# Question 1 Report

## Rohan Gore

***~rmg9725***

```
Files attached:
    -  mapper1.py, reducer1.py, job1_part-00000, job1_part-00001
    -  mapper2.py, reducer2.py, job2_output.txt,

    -  mapper3.py, reducer3.py, job3_part-00000, job3_part-00001
    -  mapper4.py, reducer4.py, job4_output
```

Employed a **multi-stage MapReduce approach**, with each job building upon the results of the previous one. For solving questions 1.1 and 1.2 the solution procedure has been broken down into **four MapReduce jobs**.

## Overall Strategy

1. **Job 1: Calculate Station-Region Average Temperatures** - This job processes the raw temperature data to compute the average temperature for each station in each region.
2. **Job 2: Calculate Temperature Anomalies** - This job takes the original temperature readings and the average temperatures calculated in Job 1 to compute temperature anomalies for each reading.
3. **Job 3: Calculate Regional Volatility -** This job takes the output from Job 2 (temperature anomalies for each region) and calculates the standard deviation of these anomalies, which represents the temperature volatility for each region.
4. **Job 4: Find Top 3 Regions by Volatility -** This job processes the output from Job 3 to identify the three regions with the highest temperature volatility.

## Detailed Approaches:

### Job 1 Approach:

1. We create a composite key by combining the station ID and region code (`StationID_RegionCode`) which allows us to group temperature readings by both station and region simultaneously.
2. **Map Phase**: The mapper extracts the station ID, region code, and temperature from each record and emits key-value pairs with the composite key and temperature value.
3. **Reduce Phase**: The reducer receives all temperature readings for each station-region pair and calculates the average temperature by **maintaining running sums and counts**.
4. The final output consists of records with the format `StationID_RegionCode\tAverageTemperature`, **which will be used in Job 2**.

### Job 2 Approach:

1. **Using the distributed cache mechanism to make the average temperatures calculated in Job 1 available to all mappers in Job 2**.
2. **Map Phase**: The mapper reads the original temperature data, looks up the corresponding station-region average temperature from cache, and calculates the anomaly.
3. **Reduce Phase**: The reducer collects all anomalies for each region, which will be used in subsequent jobs to calculate volatility.
4. The final output consists of records with the format `RegionCode\tAnomaly`, which will be used in Job 3 to calculate regional volatility.

**Job 3 Approach:**

1. We **process the temperature anomalies from Job 2** to calculate the standard deviation (volatility) for each region.
2. **Map Phase:** The mapper simply passes through the data without modification, preserving the region code and anomaly values.
3. **Reduce Phase:** The reducer collects all anomalies for each region, calculates the mean, and then computes the standard deviation using the formula sqrt(sum((x - mean)^2) / n).
4. The final output consists of records with the format RegionCode\tVolatility, which will be used in Job 4 to identify the top regions.

**Job 4 Approach:**

1. We **process the region volatility data from Job 3** to identify the top 3 regions with the highest temperature volatility.
2. **Map Phase:** The mapper passes through the region code and volatility values without modification.
3. **Reduce Phase:** Using a **single reducer** (numReduceTasks=1), we collect all region-volatility pairs, sort them by volatility in descending order, and select the top 3 regions.
4. The final output consists of a ranked list of the top 3 regions with their volatility values, formatted as "Rank X: Region YYY - Volatility: Z.ZZZ".

# Question 1.1: *Documentation with Commands*

## Job 1: Calculate Station-Region Average Temperatures

1. Uploaded mapper1.py and reducer1.py code files

```
cat > mapper3.py << ‘EOF’
--- code (attached mapper3.py) ---
EOF
```

```
    cat > reducer3.py << ‘EOF’
    --- code (attached reducer3.py) ---
EOF
```

2. Upload temperature-data and shift it to HDFS

```
hdfs dfs -put temperature-data.txt hw1-rmg9725/data/
```

3. Make the scripts executable:

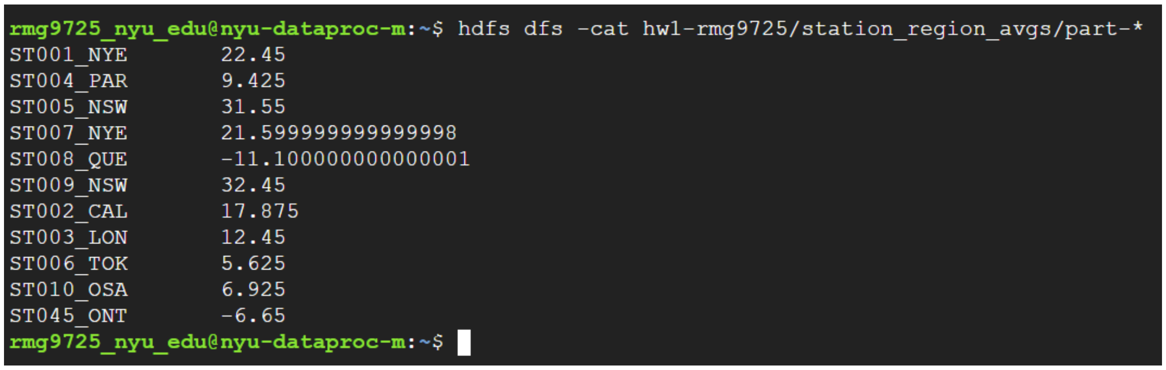
```
chmod +x mapper1.py reducer1.py
```

4. Run Job 1:

```
mapred streaming \
       -files mapper1.py,reducer1.py \
       -mapper mapper1.py \
       -reducer reducer1.py \
       -input hw1-rmg9725/temperature-data.txt \
       -output hw1-rmg9725/station_region_avgs
```

5. Watch output:

```
hdfs dfs -cat hw1-rmg9725/station_region_avgs/part-*
```



```
rmg9725_nyu_edu@nyu-dataproc-m:~$ hdfs dfs -cat hw1-rmg9725/station_region_avgs/part-*
ST001_NYE       22.45
ST004_PAR       9.425
ST005_NSW       31.55
ST007_NYE       21.599999999999998
ST008_QUE       -11.100000000000001
ST009_NSW       32.45
ST002_CAL       17.875
ST003_LON       12.45
ST006_TOK       5.625
ST010_OSA       6.925
ST045_ONT       -6.65
rmg9725_nyu_edu@nyu-dataproc-m:~$
```

6. Bringing file to local from hdfs and downloading for confirmation:

```
hdfs dfs -get hw1-rmg9725/station_region_avgs/part-00000
hdfs dfs -get hw1-rmg9725/station_region_avgs/part-00001
```

**Output Files Attached: job1_part-00000.txt, job1_part-00001.txt,**

## Job 2: Calculate Temperature Anomalies

1. Combine all parts of output of job 1 on local

```
mkdir -p job1_output
cat job1_output/part-* > station_region_avgs.txt
```

2. Created and wrote mapper3.py and reducer3.py using following commands:

```
cat > mapper3.py << ‘EOF’
--- code (attached mapper3.py) ---
EOF
```

```
cat > reducer3.py << ‘EOF’
--- code (attached reducer3.py) ---
EOF
```

3. Make the scripts executable:

```
chmod +x mapper2.py reducer2.py
```

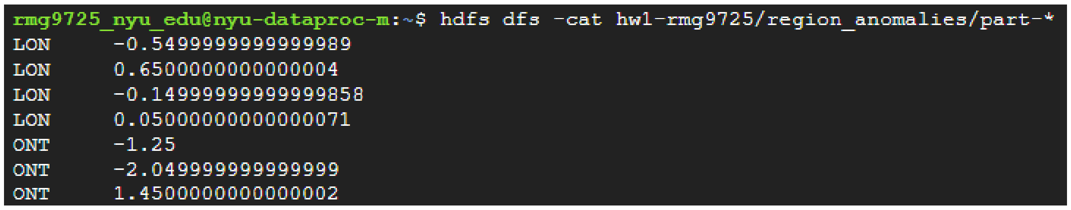
4. Run Job 2:

```
mapred streaming \
  -files mapper2.py,reducer2.py,station_region_avgs.txt \
  -mapper mapper2.py \
  -reducer reducer2.py \
  -input hw1-rmg9725/temperature-data.txt \
  -output hw1-rmg9725/region_anomalies \
  -cmdenv STATION_REGION_AVGS=station_region_avgs.txt
```

**The `-cmdenv STATION_REGION_AVGS=station_region_avgs.txt` option sets an environment variable that tells the mapper where to find the station-region averages file.**

5. Watch output:

```
hdfs dfs -cat hw1-rmg9725/region_anomalies/part-*
```



```
rmg9725_nyu_edu@nyu-dataproc-m:~$ hdfs dfs -cat hw1-rmg9725/region_anomalies/part-*
LON     -0.5499999999999989
LON     0.6500000000000004
LON     -0.14999999999999858
LON     0.05000000000000071
ONT     -1.25
ONT     -2.049999999999999
ONT     1.4500000000000002
```

6. Downloading output files

```
rm part-00000
rm part-00001
hdfs dfs -getmerge hw1-rmg9725/region_anomalies region_anomalies_combined.txt
```

**Output Files Attached: job2_output.txt**

# Question 1.2: *Documentation with Commands*

## Job 3: Calculate Regional Volatility

1. Created and wrote mapper3.py and reducer3.py using following commands:

```
cat > mapper3.py << ‘EOF’
--- code (attached mapper3.py) ---
EOF
```

```
cat > reducer3.py << ‘EOF’
--- code (attached reducer3.py) ---
EOF
```

2. Make the scripts executable:

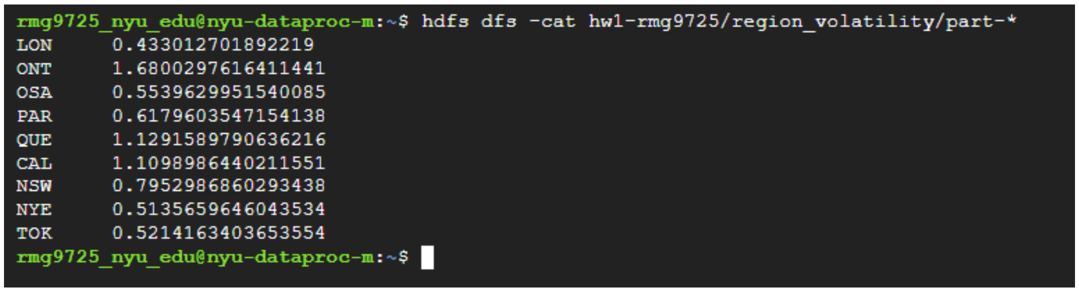
```
chmod +x mapper3.py reducer3.py
```

3. Run Job 3:

```
mapred streaming \
  -files mapper3.py,reducer3.py \
  -mapper mapper3.py \
  -reducer reducer3.py \
  -input hw1-rmg9725/region_anomalies \
  -output hw1-rmg9725/region_volatility
```

4. Watch output:

```
hdfs dfs -cat hw1-rmg9725/region_volatility/part-*
```



```
rmg9725_nyu_edu@nyu-dataproc-m:~$ hdfs dfs -cat hw1-rmg9725/region_volatility/part-*
LON     0.433012701892219
ONT     1.6800297616411441
OSA     0.5539629951540085
PAR     0.6179603547154138
QUE     1.1291589790636216
CAL     1.1098986440211551
NSW     0.7952986860293438
NYE     0.5135659646043534
TOK     0.5214163403653554
rmg9725_nyu_edu@nyu-dataproc-m:~$
```

5. Downloading output files for confirmation:

```
rm part-00000
rm part-00001
hdfs dfs -get hw1-rmg9725/region_volatility/part-00000
hdfs dfs -get hw1-rmg9725/region_volatility/part-00001
```

**Output Files Attached: job3_part-00000.txt, job3_part-00001.txt,**

## Job 4: Find Top 3 Regions by Volatility

1. Created and wrote mapper4.py and reducer4.py using following commands:

```
cat > mapper4.py << ‘EOF’
--- code (attached mapper4.py) ---
EOF
```

```
cat > reducer4.py << ‘EOF’
--- code (attached reducer4.py) ---
EOF
```

2. Make the scripts executable:

```
chmod +x mapper4.py reducer4.py
```

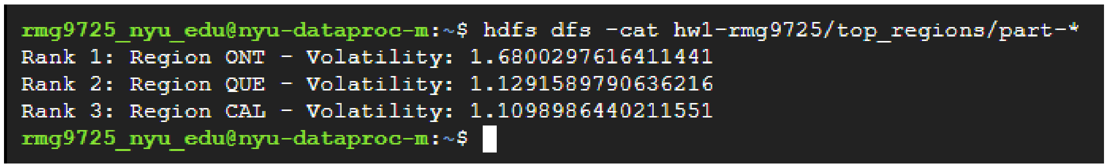
3. Run Job 4:

```
mapred streaming \
  -files mapper4.py,reducer4.py \
  -mapper mapper4.py \
  -reducer reducer4.py \
  -numReduceTasks 1 \
  -input hw1-rmg9725/region_volatility \
  -output hw1-rmg9725/top_regions
```

**The `-numReduceTasks 1` parameter ensures all data is processed by a single reducer, allowing us to find the global top 3 regions.**

4. Watch output:

```
hdfs dfs -cat hw1-rmg9725/top_regions/part-*
```



```
rmg9725_nyu_edu@nyu-dataproc-m:~$ hdfs dfs -cat hw1-rmg9725/top_regions/part-*
Rank 1: Region ONT - Volatility: 1.6800297616411441
Rank 2: Region QUE - Volatility: 1.1291589790636216
Rank 3: Region CAL - Volatility: 1.1098986440211551
rmg9725_nyu_edu@nyu-dataproc-m:~$
```

5. Download final output:

```
hdfs dfs -get hw1-rmg9725/top_regions/part-00000
```

**Output File Attached: job4_output.txt**

## Addressed the Challenges

1. **Challenge: Calculate regional averages before determining anomalies, but you can't hold all data in memory.**
    - **Solution**: We split the calculation into two separate MapReduce jobs. Job 1 calculates the average temperature for each station in each region, and Job 2 uses these averages to calculate anomalies. This approach allows us to process datasets of any size without holding all data in memory at once.

2. **Challenge: There's no global state in MapReduce streaming, so computing standard deviations requires thought.**
    - **Solution**: We address this challenge by using a multi-stage approach. In Job 2, we collect all anomalies for each region, which will be used in Job 3 (in upcoming section) to calculate standard deviations. By breaking the computation into stages and passing intermediate results between jobs, we overcome the stateless limitation of MapReduce.

3. **Challenge: Computing top-3 regions requires aggregation across the entire dataset.**
    - **Solution**: This challenge is addressed in Job 4 (not covered in this report), where we use a single reducer to process all volatility values and find the global top 3 regions. By setting `-numReduceTasks 1`, we ensure that all data is processed by a single reducer, allowing us to find the global top 3 regions.

## Meeting the Requirements

1. **Requirement: Implement this solution using MapReduce Streaming with any language of your choice (Python, Ruby, Perl, etc.)**
    - **Solution**: We implemented the solution using MapReduce Streaming with Python, which allows for rapid development and easy handling of text data.

2. **Requirement: Your solution must scale to handle datasets too large to fit in memory.**
    - **Solution**: Our solution scales to handle large datasets by:
        - Breaking the computation into multiple MapReduce jobs, each focusing on a specific task
        - Using the distributed nature of MapReduce to process data in parallel across multiple nodes
        - Minimizing memory usage in mappers and reducers by processing records one at a time
        - Using the distributed cache to efficiently share data between jobs

3. **Requirement: The solution can be solved using multiple MapReduce jobs that communicate through intermediate files, environment variables and/or command line arguments.**

- **Solution**: Our solution uses multiple MapReduce jobs that communicate through:
    - Intermediate files: The output of Job 1 (station-region averages) is used as input for Job 2
    - Environment variables: We use the `-cmdenv` parameter to pass the location of the station-region averages file to the mapper in Job 2
    - Command line arguments: We use the `-files` parameter to include the necessary scripts and data files in the job

4. **Requirement: Explain the data flow between your MapReduce jobs and how each overcomes the stateless limitation.**
    - **Solution**:
        - **Data Flow**: Job 1 processes the raw temperature data and outputs station-region average temperatures. Job 2 uses these averages along with the original data to calculate temperature anomalies for each region.
        - **Overcoming Stateless Limitation**:
            - ♣ We use the distributed cache to share the output of Job 1 with the mappers in Job 2
            - ♣ We use environment variables to tell the mappers where to find the shared data
            - ♣ We design our reducers to maintain state for the current key being processed, allowing us to perform aggregations like calculating averages
            - ♣ We break complex calculations (like standard deviation) into multiple jobs, with each job building on the results of the previous one

By carefully designing our MapReduce jobs and the data flow between them, we've created a solution that effectively addresses the challenges and meets all the requirements specified in the problem statement.

# Question 2 Report

## Rohan Gore

***~rmg9725***

```
Files attached:
    -  midterm_q2.ipynb
```

## Question 2.1: Customer Spending Analysis

## Detailed Approach

The customer spending analysis aims to identify which customers spend the most money overall by computing the total amount spent by each customer, excluding cancelled orders. The approach involves several key steps:

1. **Data Loading**: First, we load the three JSON datasets (customers, orders, and order_items) using Spark's JSON reader.
2. **Order Total Calculation**: For each order, we calculate the total amount by multiplying the quantity by the unit price for each item in the order and then summing these values.
3. **Filtering Valid Orders**: We filter out cancelled orders since they should not count toward customer spending.
4. **Customer Spending Aggregation**: We group the valid orders by customer_id and calculate the total spending for each customer.
5. **Joining with Customer Data**: We join the spending data with customer information to get additional details like customer name and tier.
6. **Rounding and Sorting**: We round the total spent to exactly two decimal places and sort the results in descending order to identify the highest spenders.

## Real-Life Use Cases

- **Customer Loyalty Programs**: Businesses can identify their highest-spending customers and offer them special rewards or premium membership status, increasing retention and lifetime value of these valuable customers.
- **Targeted Marketing Campaigns**: By understanding which customers spend the most, businesses can create personalized marketing campaigns that cater to their preferences and purchasing habits, improving marketing ROI and customer engagement.

## Question 2.2: Category Preference Analysis

## Detailed Approach

The category preference analysis aims to identify the most popular product categories for each customer tier (Gold, Silver, Bronze). The approach involves:

1. **Data Loading**: Loading the three JSON datasets as in Question 2.1.
2. **Category Extraction**: Since categories are stored as an array in the orders data, we use the explode function to convert each category in the array to a separate row.
3. **Filtering Valid Orders**: We filter out cancelled orders to ensure we're only analyzing categories from completed or processing orders.
4. **Joining with Customer Data**: We join the category data with customer information to get the tier information.
5. **Category Counting by Tier**: We count the occurrences of each category within each tier to determine popularity.
6. **Ranking Categories**: We use window functions to rank categories within each tier based on their count.
7. **Filtering Top Categories**: We filter to get only the most popular category (rank = 1) for each tier.
8. **Separate Display by Tier**: We filter the results to display separate tables for Gold, Silver, and Bronze tiers.

## Real-Life Use Cases

- **Personalized Product Recommendations**: Businesses can use tier-based category preferences to recommend relevant products to customers in each tier, increasing cross-selling opportunities and customer satisfaction.
- **Inventory Management**: Understanding which categories are most popular among different customer tiers helps businesses optimize inventory planning and ensure adequate stock for high-demand categories.

## Question 2.3: Price Range Preferences

## Detailed Approach

This analysis provides insights into whether higher-tier customers (e.g., Gold) purchase more premium products compared to lower-tier customers (e.g., Silver or Bronze). The approach involves:

1. **Price Classification**: Implementing the provided `classify_price` function as a User-Defined Function (UDF) to categorize products into three price ranges:
    - Budget: Products priced below $50.00
    - Mid-range: Products priced between $50.00 and $199.99
    - Premium: Products priced at $200.00 or above
2. **Item Total Calculation**: For each order item, calculating the total amount by multiplying the quantity by the unit price.
3. **Filtering Valid Orders**: Filtering out cancelled orders since they should not be included in the spending analysis.
4. **Data Integration**: Joining the classified items with valid orders and customer information to associate each purchase with the customer's tier.
5. **Spending and Product Count Aggregation**: Grouping the data by customer tier and price category to calculate the total spending and product count in each segment.
6. **Percentage Calculation**: For each tier, calculating what percentage of their total spending goes to each price category.
7. **Rounding and Formatting**: Rounding all monetary values and percentages to exactly two decimal places for consistency and readability.
8. **Separate Display by Tier**: Filtering the results to display separate tables for Gold, Silver, and Bronze tiers.

## Real-Life Use Cases

- **Tiered Pricing Strategy Optimization**: Businesses can optimize their tiered pricing strategy by understanding how different customer segments respond to products across price ranges, potentially increasing revenue by aligning pricing with customer preferences.
- **Targeted Product Development**: Companies can develop products that align with the price preferences of each customer tier, focusing resources on creating offerings that resonate with their most valuable customer segments.

## Question 2.4: Top Customer Identification

## Detailed Approach

The top customer identification analysis aims to identify the top 2 highest-spending customers within each tier-price category combination. This provides a granular view of who the most valuable customers are within each specific segment. The approach involves:

1. **Data Integration**: Using the classified items DataFrame from Question 2.3, we join it with valid orders and customer information to associate each purchase with the customer's information and tier.
2. **Spending Aggregation by Customer, Tier, and Price Category**: We group the data by customer ID, name, tier, and price category to calculate the total spending for each customer within each tier-price category combination.
3. **Ranking Customers**: We use window functions to rank customers within each tier-price category combination based on their total spending.
4. **Filtering Top Customers**: We filter to get only the top 2 customers (rank <= 2) for each tier-price category combination.
5. **Rounding and Formatting**: We round all monetary values to exactly two decimal places for consistency and readability.
6. **Separate Display by Tier**: We filter the results to display separate tables for Gold, Silver, and Bronze tiers.

## Real-Life Use Cases

- **VIP Customer Programs**: Businesses can identify their most valuable customers within each segment and offer them special VIP treatment, exclusive access to new products, or personalized services, enhancing loyalty among high-value customers.
- **Personalized Marketing**: Companies can create highly targeted marketing campaigns for top customers in each segment, tailoring messages and offers to their specific preferences and spending patterns, resulting in higher engagement and conversion rates.

# Question 3 Report

## Rohan Gore

***~rmg9725***

```
Files attached:
    -  midterm_q3.ipynb
```

## Question 3.1: Customer Value Segmentation

## Detailed Approach

1. **Recency Calculation**:
   - Filter completed orders.
   - Group by customer_id.
   - Calculate the difference between the current date (April 13, 2025) and the most recent order date for each customer.
2. **Frequency Calculation**:
   - Filter completed orders.
   - Group by customer_id.
   - Count the number of orders for each customer.
3. **Monetary Value Calculation**:
   - Calculate total amount for each order by multiplying quantity and unit price.
   - Join with completed orders.
   - Calculate average order value for each customer.
   - Round the average value to two decimal places.
4. **RFM Combination**: Join the recency, frequency, and monetary value dataframes on customer_id.
5. **Segmentation Function**:
   - Implement the provided `segment_customer` function as a User-Defined Function (UDF).
   - This function assigns scores to recency, frequency, and monetary value.
   - Calculates a total score and assigns a segment based on the score.
6. **Segmentation Application**: Apply the UDF to the combined RFM dataframe to assign segments to each customer.
7. **Result Preparation**:
   - Join the segmentation results with customer information.
   - Select only the required columns: customer_id, name, tier, and segment.
   - Show the customer segmentation results.

## Real-Life Use Cases

1. **Personalized Marketing Campaigns**: Businesses can tailor their marketing strategies based on customer segments. For example, they might offer exclusive deals to "High Value" customers to maintain their loyalty, while providing incentives to "Medium Value" customers to increase their engagement and potentially move them to the "High Value" segment.
2. **Resource Allocation**: Companies can optimize their customer service and support resources by prioritizing "High Value" customers, ensuring they receive premium support to maintain their satisfaction and loyalty. This segmentation also helps in identifying "Low Value" customers who might need different strategies to increase their value or reduce service costs.

### Question 3.2: Top 3 Categories

## Detailed Approach

1. **Data Loading**: Load the customers and orders JSON datasets.
2. **Category Extraction**:
   - Filter out cancelled orders.
   - Use the `explode` function to transform the categories array into individual rows, creating a separate row for each category in an order.
3. **Category Counting**:
   - Group by customer_id and category.
   - Count the occurrences of each category for each customer.
4. **Category Ranking**:
   - Use a window function to rank categories for each customer based on purchase count.
   - The ranking is done in descending order of purchase count.
5. **Top 3 Selection**: Filter to keep only the top 3 ranked categories for each customer.
6. **Result Preparation**:
   - Join the top categories with customer information.
   - Select relevant columns: customer_id, name, category, purchase_count, and rank.
7. **Result Display**:
   - Show the top 3 categories for each customer.
   - Optionally, display a more detailed, customer-by-customer breakdown of the top categories.

## Real-Life Use Cases

1. **Personalized Product Recommendations**: E-commerce platforms can use this analysis to create tailored product recommendations for each customer. By knowing a customer's top 3 categories, the platform can prioritize showing products from these categories or related categories, potentially increasing the likelihood of additional purchases and improving the customer's shopping experience.
2. **Inventory Management and Stock Optimization**: Retailers can use this information to optimize their inventory levels and stock placement. Understanding which categories are most popular among their customer base allows them to ensure adequate stock of high-demand items and potentially reduce overstocking in less popular categories, leading to more efficient inventory management and reduced carrying costs.