it constructs a dictionary where each part of speech is a key, and the associated value is a list of words tagged with that part of speech. Finally, for each part of speech in the dictionary, the script prints the POS tag followed by the alphabetically sorted and deduplicated list of words corresponding to that tag. Note that the screenshot below has been cropped.

```
NNS: ['accounts', 'anecdotes', 'annoyances', 'articles', 'attempts', 'barrels', 'barren', 'bears', 'bellowings', 'boats', 'bodies', 'bolt', '
VBP: ['abyss', 'account', 'apprehend', 'awe', 'beholder', 'boat', 'brink', 'burst', 'burthen', 'calmest', 'cataract', 'channel', 'coast', 'co
JJ: ['absolute', 'afford', 'agony', 'amazing', 'ambaaren', 'american', 'angry', 'anxious', 'apparent', 'arose', 'arrival', 'astern', 'attempt
NN: ['aback', 'absurd', 'abundance', 'abyss', 'accident', 'account', 'action', 'admiration', 'advantage', 'afternoon', 'agitation', 'agony',
VBZ: ['appears', 'assumes', 'boys', 'dark', 'decreases', 'ends', 'hairs', 'heavens', 'leagues', 'length', 'masses', 'passages', 'precipitates
RB: ['abyss', 'accurately', 'afterward', 'ago', 'ahead', 'altogether', 'appearance', 'beneath', 'bodily', 'borne', 'bound', 'brightly', 'brot
JJR: ['counter', 'cylinder', 'deeper', 'greater', 'higher', 'kircher', 'larger', 'lower', 'restore', 'smaller', 'weightier', 'yonder']
VBN: ['absorbed', 'acquired', 'agreed', 'approached', 'ascended', 'assumed', 'batten', 'beheld', 'blazed', 'broken', 'buried', 'called', 'car
JJS: ['crest', 'divest', 'eldest', 'faintest', 'finest', 'greatest', 'highest', 'honest', 'keenest', 'largest', 'lightest', 'loftiest', 'loud
VBD: ['absorbed', 'admitted', 'allowed', 'appeared', 'arose', 'ascertained', 'assented', 'attached', 'attracted', 'beat', 'becalmed', 'began',
IN: ['abyss', 'amid', 'boat', 'britannica', 'broken', 'drove', 'otterholm', 'overcast', 'round', 'teeth', 'thereabout', 'thrown', 'tide', 've
RBR: ['cylinder', 'explore', 'farther', 'feather', 'higher', 'limbs', 'longer', 'matter', 'shadow']
VBG: ['answering', 'appalling', 'approaching', 'ascertaining', 'attempting', 'bearing', 'beetling', 'bewildering', 'blowing', 'boasting', 'bo
FW: ['elbow']
VB: ['channel', 'deck', 'elder', 'hold', 'keel', 'morrow', 'raise', 'shake', 'slack', 'slow', 'timid']
```

d. [8 points] In this framework, each row will represent a story. The columns will be as follows: the text of the story, two-letter prefixes of each tag, for example NN, VB, RB, JJ etc., and the words belonging to that tag in the story. Show your code and the tag columns, at least for the one story.

This code below implements a text analysis framework using PySpark and NLTK to process a collection of Edgar Allan Poe's stories, focusing on tokenization and part-of-speech (POS) tagging. Note that, due to insufficient JAVA heap space, this code does not add to the DataFrame the column that shows the entire original text of each story.

```
# CREATE DATAFRAME
# Does not contain column that shows the entire original text of each story due to insufficient JAVA heap space.

import re
import requests
import nltk
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, explode, col, collect_set
from pyspark.sql.types import StringType, ArrayType, StructType, StructField

# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Initialize Spark Session
spark = SparkSession.builder \
    .appName("Poe Stories Analysis") \
    .master("local[*]") \
    .getOrCreate()

# Fetch stopwords
stopwords_url =
"https://gist.githubusercontent.com/rg089/35e00abf8941d72d419224cfd5b5925d/raw/12d899b70156fd0041f
a9778d657330b024b959c/stopwords.txt"
stopwords = set(requests.get(stopwords_url).text.lower().split())

# Clean and tokenize text
def clean_and_tokenize(text):
    sentences = nltk.sent_tokenize(text)
    cleaned_tokens = []
    for sentence in sentences:
        tokens = nltk.word_tokenize(sentence)
        for token in tokens:
            if token.lower() not in stopwords and token.isalpha():
                if token == tokens[0] or token.isupper():
                    cleaned_tokens.append(token)
                else:
                    cleaned_tokens.append(token.lower())
    return cleaned_tokens
```

CS119 QUIZ 7 - Darcy Corson

```
clean_tokenize_udf = udf(clean_and_tokenize, ArrayType(StringType()))

# POS tagging
def pos_tag(tokens):
    return nltk.pos_tag(tokens)

pos_tag_udf = udf(pos_tag, ArrayType(StructType([
    StructField("word", StringType(), False),
    StructField("tag", StringType(), False)
])))

story_titles = [
    "A_DESCENT_INTO_THE_MAELSTROM",
    "BERENICE",
    "ELEONORA",
    "LANDORS_COTTAGE",
    "MESMERIC_REVELATION",
    "SILENCE-A_FABLE",
    "THE_ASSIGNATION",
    "THE_BLACK_CAT",
    "THE_CASK_OF_AMONTILLADO",
    "THE_DOMAIN_OF_ARNHEIM",
    "THE_FACTS_IN_THE_CASE_OF_M._VALDEMAR",
    "THE_FALL_OF_THE_HOUSE_OF_USHER",
    "THE_IMP_OF_THE_PERVERSE",
    "THE_ISLAND_OF_THE_FAY",
    "THE_MASQUE_OF_THE_RED_DEATH",
    "THE_PIT_AND_THE_PENDULUM",
    "THE_PREMATURE_BURIAL",
    "THE_PURLOINED_LETTER",
    "THE_THOUSAND-AND-SECOND_TALE_OF_SCHEHERAZADE",
    "VON_KEMPELEN_AND_HIS_DISCOVERY",
    "WILLIAM_WILSON"
]

# Base URL for the text files
base_url = "https://raw.githubusercontent.com/singhj/big-data-repo/main/text-proc/poe-stories/"

# Process each story and collect results
stories_data = []
for title in story_titles:
    file_url = base_url + title
    response = requests.get(file_url)
    if response.status_code == 200:
        stories_data.append((title, response.text))
    else:
        print(f"Failed to fetch '{title}'. HTTP Status: {response.status_code}")

# Create DataFrame from stories data
if stories_data:
    df = spark.createDataFrame(stories_data, ["Title", "Story_Text"])
    df = df.withColumn("Tokens", clean_tokenize_udf("Story_Text"))
    df = df.withColumn("POS_Tags", pos_tag_udf("Tokens"))
```

CS119 QUIZ 7 - Darcy Corson

```
    df_exploded = df.select("Title", explode("POS_Tags").alias("POS"))
    df_exploded = df_exploded.select("Title", col("POS.word").alias("Word"), col("POS.tag").alias("Tag"))

    pos_grouped = df_exploded.groupBy("Title").pivot("Tag").agg(collect_set("Word"))
    pos_grouped.show(truncate=False)

spark.stop()
```

First, a Spark session is established. Methods are declared to clean and tokenize the text, as well as to perform POS tagging. The clean_and_tokenize function breaks down the text into sentences, tokenizes these sentences into words, and filters out stopwords and non-alphabetical tokens. It also ensures that proper nouns retain their capitalization. The pos_tag function applies NLTK's POS tagging to the list of cleaned tokens. Then, the code retrieves the texts of various stories by Edgar Allan Poe, processes each story if successfully fetched, and stores the results in a Spark DataFrame. Each story's data includes its title and the processed tokens. This DataFrame is subsequently manipulated to explode the POS tags into individual entries, facilitating the grouping of data by story title and the aggregation of words under each POS tag using the collect_set function to avoid duplicates. The resulting structured table is displayed with the DataFrame's show method, and it categorizes the words by their POS tags for each story title. Note that the screenshot pasted below has been cropped.

Title	CC	CD	DT	EX	FW	IN
SILENCE-A_FABLE	[]	[]	[behemoth]	[]	[murmur]	[wind,
A_DESCENT_INTO_THE_MAELSTROM	[]	[]	[]	[]	[elbow]	[wind,
BERENICE	[]	[]	[]	[]	[mademoiselle, salle]	[aloud,
THE_CASK_OF_AMONTILLADO	[loth, nullus, epoch]	[lapse, husband]	[recall, befall, nether]	[mere]	[veil, eleonora, mere, lucid, valley, enwrapt, mademoiselle, salle, marchesa, quelqu]	[unboun
ELEONORA	[epoch]	[]	[befall]	[]	[eleonora, lucid]	[devout
THE_DOMAIN_OF_ARNHEIM	[]	[]	[]	[]	[veil, mere, enwrapt]	[afford
THE_BLACK_CAT	[]	[]	[]	[mere]	[]	[tender
THE_ASSIGNATION	[]	[]	[]	[]	[marchesa]	[apollo
LANDORS_COTTAGE	[loth]	[]	[]	[]	[]	[overgr
MESMERIC_REVELATION	[ether]	[]	[ether]	[]	[vankirk]	[atom,
THE_PREMATURE_BURIAL	[]	[lapse, husband]	[]	[]	[mademoiselle]	[amid,
THE_IMP_OF_THE_PERVERSE	[]	[]	[]	[]	[]	[aloud,
THE_PIT_AND_THE_PENDULUM	[]	[]	[nether]	[]	[]	[unboun
THE_FALL_OF_THE_HOUSE_OF_USHER	[]	[zigzag]	[]	[]	[vivid]	[wind,
VON_KEMPELEN_AND_HIS_DISCOVERY	[]	[kempelen, molten]	[]	[]	[kissam, parcel, viele]	[arago,
THE_FACTS_IN_THE_CASE_OF_M._VALDEMAR	[]	[]	[half]	[]	[skin]	[profus
WILLIAM_WILSON	[]	[]	[recall]	[]	[valley]	[prove,
THE_ISLAND_OF_THE_FAY	[Nullus]	[]	[]	[]	[quelqu]	[blades
THE_MASQUE_OF_THE_RED_DEATH	[]	[]	[]	[]	[lapse]	[assume
THE_PURLOINED_LETTER	[]	[knew]	[]	[]	[knob, vis]	[truth,

only showing top 20 rows

This code below implements a text analysis framework using PySpark and NLTK to process Edgar Allan Poe's "A Descent into the Maelström", focusing on tokenization and part-of-speech (POS) tagging. Note that this code does add to the DataFrame the column that shows the entire original text of the story. This is done to demonstrate that such functionality was successfully implemented as per the requirements of this assignment. Thus, for this specific story from Poe, the code fetches the text using an HTTP request and the story text is loaded into a Spark DataFrame. The rest of this code's functionality is the same as explained above.

```
# COMPLETE DATAFRAME FOR FIRST STORY
# Assume that the first column should contain the entire original text of the story, not the cleaned text.

from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, explode, col, collect_set
from pyspark.sql.types import StringType, ArrayType, StructType, StructField
import requests
```

CS119 QUIZ 7 - Darcy Corson

```
import nltk

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

# Initialize Spark Session
spark = SparkSession.builder \
    .appName("Poe Stories Analysis") \
    .master("local[*]") \
    .getOrCreate()

# Get stopwords
stopwords_url =
"https://gist.githubusercontent.com/rg089/35e00abf8941d72d419224cfd5b5925d/raw/12d899b70156fd0041fa9778d6
57330b024b959c/stopwords.txt"
stopwords = set(requests.get(stopwords_url).text.lower().split())

# Clean and tokenize text
def clean_and_tokenize(text):
    sentences = nltk.sent_tokenize(text)
    cleaned_tokens = []
    for sentence in sentences:
        tokens = nltk.word_tokenize(sentence)
        for token in tokens:
            if token.lower() not in stopwords and token.isalpha():
                if token == tokens[0] or token.isupper():
                    cleaned_tokens.append(token)
                else:
                    cleaned_tokens.append(token.lower())
    return cleaned_tokens

clean_tokenize_udf = udf(clean_and_tokenize, ArrayType(StringType()))

# POS tagging
def pos_tag(tokens):
    return nltk.pos_tag(tokens)

pos_tag_udf = udf(pos_tag, ArrayType(StructType([
    StructField("word", StringType(), False),
    StructField("tag", StringType(), False)
])))

# Title of the first story
title = "A_DESCENT_INTO_THE_MAELSTROM"

# Base URL for the text files
base_url = "https://raw.githubusercontent.com/singhj/big-data-repo/main/text-proc/poe-stories/"
file_url = base_url + title

response = requests.get(file_url)
if response.status_code == 200:
    story_text = response.text
    df = spark.createDataFrame([(title, story_text)], ["Title", "Story_Text"])
    df = df.withColumn("Tokens", clean_tokenize_udf("Story_Text"))
```

CS119 QUIZ 7 - Darcy Corson

```
    df = df.withColumn("POS_Tags", pos_tag_udf("Tokens"))

    df_exploded = df.select("Title", "Story_Text", explode("POS_Tags").alias("POS"))
    df_exploded = df_exploded.select("Title", "Story_Text", col("POS.word").alias("Word"), col("POS.tag").alias("Tag"))

    pos_grouped = df_exploded.groupBy("Title", "Story_Text").pivot("Tag").agg(collect_set("Word"))
    pos_grouped.show(truncate=False)
else:
    print(f"Failed to fetch '{title}'. HTTP Status: {response.status_code}")

spark.stop()
```

See the POS tags below.

```
+---+
|Tag|
+---+
|JJS|
|JJR|
|RBR|
|NNS|
|JJ |
|FW |
|VBZ|
|VBG|
|RB |
|VBN|
|VBD|
|VB |
|IN |
|NN |
|VBP|
|NNP|
+---+
```

e. [5 points] Discuss what parts of your solution would need to change to have it scale to handle a corpus of 10,000 stories.

   To scale the solution for processing a corpus of 10,000 stories, substantial modifications would be necessary to enhance its scalability, efficiency, and robustness. Optimizing the Spark configuration is essential to handle the increased data volume, which would likely involve augmenting the number of executors alongside boosting their memory and CPU capacities. Employing cloud services that provide scalable and manageable cluster configurations, such as Google Dataproc, would facilitate effective management of these adjustments. In terms of data handling and storage, using a distributed file system like HDFS would ensure high throughput and fault tolerance, which are crucial for integrating smoothly with Spark. Performance optimizations would also play a critical role and could include leveraging advanced Spark features such as data partitioning, caching, and broadcasting to enhance processing efficiency.

Incorporating libraries like Spark NLP, which support distributed computing, could significantly improve preprocessing times for computationally intensive tasks. Establishing comprehensive error handling and monitoring systems would be important for managing and troubleshooting distributed tasks effectively. Further, leveraging Spark's dynamic resource allocation could optimize resource usage based on workload demands, adjusting automatically during peak processing times. Conducting extensive testing and validation with a subset of the corpus would help fine-tune configurations and ensure robust system performance under increased loads. Finally, implementing stringent backup and recovery procedures is crucial to prevent data loss and maintain data integrity, ensuring the system's reliability as it scales to handle larger datasets.