VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

SEMESTER 2 (2024-2025)

IT161IU
Big Data Technology

TITLE

Lecturer: **Dr. Ho Long Van**

By Group: **Bang Hai Tac**

Ho Chi Minh City, Vietnam
May 2025

No.	**Full name - Student name**	**Student ID**	**Contribution (%)**
1	PHAM VU TUYET ANH	ITDSIU21073	25
2	DINH THI THANH HANG	ITDSIU21083	25
3	LE THI TUYET MY	ITDSIU21005	25
4	DINH VU NGOC LINH	ITDSIU21095	25

Table 1: Student Contributions

Acknowledgments

We would like to sincerely thank **Dr. Ho Long Van** for his invaluable guidance and support throughout the *Big Data* course. His insightful lectures and clear explanations provided us with a strong foundation in data analysis, machine learning, and the practical use of big data tools, which were essential in the successful completion of this project. We are especially grateful for the opportunity to apply what we learned in a hands-on manner, allowing us to deepen our understanding of the concepts through real-world implementation.

We also extend our heartfelt thanks to our entire team for their cooperation, enthusiasm, and dedication throughout every stage of this project. From data preprocessing and exploratory data analysis to the development of the clustering model and the creation of a user-friendly `Streamlit` dashboard, each team member contributed significantly to the final outcome. This collaborative effort not only enhanced our technical skills but also strengthened our ability to work effectively as a team on a complex data science workflow.

This project has been an enriching experience that brought together theory and practice, and we are deeply appreciative of the support and teamwork that made it possible.

Table of Contents

List of Tables

List of Figures

Chapter 1
INTRODUCTION

1.1 Background

In the present day, the retail market is highly saturated, with many sources of demand providing products for a limited amount of customers. Understanding customer behavior is critical, to develop personalized marketing, sales and customer retaining strategies that help business owners to achieve their own business goals, with optimized use of resources. By leveraging tools such as Python, Spark, and Streamlit, we conducted a comprehensive analysis to uncover insights into customer purchasing patterns.

1.2 Objectives

The key objective of this project is to help business owners develop personalized marketing, and sales strategies for each customer segments. The following objective is achieved through the procedure below. The procedure started with cleaning and preprocessing the raw transaction data, then the RFM framework is applied, and customers are segmented into two groups: churned or active based their activity. A clustering model is then applied to cluster customer into sub-segments. Finally, Streamlit will serve as a visualization platform to visualze the results of previous part, enabling data visualization for end users.

1. This is the first **bold keyword** of the ordered list to emphasize an important concept.
2. The second point in the ordered list addresses **another key idea** in the discussion.

    - This is the first **bold keyword** of the unordered list to emphasize an important concept.
    - The second point addresses *italic keyword* in the unordered list discussion.

Chapter 2
RELATED WORK

2.1 Streamlit

2.1.1 Definition

Streamlit is an open-source Python framework (library) designed for data scientists and AI/ML engineers to build interactive web applications for data visualization and analysis. In essence, Streamlit "makes it easy to build beautiful custom web-apps for machine learning and data science" [1, 2]. It allows users to turn data scripts into shareable web apps "in just a few minutes, all using pure Python"[3]. By abstracting away traditional web development (HTML/CSS/JavaScript), Streamlit enables a "no-front-end" workflow: one writes a normal Python script that directly produces a GUI in the browser[3].

2.1.2 Benefits and Use Cases

Streamlit's key advantages include rapid prototyping and ease of use. Its simple, Pythonic API lets developers create fully-featured apps with only a few lines of code [3]. Because Streamlit natively integrates with common data libraries (Pandas, NumPy, Matplotlib, Plotly, Altair, etc.), it facilitates seamless visualization and exploration of data [1, 3]. Streamlit apps automatically re-run and update when input widgets change, enabling real-time interactivity without manual refreshes [3]. The framework also supports deployment on various platforms (e.g. Streamlit Community Cloud, Heroku, AWS, or within Snowflake) with minimal effort [3, 4]. As a free, open-source tool under an Apache-2.0 license, Streamlit has gained traction as a convenient way to deploy ML models and dashboards. For example, data scientists use Streamlit to create interactive model-demo apps (e.g. a loan-prediction interface) and exploratory dashboards (e.g. visualizing US population data) without writing any JavaScript. In practice, Streamlit is commonly used for quick data dashboards, EDA interfaces, teaching tools, and prototyping machine-learning apps. [4]

2.1.3 Setting Up the Environment and Required Libraries

To develop a Streamlit application efficiently and avoid library conflicts, it's recommended to create a separate Python environment. Below are two common methods to set up a new environment: using `conda` or `venv`.

Method 1: Using `conda` (if you use Anaconda or Miniconda)

1. Create a new environment:

```
conda create -n streamlit_app -y
```

- `-n streamlit_app`: specifies the name of the new environment as `streamlit_app`.
- `-y`: automatically confirms all prompts during installation.

2. Activate the environment:

```
conda activate streamlit_app
```

Method 2: Using `venv` (if you are not using Anaconda)

1. Create a virtual environment:

```
python3 -m venv streamlit_env
```

2. Activate the environment:

- On macOS or Linux:

```
    source streamlit_env/bin/activate
```

2.1.4 Installing Required Libraries

Once the environment is activated, install the necessary libraries using the following command:

```
pip install streamlit pandas plotly openpyxl
```

- **streamlit**: used to build interactive web applications in Python.
- **pandas**: for data manipulation and analysis using DataFrames.
- **plotly**: for creating visualizations and plotting data with enabling interactive.
- **openpyxl**: for reading and writing Excel files with the `.xlsx` extension.

2.1.5 Functional Overview

Streamlit apps are written as simple Python scripts. When the script runs, Streamlit interprets commands like st.write(), st.map(), etc., and automatically generates a web UI. Below is an overview of the core components and features of Streamlit [1]:

1. Text and Markdown Output:
   - The function st.write() is very flexible – it can display text, Markdown, data frames, and even Matplotlib/Plotly figures
   - For formatted text, Streamlit provides convenience functions: st.title() for main title of the application, st.header() for header of page, st.subheader() for lower level of front size, st.markdown(), st.code(), st.latex(), and others

For example, st.markdown("**bold text**") renders Markdown, and st.write("Hello World") prints plain text. These elements allow building rich text content in the app.

2. Tabular Data and Metrics:

   - To display data, Streamlit offers st.dataframe() and st.table(). st.dataframe(df) shows an interactive table (sortable, scrollable) of a Pandas DataFrame df, while st.table(df) renders a static table view.
   - Numeric key performance indicators can be displayed with st.metric(label, value, delta), which shows a big number (and an optional delta)

   For example, one can create a sidebar summary of metrics or show a data table of results. Streamlit also supports st.json() for pretty-printing JSON and st.image()/st.video()/st.audio() for media content (not shown above but similarly straightforward).

3. Charts and Plots:

   - Streamlit includes several built-in chart primitives. Functions like st.line_chart(df), st.area_chart(df), and st.bar_chart(df) quickly generate simple charts from a DataFrame.
   - For more customized plots, Streamlit integrates with popular visualization libraries. For instance, a Plotly figure can be displayed via st.plotly_chart(fig), an Altair chart with st.altair_chart(chart), a Matplotlib figure with st.pyplot(fig), etc.
   - There are also st.scatter_chart(), st.map(df) (to plot latitude/longitude points on an interactive map), st.bokeh_chart(), st.pydeck_chart(), and others.

4. Widgets and Interactivity: Streamlit provides many input widgets that let users interact with the app. Common widgets include sliders (st.slider), buttons (st.button), dropdowns (st.selectbox), checkboxes (st.checkbox), text inputs (st.text_input), color pickers, date pickers, and more. Each widget returns a value that can be used in the script. Whenever a widget's state changes, Streamlit automatically re-runs the script from the top, updating outputs. To help organize UI, Streamlit offers layout primitives: st.sidebar places widgets in a sidebar panel, while st.columns(n) creates n columns for side-by-side containers. There are also st.expander (collapsible sections), st.tabs (tabbed panels), and st.container/st.empty for advanced layout control. For example, one blog illustrates using st.columns to arrange three columns and placing a metric widget in one of them.

5. Caching and State: Since Streamlit reruns the script on each interaction, it provides caching to improve performance. The decorator "@st.cache_data" can be applied to functions that load or compute data to avoid recomputation. Similarly, "@st.cache_resource" can cache objects like trained models or database connections. To retain user-specific state across reruns, Streamlit offers "st.session_state", a dict-like object where the app can store and retrieve variables.

   In summary, Streamlit abstracts the boilerplate of web development and provides a rich set of UI components (text, tables, charts, widgets, layouts) that data scientists can use directly from Python. Its ease of installation and deployment, along with interactive features, make it a powerful tool for developing and sharing data science and ML applications.

Chapter 3
METHODOLOGY

3.1 Use Case Diagram

The requirement analysis phase is a foundational step in designing the application of spark in analyst and clustering data, which involves identifying the essential features of the system and understanding the needs of its target users. To ensure clarity and alignment with user expectations, this section includes visual representations like use case diagrams and system overiew.

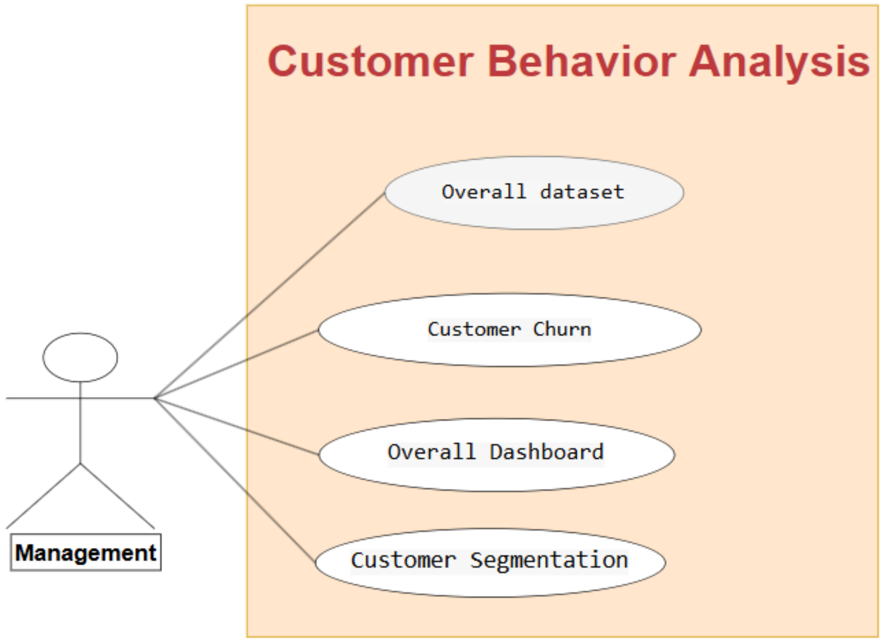


Figure 3.1: Use case diagrams

Figure 3.1 presents the use case diagram of the system, which provides a system boundaries and a comprehensive overview of the system's functional requirements, elucidating the tasks it supports.

Actor:

The management is the central actor in the system.

Core Use Cases:

- Overall dataset: The management can see the overall information of dataset, which include data dimension, key metrics in dataset and the overall of dataset.
- Customer churn: The management can manage the quality of customer to know the analyst of customer churn based on some key metric like "Recency", "Frequency". "Montetary".
- Overall Dashboard show the overall factors effect on online retail transaction, where management can see all chart about the factor on this.
- Customer segement is where cluster all customer into group. Then based on this can get the decision for each group.

3.2 Preprocessing

3.2.1 Dataset schema

A dataset schema is a blueprint that outlines how particular data, such as in a database, is structured, configured and organized. It provides a reference point that indicates what fields of information the project contains. This makes the data easily understandable and improves management and efficiency. A schema does not contain the actual data but describes the structure and constraints that apply to that data.

3.2.2 Missing data

Missing data occurs when one or more important fields in a customer or transaction are not recorded. These gaps can impact the accuracy of behavioral analytics and predictive loyalty models. Missing data can be the result of data collection errors or incomplete customer information. Proper handling of missing data is essential to maintaining the reliability and completeness of the data set.

3.2.3 Duplicate record

Duplicate records refer to repeated or nearly identical entries in the dataset that represent the same customer transaction or event. For example, an invoice might be recorded multiple times due to system errors or data entry mistakes. Duplicate data can distort customer behavior analysis and loyalty measurement, leading to incorrect insights and inefficient resource use.

3.2.4 Abnormal Values

Abnormal values are data entries that deviate significantly from expected or logical norms, often indicating errors, exceptions, or special cases. In transactional datasets, these typically include:

- Negative Quantity or UnitPrice: Normally, sales transactions should not have negative quantity or unit price values. These can result from data entry errors or misrecorded refund processes. Unless specifically intended to represent returns, such values should be filtered out to preserve data integrity.

- Cancelled Orders: In datasets like retail invoices, cancelled orders are often marked with a distinct pattern (e.g., invoice IDs starting with 'C'). These represent voided transactions and should be excluded when calculating metrics like revenue, total quantity sold, or customer behavior.

Handling abnormal values is essential to avoid misleading insights and ensure that statistical or machine learning models are trained on reliable, valid data. To ensure accurate RFM feature calculation and behavioral insights, such entries are treated as abnormal and excluded from the dataset.

3.2.5 Abnormal product entries

Certain records in the dataset contain StockCode and Description pairs that do not correspond to actual sellable products. These include administrative entries such as shipping charges (CARRIAGE), discounts (D), manual adjustments (ADJUSTMENT), bank fees (BANK CHARGES), or free samples (SAMPLES). Including these non-product records may distort customer behavior analysis by inflating the number of transactions or misrepresenting monetary values.

3.3 Feature Engineering

Feature engineering involves transforming raw data into meaningful features that improve model performance. For customer behavior modeling, RFM (Recency, Frequency, Monetary) analysis is a proven technique.

3.3.1 RFM Model

The RFM model is a behavioral-based segmentation approach used to evaluate customer value based on:

- Recency (R): Time since the customer's last transaction.
- Frequency (F): Total number of transactions within a given period.
- Monetary (M): Total spending of the customer.

In this project, only Recency and Frequency are used to derive a churn label.

3.3.2 Churn Labeling using Recency and Frequency

Customer churn is defined as the likelihood that a customer stops purchasing. By setting thresholds on Recency and Frequency, a binary churn label is created: Customers with high recency (haven't bought in a long time) and low frequency (few purchases) are considered churned. Threshold values are derived based on business understanding or statistical measures (e.g., median, quantiles). This label is later used as the target variable for classification models.

3.4 Analyze

This analysis aims to understand customer churn behavior and uncover opportunities to drive long-term growth by applying the 3Ps-G Framework — a strategic lens that evaluates customer dynamics through four key dimensions: Place, People, Product, and Growth. In addition, we utilized the RFM frameworks, which examine our customers through three aspects: Recency (the time from the first time a customer purchase), Frequency (the number of purchases customer make), Monetary (the total amount of money a customer spend on our brand). This analysis will use a combined approach of both 3Ps-G framework and RFM metrics. The 3Ps-G is a framework that we develop from the famous Marketing for Ps, with an added dimension: Growth to follow the growth in our business.

3.5 K-Means Clustering

K-Means clustering algorithm – one of the most commonly used clustering algorithms. It is an unsupervised machine learning technique, widely employed to discover intrinsic groupings within a dataset. It tries to find the cluster centers that are representative of certain regions of the data. The algorithm alternates between two steps: assigning each data point to the closest cluster center and then setting each cluster center as the mean of the data points that are assigned to it. The algorithm is finished when the assignment of instances to clusters no longer changes.

3.5.1 Algorithm Steps

The K-Means algorithm proceeds through a series of iterative steps until convergence is achieved:

1. **Initialization:**
   Initially, k data points are randomly selected from the dataset to serve as the preliminary cluster centroids. To mitigate the risk of converging to suboptimal solutions, advanced initialization methods like *K-Means++* are often employed [**?**]. This method intelligently selects initial centroids to ensure a better distribution across the data space, leading to more robust clustering results.

2. **Assignment Step (E-step):**
   Each data point in the dataset is assigned to the nearest centroid. The most commonly used distance metric for this assignment is the Euclidean distance. The Euclidean distance $d(x, c)$ between a data point $x = (x_1, x_2, \ldots, x_p)$ and a centroid $c = (c_1, c_2, \ldots, c_p)$

in p-dimensional space is calculated as:

$$d(x, c) = \sqrt{\sum_{i=1}^{p} (x_i - c_i)^2}$$

3. **Update Step (M-step):**
   Following the assignment of all data points, the centroids of the k clusters are recomputed. The new centroid for each cluster is the mean of all data points currently assigned to that cluster.

4. **Iteration and Convergence:**
   Steps 2 and 3 are repeated iteratively until either the cluster assignments no longer change between iterations, or the change in centroids falls below a predefined tolerance level, indicating convergence. This iterative refinement process ensures that the within-cluster variance is minimized.

3.5.2 Data Preparation

Prior to applying the K-Means algorithm, raw datasets typically undergo essential preprocessing steps to ensure data quality and optimize clustering performance:

1. **Handling Missing Values:**
   Missing values, if present, are addressed through various strategies such as imputation (e.g., using the mean or median of the respective feature) or removal of incomplete rows or columns. The choice of method depends on the nature and extent of missingness in the dataset.

2. **Feature Assembly:**
   Before applying clustering algorithms—especially in pipeline-based machine learning frameworks—it is necessary to combine individual numerical feature columns into a single vector column. Tools like `VectorAssembler` serve this purpose by taking a list of numerical or vectorized features as input and producing a single feature vector column. This step consolidates all relevant predictive features into a format expected by algorithms like K-Means.

3. **Feature Scaling:**
   K-Means is sensitive to the scale of features due to its reliance on distance computations. To prevent features with larger ranges from disproportionately influencing clustering results, it is essential to scale all numerical features.

   Standardization (Z-score normalization):
   One common method of scaling is standardization, which transforms data such that each feature has a mean of 0 and a standard deviation of 1. The formula for Z-score normalization is:

$$x' = \frac{x - \mu}{\sigma}$$

where x' is the standardized value, x is the original value, μ is the mean of the feature, and σ is the standard deviation.

Min-Max Scaling:
Min-Max Scaling is a normalization technique that rescales data to a fixed range, typically between 0 and 1. This technique is useful when features have varying scales but you want to preserve the relationships among values. The formula for Min-Max Scaling is:

$$x' = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

where x is the original value, x_{min} is the minimum value of the feature, and x_{max} is the maximum value of the feature.

3.5.3 Determining the Optimal Number of Clusters (k)

A crucial step in K-Means clustering is determining the optimal number of clusters, k. One of the most common methods for evaluating the quality of clustering results is the **Silhouette Score**.

The silhouette score measures how well each data point fits into its assigned cluster compared to how poorly it fits into neighboring clusters. The score ranges from -1 to $+1$, with higher values indicating more distinct and well-separated clusters. The optimal k is typically associated with the highest average silhouette score.

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)} \tag{3.1}$$

where:

- a_i is the average distance from point i to all other points in its own cluster.
- b_i is the minimum average distance from point i to points in a different cluster (i.e., the closest other cluster).

Once an optimal k is determined and the K-Means algorithm is executed, the resulting cluster assignments and the positions of the cluster centers can be visualized. Figure 3.2 presents an illustrative example of cluster assignments and centers when using three clusters.

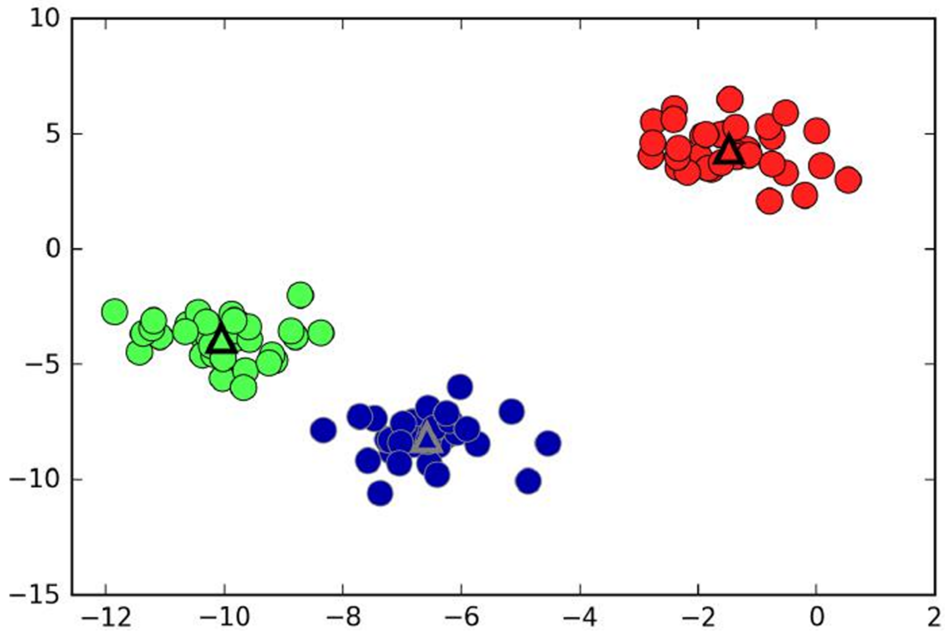


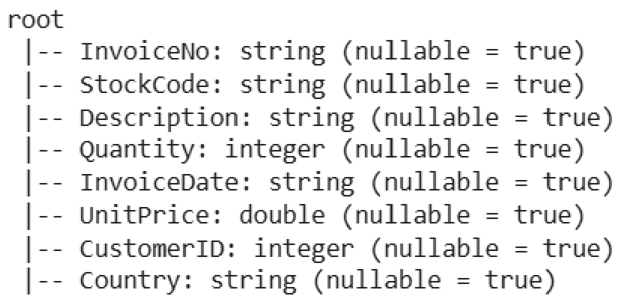
Figure 3.2: Cluster assignments and cluster centers found by K-Means with three clusters

Chapter 4
IMPLEMENTATION

4.1 Implementation of processing

4.1.1 Handle unmatched data type

Data types in the raw dataset were checked and corrected to ensure consistency with the schema. In the raw dataset, we can see that ‘InvoiceDate‘ has string datatype while its record is in datetime format.



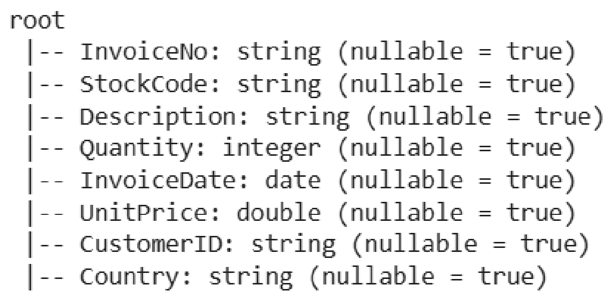
```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: string (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: integer (nullable = true)
 |-- Country: string (nullable = true)
```

Figure 4.1: Raw schema with unmatched datatype

To handle this problem, we use the withColumn() method to change the ’InvoiceDate’ format to the M/d/yyyy H:mm format.

```python
from pyspark.sql.functions import col, to_date
# Convert datatype of InvoiceDate to timestamp
df_cleaned = df.withColumn("InvoiceDate",
                     to_timestamp("InvoiceDate", "M/d/yyyy H:mm"))

# Re-check schema
df_cleaned.printSchema()
```

After implementing, we achieved the result:



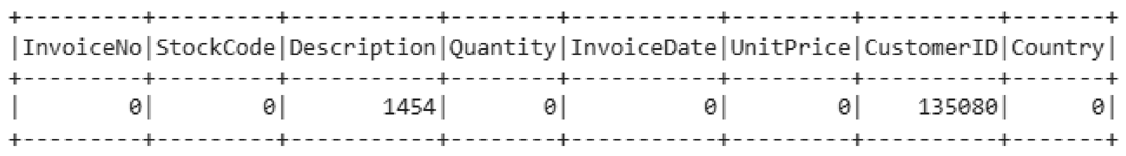
```
root
 |-- InvoiceNo: string (nullable = true)
 |-- StockCode: string (nullable = true)
 |-- Description: string (nullable = true)
 |-- Quantity: integer (nullable = true)
 |-- InvoiceDate: date (nullable = true)
 |-- UnitPrice: double (nullable = true)
 |-- CustomerID: string (nullable = true)
 |-- Country: string (nullable = true)
```

Figure 4.2: Raw schema with unmatched datatype

4.1.2 Handle missing data

First, we need to check the number of missing data in each column of the dataset.

```
# Check for missing values in original data
missing_values = df_cleaned.select([
    count(when(col(c).isNull(), c)).alias(c) for c in df_cleaned.columns
])
missing_values.show()
```



```
+---------+---------+-----------+--------+-----------+---------+----------+-------+
|InvoiceNo|StockCode|Description|Quantity|InvoiceDate|UnitPrice|CustomerID|Country|
+---------+---------+-----------+--------+-----------+---------+----------+-------+
|        0|        0|       1454|       0|          0|        0|    135080|      0|
+---------+---------+-----------+--------+-----------+---------+----------+-------+
```

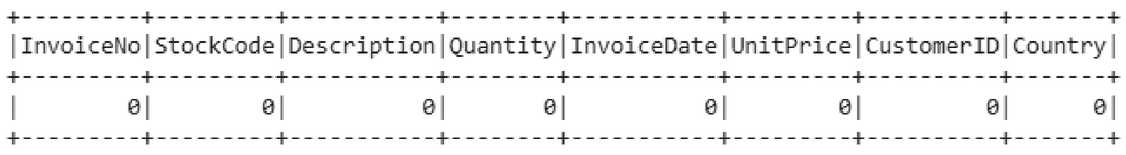
Figure 4.3: Check the number of missing values

Turns out there are 1,454 rows has missing data in the Description column, and we first remove it since our target is customer behavior, not product-detail. In addition, there are 135,080 rows in CustomerID. That is a huge amount of data, so we cannot remove it recklessly. We might turn the null value into Unknown value rather than removing them.

```
# Delete the null values in Description
df_cleaned = df_cleaned.filter(df_cleaned.Description.isNotNull())

# Change the value in CustomerID into "Unknown" for null values
df_cleaned = df_cleaned.withColumn("CustomerID",
                        when(df_cleaned.CustomerID.isNull(), "Unknown")\
                                    .otherwise(df_cleaned.CustomerID))
```

After handling process, now we re-check the number of missing data in each column to make sure the procedure worked.

```python
# Re-check for missing values in the cleaned data
missing_values = df_cleaned.select([
    count(when(col(c).isNull(), c)).alias(c) for c in df_cleaned.columns
])
missing_values.show()
```



```
+---------+---------+-----------+--------+-----------+---------+----------+-------+
|InvoiceNo|StockCode|Description|Quantity|InvoiceDate|UnitPrice|CustomerID|Country|
+---------+---------+-----------+--------+-----------+---------+----------+-------+
|        0|        0|          0|       0|          0|        0|         0|      0|
+---------+---------+-----------+--------+-----------+---------+----------+-------+
```

Figure 4.4: Re-check the number of missing value

4.1.3 Handle duplicate record

First, we check the number of duplicate records in our data:

```python
# Check the number of duplicate rows
duplicate_count = df_cleaned.count() - df_cleaned.dropDuplicates().count()
print("Number of duplicate rows: ", duplicate_count)
```

```
Number of duplicate rows: 5268
```

We identified 5,268 duplicate orders in the dataset. These were removed to prevent the model from learning redundant patterns and to ensure the integrity of the training data.

```python
# Drop duplicate rows
df_cleaned = df_cleaned.dropDuplicates()
```

After removing the duplicate orders, we rechecked the dataset to ensure no duplicate records remained.

```python
# Re-check the number of duplicate rows
duplicate_count = df_cleaned.count() - df_cleaned.dropDuplicates().count()
print("Number of duplicate rows: ", duplicate_count)
```

```
Number of duplicate rows: 0
```

4.1.4 Handle abnormal value

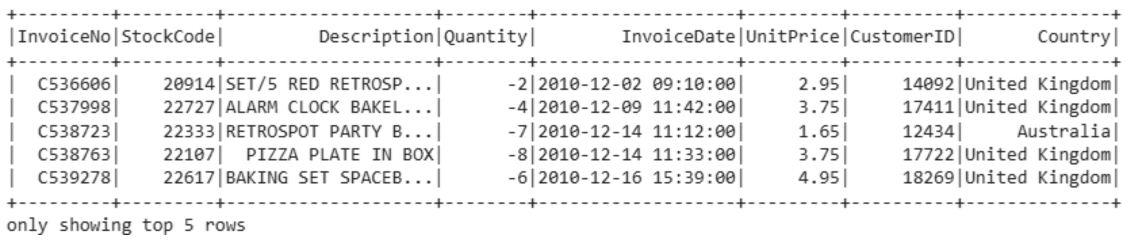
Regarding abnormal values in the dataset, we identified two main categories: cancelled orders, which are irrelevant to our project's objective, and invalid entries such as negative quantities and unit prices.

Cancelled orders

Cancelled orders are identified by invoice numbers starting with the letter 'C'. These transactions were excluded since they do not represent completed purchases and are not relevant to our analysis goals.

```python
cancel = df_cleaned.filter(col("InvoiceNo").startswith("C"))
print("Number of canceled orders: ", cancel.count())
cancel.show(5)
```

```
Number of canceled orders: 9251
```



```
+---------+---------+--------------------+--------+-------------------+---------+----------+--------------+
|InvoiceNo|StockCode|         Description|Quantity|        InvoiceDate|UnitPrice|CustomerID|       Country|
+---------+---------+--------------------+--------+-------------------+---------+----------+--------------+
|  C536606|    20914|SET/5 RED RETROSP...|      -2|2010-12-02 09:10:00|     2.95|     14092|United Kingdom|
|  C537998|    22727|ALARM CLOCK BAKEL...|      -4|2010-12-09 11:42:00|     3.75|     17411|United Kingdom|
|  C538723|    22333|RETROSPOT PARTY B...|      -7|2010-12-14 11:12:00|     1.65|     12434|     Australia|
|  C538763|    22107|  PIZZA PLATE IN BOX|      -8|2010-12-14 11:33:00|     3.75|     17722|United Kingdom|
|  C539278|    22617|BAKING SET SPACEB...|      -6|2010-12-16 15:39:00|     4.95|     18269|United Kingdom|
+---------+---------+--------------------+--------+-------------------+---------+----------+--------------+
only showing top 5 rows
```

Figure 4.5: Check the number of cancelled orders

Cancelled orders often contain negative quantities. Therefore, by removing records with negative Quantity values, we also effectively filter out most of the cancelled orders from the dataset.

Negative value in Quantity and UnitPrice

Quantity and Unit Price are numerical columns that should not have negative values in a retail dataset. Next, we checked for records with negative Quantity and Unit Price values.

```python
# Quantity has negative value
negative_Quantity = df_cleaned.filter(col("Quantity") < 0).count()
print("Number of negative Quantity: ", negative_Quantity)
df_cleaned.filter(col("Quantity") < 0).select("InvoiceNo", "Quantity").show(5)
```

```
Number of negative Quantity: 9725
```

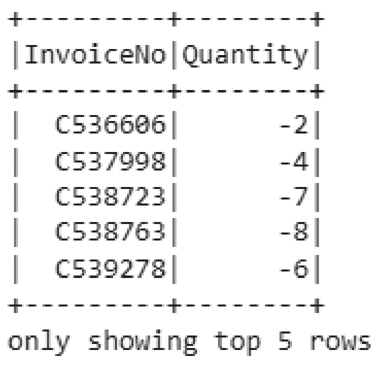

```
+---------+--------+
|InvoiceNo|Quantity|
+---------+--------+
|  C536606|      -2|
|  C537998|      -4|
|  C538723|      -7|
|  C538763|      -8|
|  C539278|      -6|
+---------+--------+
only showing top 5 rows
```

Figure 4.6: Check the number of negative quantities

```python
# UnitPrice has negative value
negative_UnitPrice = df_cleaned.filter(col("UnitPrice") < 0).count()
print("Number of negative UnitPrice: ", negative_UnitPrice)
df_cleaned.filter(col("UnitPrice") < 0).select("InvoiceNo", "UnitPrice").show(5)
```

```
Number of negative UnitPrice: 2
```

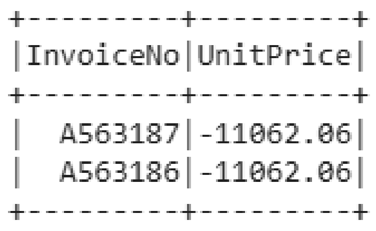

```
+---------+---------+
|InvoiceNo|UnitPrice|
+---------+---------+
|  A563187|-11062.06|
|  A563186|-11062.06|
+---------+---------+
```

Figure 4.7: Check the number of negative unit prices

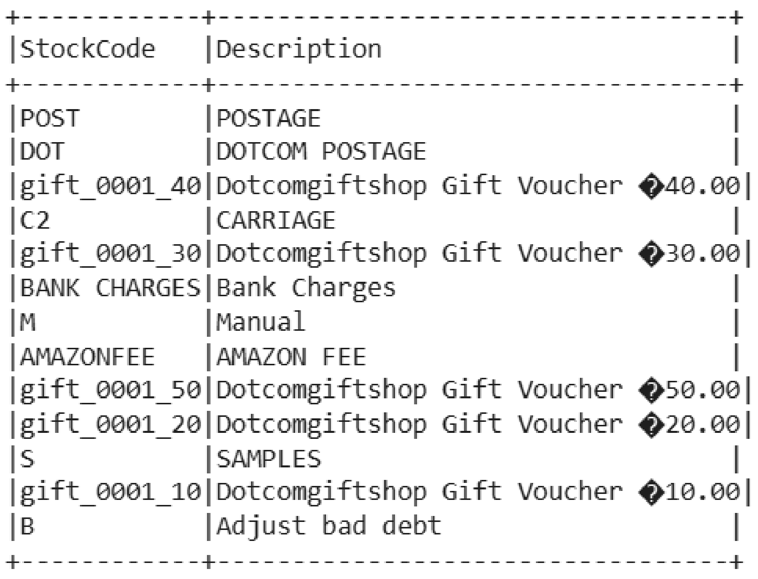
All of these invalid entries, including cancelled orders and negative value records were then removed from the dataset in a single filtering step.

```python
# Filter out negative Quantity and UnitPrice
df_cleaned = df_cleaned.filter((col("Quantity") > 0) & (col("UnitPrice") > 0))

# Check negative Quantity and UnitPrice again
print("Negative Quantity count:",
                df_cleaned.filter(col("Quantity") < 0).count())
print("Negative UnitPrice count:",
                df_cleaned.filter(col("UnitPrice") < 0).count())

# Check number of cancelled orders again
cancel = df_cleaned.filter(col("InvoiceNo").startswith("C"))
print("Number of cancelled orders: ", cancel.count())
```

```
Negative Quantity count: 0
Negative UnitPrice count: 0
Number of cancelled orders: 0
```

4.1.5 Handling Non-product Entries

The dataset also contains non-product records such as postage codes, samples, vouchers, and bank debits.

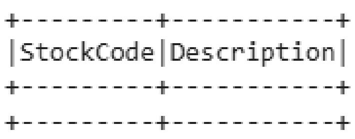


```
+------------+----------------------------------+
|StockCode   |Description                       |
+------------+----------------------------------+
|POST        |POSTAGE                           |
|DOT         |DOTCOM POSTAGE                    |
|gift_0001_40|Dotcomgiftshop Gift Voucher �40.00|
|C2          |CARRIAGE                          |
|gift_0001_30|Dotcomgiftshop Gift Voucher �30.00|
|BANK CHARGES|Bank Charges                      |
|M           |Manual                            |
|AMAZONFEE   |AMAZON FEE                        |
|gift_0001_50|Dotcomgiftshop Gift Voucher �50.00|
|gift_0001_20|Dotcomgiftshop Gift Voucher �20.00|
|S           |SAMPLES                           |
|gift_0001_10|Dotcomgiftshop Gift Voucher �10.00|
|B           |Adjust bad debt                   |
+------------+----------------------------------+
```

Figure 4.8: Table of non-product stock codes and its description

These entries were identified and removed to ensure the dataset only includes valid product transactions relevant to our analysis.

```python
df_cleaned = df_cleaned.filter(~col("StockCode")\
                        .isin(excluded_stockcodes))
# Re-check the abnormal stock code
df_excluded = df_cleaned.filter(col("StockCode").isin(excluded_stockcodes))
df_excluded.select("StockCode", "Description").distinct().show(truncate=False)
```



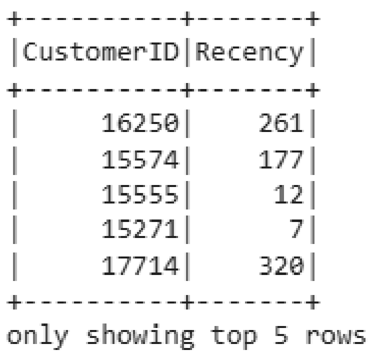
```
+---------+-----------+
|StockCode|Description|
+---------+-----------+
+---------+-----------+
```

Figure 4.9: Handle the number of the non-product items

4.2 Implementation of feature engineering

Since our project focuses on customer behavior, we prepared Recency, Frequency, and Monetary (RFM) features to enable deeper data analysis.

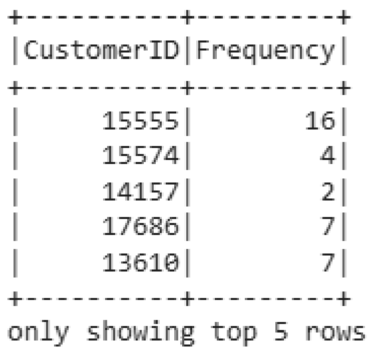
```python
# Get the max date of the dataset
max_date = df_fe.agg(max("InvoiceDate")).collect()[0][0]

# Calculate recency (days since a customer's last purchase)
recency_df = df_fe.groupBy("CustomerID").agg(
    datediff(lit(max_date), max("InvoiceDate")).alias("Recency")
)
recency_df.show(5)
```



```
+----------+-------+
|CustomerID|Recency|
+----------+-------+
|     16250|    261|
|     15574|    177|
|     15555|     12|
|     15271|      7|
|     17714|    320|
+----------+-------+
only showing top 5 rows
```

Figure 4.10: Create the recency column

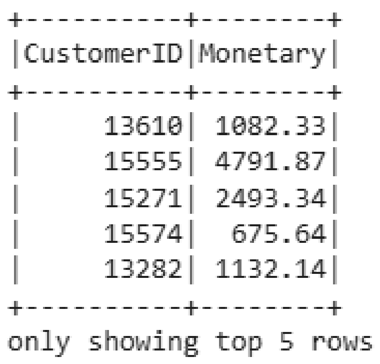
```python
# Calculate frequency (number of invoices per customer)
frequency_df = df_fe.groupBy("CustomerID").agg(
    countDistinct("InvoiceNo").alias("Frequency")
)
frequency_df.show(5)
```



```
+----------+---------+
|CustomerID|Frequency|
+----------+---------+
|     15555|       16|
|     15574|        4|
|     14157|        2|
|     17686|        7|
|     13610|        7|
+----------+---------+
only showing top 5 rows
```

Figure 4.11: Create the frequency column

```python
# Calculate Monetary (Total money spent by customer)
monetary_df = df_fe.withColumn("TotalPrice",
col("Quantity") * col("UnitPrice"))\
    .groupBy("CustomerID") \
    .agg(round(sum("TotalPrice"), 3).alias("Monetary"))
monetary_df.show(5)
```

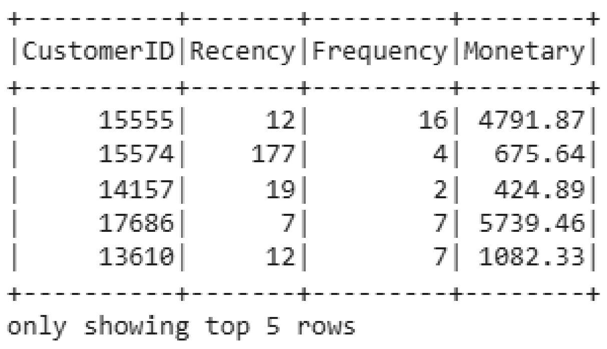


```
+----------+--------+
|CustomerID|Monetary|
+----------+--------+
|     13610| 1082.33|
|     15555| 4791.87|
|     15271| 2493.34|
|     15574|  675.64|
|     13282| 1132.14|
+----------+--------+
only showing top 5 rows
```

Figure 4.12: Create the monetary column

After computing the RFM features, we merged them into a single DataFrame for further analysis.

```
# Join all RFM features
dfs = [recency_df, frequency_df, monetary_df]
rfm_df = reduce(lambda df1, df2: df1.join(df2, "CustomerID"), dfs)
rfm_df.show(5)
```

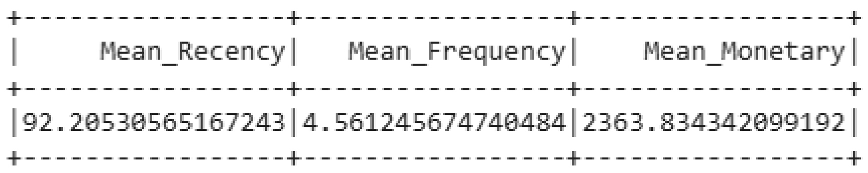


```
+----------+-------+---------+--------+
|CustomerID|Recency|Frequency|Monetary|
+----------+-------+---------+--------+
|     15555|     12|       16| 4791.87|
|     15574|    177|        4|  675.64|
|     14157|     19|        2|  424.89|
|     17686|      7|        7| 5739.46|
|     13610|     12|        7| 1082.33|
+----------+-------+---------+--------+
only showing top 5 rows
```

Figure 4.13: Join all the RFM features

To define the churn label, we first computed the mean values of the RFM (Recency, Frequency, Monetary) features to establish appropriate thresholds. Based on this analysis, customers were labeled as **churned** if they satisfied the following conditions: `Recency > 95` and `Frequency < 7`.

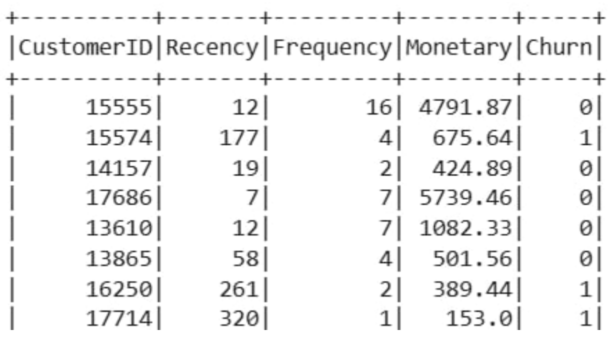
```python
from pyspark.sql.functions import mean

rfm_df.select(
    mean("Recency").alias("Mean_Recency"),
    mean("Frequency").alias("Mean_Frequency"),
    mean("Monetary").alias("Mean_Monetary")
).show()
```



```
+-----------------+-----------------+-----------------+
|     Mean_Recency|   Mean_Frequency|    Mean_Monetary|
+-----------------+-----------------+-----------------+
|92.20530565167243|4.561245674740484|2363.834342099192|
+-----------------+-----------------+-----------------+
```

Figure 4.14: Calculate mean of each RFM features

```python
# Define thresholds based on the mean values
recency_threshold = 95
frequency_threshold = 7

# Create a new column 'Churn' based on the thresholds
rfm_df = rfm_df.withColumn(
    "Churn",
    when((col("Recency") > recency_threshold) &
        (col("Frequency") <= frequency_threshold), 1).otherwise(0)
)

# Show the resulting DataFrame with Churn column
rfm_df.select("CustomerID", "Recency", "Frequency", "Monetary", "Churn").show(50)
```



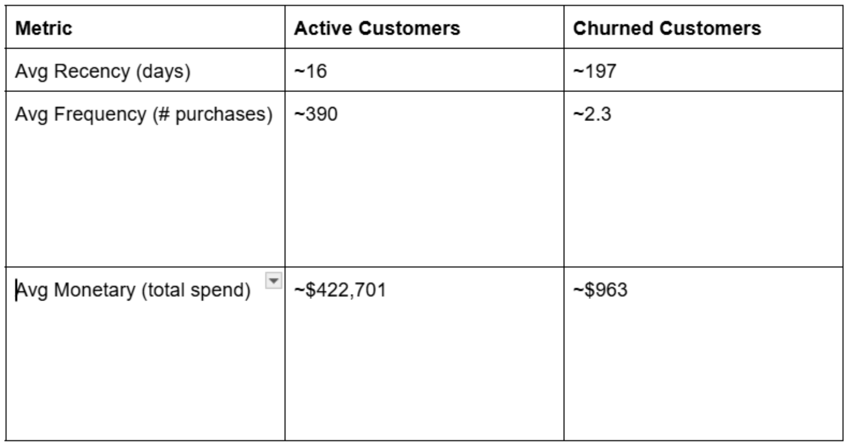
```
+----------+-------+---------+--------+-----+
|CustomerID|Recency|Frequency|Monetary|Churn|
+----------+-------+---------+--------+-----+
|     15555|     12|       16| 4791.87|    0|
|     15574|    177|        4|  675.64|    1|
|     14157|     19|        2|  424.89|    0|
|     17686|      7|        7| 5739.46|    0|
|     13610|     12|        7| 1082.33|    0|
|     13865|     58|        4|  501.56|    0|
|     16250|    261|        2|  389.44|    1|
|     17714|    320|        1|   153.0|    1|
```

Figure 4.15: Create churn label based on defined threshold

After creating the churn label, we merge it with the RFM features to form the final DataFrame. At this point, the data was ready for the next stage: data analysis.

4.3 Implementation of analyze

Firstly, we examine the key differences between customers who stayed and engaged with brand, and customers who left by breaking the RFM metrics. Our dataset has a total of 4,335 customers, in which 32 percent of them has churned.



Metric	Active Customers	Churned Customers
Avg Recency (days)	~16	~197
Avg Frequency (# purchases)	~390	~2.3
Avg Monetary (total spend)	~$422,701	~$963

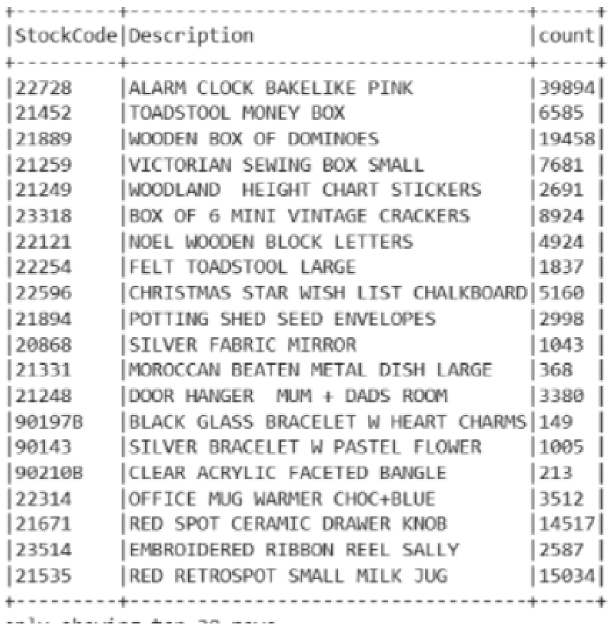
Figure 4.16: RFM Breakdown

These metrics highlighted that churned customers tend to make a single or very few purchases and then disengage. Active customers tend to make new purchases every 16 days, while churn customers tend to make an average of 2.3 purchases every 197 days. The average monetary value of active customers is higher that of churned customers by approximately **43,800%.** Furthermore, the analysis followed the 3P(s)-G framework, which explores 4 dimension: place, product, people and growth. For the product dimension, we assess the famous product lines for high recency customers- mostly churned customers to get an overview of their personas. Firstly, we examine the first dimension: product by retrieving the list of customers who have not engaged with our brand for a long time (customers with high recency value). We implemented this by pyspark.sql with the Window function to filter customers with customers with recency value on 75th percentile and 25th percentile, with customers on the 25th percentile as the new customers with low recency score, and customers on the 75th percentile as the long-time disengaged customers.

The Source Code To Retrieve Favorite Product List Of Customers With High Recency

```
from pyspark.sql.functions import col, percentile_approx, first
recency_threshold = final_df.select(
    percentile_approx("Recency", 0.75).alias("recency_75th")
).collect()[0]["recency_75th"]
high_recency_customers = final_df.filter(col("Recency") >= recency_threshold)\
    .select(['CustomerID'])
high_recency_txns = high_recency_customers.join(final_df,
                                            on="CustomerID",
                                            how="inner")
high_recency_products = high_recency_txns.groupBy("StockCode")\
```

```python
    .count() \
    .orderBy(col("count").desc())
product_desc = final_df.select("StockCode", "Description") \
    .dropna() \
    .dropDuplicates(["StockCode"])

high_recency_products_with_desc = high_recency_products.join(product_desc,
                                                              on="StockCode",
                                                              how="left")
high_recency_products_with_desc.select("StockCode", "Description", "count")\
    .show(20, truncate=False)
```



```
+---------+-----------------------------------+-----+
|StockCode|Description                        |count|
+---------+-----------------------------------+-----+
|22728    |ALARM CLOCK BAKELIKE PINK          |39894|
|21452    |TOADSTOOL MONEY BOX                |6585 |
|21889    |WOODEN BOX OF DOMINOES             |19458|
|21259    |VICTORIAN SEWING BOX SMALL         |7681 |
|21249    |WOODLAND  HEIGHT CHART STICKERS    |2691 |
|23318    |BOX OF 6 MINI VINTAGE CRACKERS     |8924 |
|22121    |NOEL WOODEN BLOCK LETTERS          |4924 |
|22254    |FELT TOADSTOOL LARGE               |1837 |
|22596    |CHRISTMAS STAR WISH LIST CHALKBOARD|5160 |
|21894    |POTTING SHED SEED ENVELOPES        |2998 |
|20868    |SILVER FABRIC MIRROR               |1043 |
|21331    |MOROCCAN BEATEN METAL DISH LARGE   |368  |
|21248    |DOOR HANGER  MUM + DADS ROOM       |3380 |
|90197B   |BLACK GLASS BRACELET W HEART CHARMS|149  |
|90143    |SILVER BRACELET W PASTEL FLOWER    |1005 |
|90210B   |CLEAR ACRYLIC FACETED BANGLE       |213  |
|22314    |OFFICE MUG WARMER CHOC+BLUE        |3512 |
|21671    |RED SPOT CERAMIC DRAWER KNOB       |14517|
|23514    |EMBROIDERED RIBBON REEL SALLY      |2587 |
|21535    |RED RETROSPOT SMALL MILK JUG       |15034|
+---------+-----------------------------------+-----+
```

Figure 4.17: Result of Retrieve Favorite Product List Of Customers With High Recency

Furthermore, the product list favored by new customers, is compared with the other list, to retrieve the key differences between the shopping behavior of new and old (highly churned customers). It is evident that while longtime disengaged customers, customers with a high recency score, show a strong interest for seasonal items, DIY-themed and personalized office decor. Many of these customers appear to be seasonal buyers, with their purchases tied to specific occasions, and only purchased for a few time. This suggest that there is an issue to retain our customers after first few time of purchasing, to understand and address this issue, we need to examine the customer service performance to determine was lacking in this part, caused our customers to leave, and collect customer feedbacks to determine if product quality is the main drive of churn customers.

The Source Code To Retrieve Favorite Product List Of Customers With Low Recency

```python
from pyspark.sql.functions import col, percentile_approx, row_number
from pyspark.sql.window import Window
```

```

recency_threshold = final_df.select(
    percentile_approx("Recency", 0.25).alias("recency_25th")
).collect()[0]["recency_25th"]

low_recency_customers = final_df.filter(col("Recency") <= recency_threshold)\
    .select(['CustomerID'])

low_recency_txns = low_recency_customers.join(final_df,
                                              on="CustomerID", how="inner")

low_recency_products = low_recency_txns.groupBy("StockCode") \
    .count() \
    .orderBy(col("count").desc())

windowSpec = Window.partitionBy("StockCode").orderBy("Description")

product_desc = final_df.select("StockCode", "Description") \
    .dropna() \
    .withColumn("row_num", row_number().over(windowSpec)) \
    .filter(col("row_num") == 1) \
    .drop("row_num")

low_recency_products_with_desc = low_recency_products.join(product_desc,
                                                            on="StockCode",
                                                            how="left")

low_recency_products_with_desc.select("StockCode", "Description", "count") \
    .show(20, truncate=False)
```

On the other hand, new customers favored child-related, and home-related decorative items. Additionally, we asses the popularity of different product categories, to retrieve sale trend by customer category.

The Source Code To Retrieve Best Selling Products

```
from pyspark.sql.functions import sum as _sum
top_products = final_df.groupBy("Description") \
    .agg(_sum("Quantity").alias("TotalQuantity")) \
    .orderBy("TotalQuantity", ascending=False)
top_products.show(10, truncate=False)
```

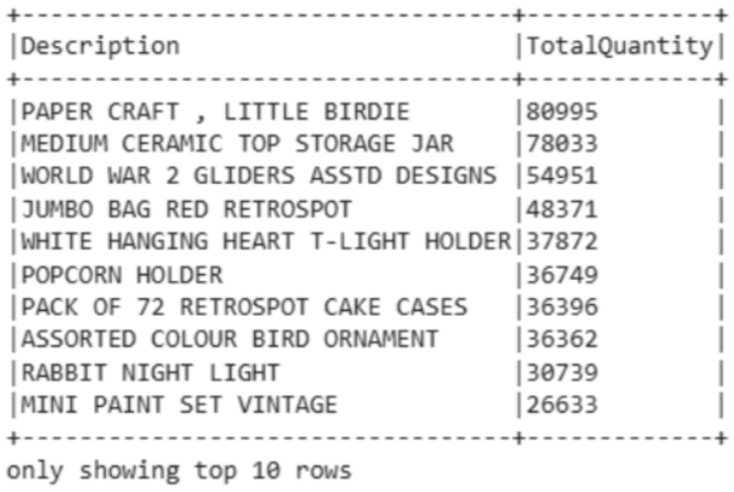

```
+-----------------------------------+------------+
|Description                        |TotalQuantity|
+-----------------------------------+------------+
|PAPER CRAFT , LITTLE BIRDIE        |80995       |
|MEDIUM CERAMIC TOP STORAGE JAR     |78033       |
|WORLD WAR 2 GLIDERS ASSTD DESIGNS  |54951       |
|JUMBO BAG RED RETROSPOT            |48371       |
|WHITE HANGING HEART T-LIGHT HOLDER|37872       |
|POPCORN HOLDER                     |36749       |
|PACK OF 72 RETROSPOT CAKE CASES    |36396       |
|ASSORTED COLOUR BIRD ORNAMENT      |36362       |
|RABBIT NIGHT LIGHT                 |30739       |
|MINI PAINT SET VINTAGE             |26633       |
+-----------------------------------+------------+
only showing top 10 rows
```

Figure 4.18: Best Selling Products

The top-selling product, Paper Craft with over 80,000 units sold, indicating strong customer interest in affordable, decorative, and possibly DIY-themed items. Other best selling items also suggest a sale trend for seasonal, and DIY-themed items.

The Source Code To Retrieve Revenue Contribution By Country

```python
from pyspark.sql.functions import col, sum as _sum

# Compute revenue per country
revenue_by_country_df = final_df.withColumn("Revenue",
    col("Quantity") * col("UnitPrice")) \
    .groupBy("Country") \
    .agg(_sum("Revenue").alias("Total_Revenue")) \
    .orderBy(col("Total_Revenue").desc())
revenue_pd = revenue_by_country_df.toPandas() import matplotlib.pyplot as plt

# Plot
plt.figure(figsize=(14, 7))
plt.bar(revenue_pd["Country"], revenue_pd["Total_Revenue"], color='skyblue')
plt.xticks(rotation=90)
plt.xlabel("Country")
plt.ylabel("Total Revenue")
plt.title("Revenue per Country")
plt.tight_layout()
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()
```

We examine the second dimension of framework: place, by breaking down the revenue by countries. The United Kingdom significantly outperforms other countries in terms of revenue, showing a strong and well-established customer base in this market. While the Netherlands, Ireland (EIRE), Germany, and France follow, their contributions are relatively minor compared to the UK. Overall, the top-performing countries are primarily concentrated within Europe, which suggest that Europe is a main contributor of our revenue. We also break down churn rate by countries, to examine if high churn rate only exists in some countries.

The Source Code To Retrieve Churn Rate By Country

```
from pyspark.sql.functions import col, countDistinct, when
churn_by_country = final_df.groupBy("Country").agg(
    countDistinct(when(col("Churn") == 1, col("CustomerID"))) \
                            .alias("Churned_Customers"),
    countDistinct(when(col("Churn") == 0, col("CustomerID"))) \
                                    .alias("Active_Customers")
)

churn_by_country = churn_by_country.withColumn(
    "Churn_Rate",
    col("Churned_Customers") / (col("Churned_Customers") \
                                + col("Active_Customers"))
)
churn_pd = churn_by_country.orderBy(col("Churn_Rate") \
                    .desc()).toPandas() import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(12,6))
sns.barplot(data=churn_pd, x="Churn_Rate", y="Country", palette="Reds_r")
plt.title("Churn Rate by Country")
plt.xlabel("Churn Rate")
plt.ylabel("Country")
plt.grid(True, axis='x', linestyle='--', alpha=0.6)
plt.tight_layout()
plt.show()
```

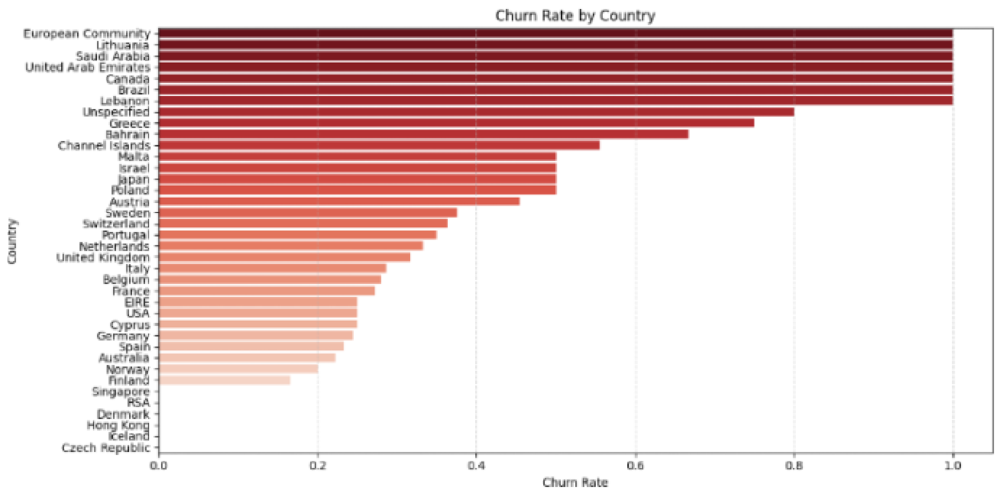


Figure 4.19: Churn Rate Breakdown By Country

United Kingdom, while being the largest market, one third of total customers are churned, suggesting a urgent need to investigate the root causes of churn customers in this country. Singapore, Saudi Arabia, and Lithuania, had missing churn rate due to missing data on either active or churned customers. Greece, Canada, and Bahrain, had small customer base (1 to 3) customers, and high churn rate, which could be because of a data entry error. Moreover, there is a category labeled Unspecified, which seems to be a data entry error. Furthermore, Finland, Norway, Germany, and Spain have mid-sized markets combined with low churn rates. These countries should be prioritized as potential growth markets. In addition, we discover the sales trend of top 3 countries revenue by plotting their sales trend.

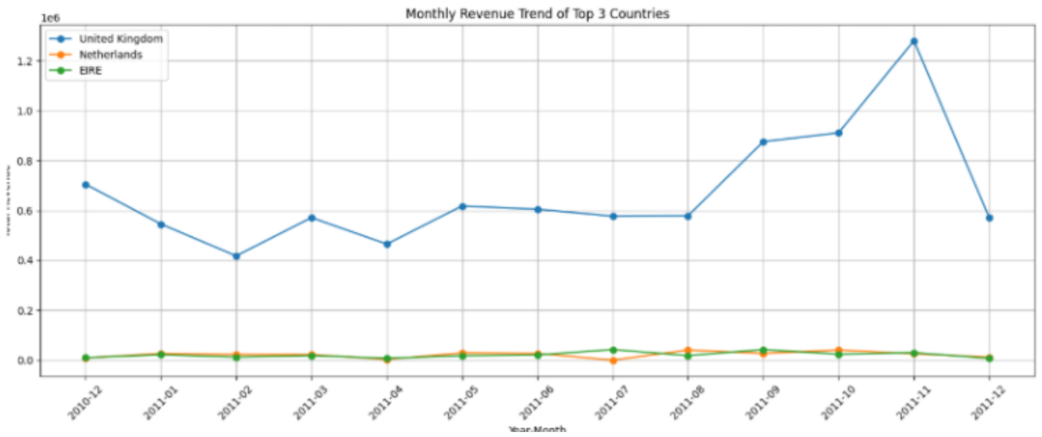


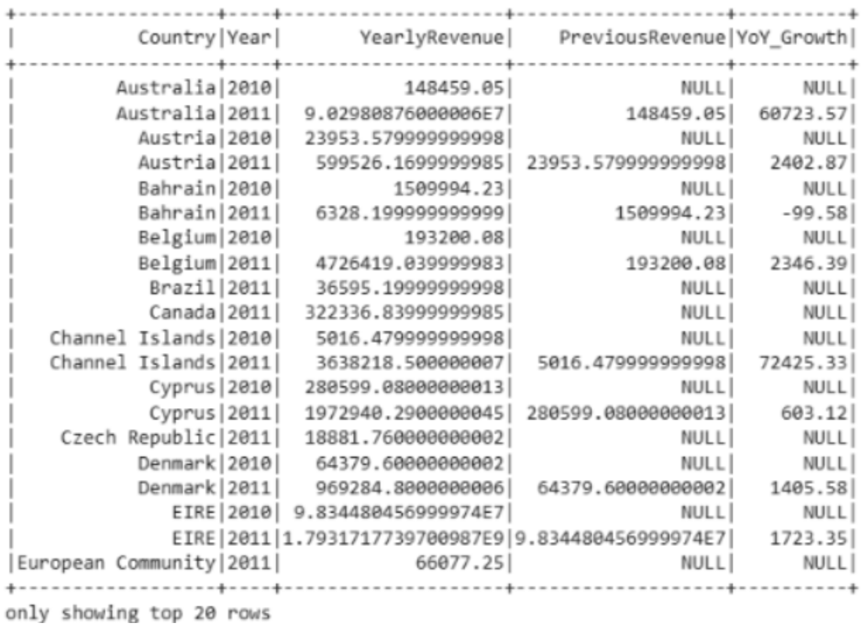
Figure 4.20: Sales Trend Of Top 3

The United Kingdom sales reached the peak in November of 2011, and then failed right after that to a half in revenue. This peak in is seasonal and highly tied with Christmas. On the other hand, Neitherland and Ireland has a slight peak in mid-year season

The next dimension, we will breakdown is growth, by examining the potential of other markets trough Year Over Year Growth metric (YoY). This metric measured the growing potential of a market by comparing the percent of growth in revenue in latest year, and previous year.

YoY Growth Source Code

```python
from pyspark.sql.functions import year, lag, round
from pyspark.sql.window import Window

df_with_year = final_df.withColumn("Year", year("InvoiceDate"))
# revenue per year
revenue_per_year = df_with_year.groupBy("Country", "Year") \
    .agg(Fsum("Monetary").alias("YearlyRevenue"))

# yoy growth
window_spec = Window.partitionBy("Country").orderBy("Year")

revenue_growth = revenue_per_year.withColumn(
    "PreviousRevenue", lag("YearlyRevenue").over(window_spec)
).withColumn(
    "YoY_Growth", round(((col("YearlyRevenue") - col("PreviousRevenue")) \
                                    / col("PreviousRevenue")) * 100, 2)
).orderBy("Country", "Year")

revenue_growth.show()
```



```
+-----------------+----+-------------------+-------------------+----------+
|          Country|Year|      YearlyRevenue|    PreviousRevenue|YoY_Growth|
+-----------------+----+-------------------+-------------------+----------+
|        Australia|2010|          148459.05|               NULL|      NULL|
|        Australia|2011|  9.02980876000006E7|          148459.05|  60723.57|
|          Austria|2010|  23953.579999999998|               NULL|      NULL|
|          Austria|2011|   599526.1699999985| 23953.579999999998|   2402.87|
|          Bahrain|2010|         1509994.23|               NULL|      NULL|
|          Bahrain|2011|   6328.199999999999|         1509994.23|    -99.58|
|          Belgium|2010|          193200.08|               NULL|      NULL|
|          Belgium|2011|   4726419.039999983|          193200.08|   2346.39|
|           Brazil|2011|   36595.19999999998|               NULL|      NULL|
|           Canada|2011|  322336.83999999985|               NULL|      NULL|
|  Channel Islands|2010|   5016.479999999998|               NULL|      NULL|
|  Channel Islands|2011|   3638218.500000007|  5016.479999999998|  72425.33|
|           Cyprus|2010|  280599.08000000013|               NULL|      NULL|
|           Cyprus|2011|  1972940.2900000045| 280599.08000000013|    603.12|
|   Czech Republic|2011|  18881.760000000002|               NULL|      NULL|
|          Denmark|2010|   64379.60000000002|               NULL|      NULL|
|          Denmark|2011|   969284.8000000006|  64379.60000000002|   1405.58|
|             EIRE|2010| 9.834480456999974E7|               NULL|      NULL|
|             EIRE|2011|1.793171773970098‌7E9|9.834480456999974E7|   1723.35|
|European Community|2011|           66077.25|               NULL|      NULL|
+-----------------+----+-------------------+-------------------+----------+
only showing top 20 rows
```

Figure 4.21: YoY Growth Source result

A key highlight is the Channel Islands, which is the fastest-growing market with an impressive year-over-year (YoY) growth rate exceeding 70 percent. Australia is the second fastest-growing market with over 60 percent YoY growth. On the other hand, the lowlight is Bahrain losing over 99 percent of revenue.

4.4 Implementation of model

This section outlines the process of implementing the K-Means clustering algorithm using PySpark on customer RFM (Recency, Frequency, Monetary) features. The implementation includes data preprocessing, feature scaling, optimal cluster selection, model fitting, cluster assignment, and result visualization.

4.4.1 Feature Selection and Preparation

Before implementing feature engineering and clustering, we must import the necessary Python and PySpark libraries.

```python
from pyspark.ml.feature import StandardScaler
from pyspark.ml.linalg import Vectors
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator

import matplotlib.pyplot as plt
import pandas as pd
from pyspark.sql import functions as F
```

The clustering analysis commenced by selecting the relevant features from the `final_df` DataFrame. Specifically, the `CustomerID`, `Recency`, `Frequency`, and `Monetary` columns were extracted to form the `rfm` DataFrame. These RFM features are commonly used in customer segmentation as they reflect behavioral patterns relevant to marketing strategies.

```python
rfm = final_df.select("CustomerID", "Recency", "Frequency", "Monetary")
```

- **Recency:** Measures the number of days since a customer's last purchase.
- **Frequency:** Counts how often a customer has made purchases in a given period.
- **Monetary:** Represents the total amount a customer has spent.

These features are commonly used in customer segmentation as they reflect behavioral patterns relevant to marketing strategies.

4.4.2 Feature Assembly and Scaling

To prepare the selected features for the K-Means algorithm, which typically expects a single vector of numerical features, a `VectorAssembler` was employed. The `VectorAssembler` transformed the individual `"Recency,"` `"Frequency,"` and `"Monetary"` columns into a single vector-type column named `"features"`. This step is essential for consolidating multiple input columns into a format compatible with Spark MLlib's machine learning algorithms.

```
assembler = VectorAssembler(
    inputCols=["Recency", "Frequency", "Monetary"],
    outputCol="features"
)
final_vector = assembler.transform(rfm)
```

Following feature assembly, the `"features"` vector was subjected to standardization using `StandardScaler`. This preprocessing step is critical for K-Means, as the algorithm is sensitive to the scale of input features. Features with larger numerical ranges can disproportionately influence the distance calculations, leading to biased clustering. The `StandardScaler` was configured with `withStd=True` to scale features to unit standard deviation and `withMean=False` to avoid centering the data around zero, preserving potential sparsity. The scaler was fitted on the assembled features and then used to transform it, resulting in the `scaled_features` column.

```
scaler = StandardScaler(
    inputCol="features",
    outputCol="scaled_features",
    withStd=True,
    withMean=False
)
scaler_model = scaler.fit(final_vector)
final_scaled = scaler_model.transform(final_vector)
```

4.4.3 Determining Optimal Number of Clusters (K)

To identify the most appropriate number of clusters (k) for the dataset, an iterative approach was adopted. A loop was executed for k values ranging from 2 to 9, with the following steps:

- A `KMeans` model was instantiated with `featuresCol="scaled_features"`, `predictionCol="cluster"`, and `seed=1` for reproducibility.
- The model was trained on the `final_scaled` DataFrame.
- Predictions were generated, assigning each data point to a cluster.
- A `ClusteringEvaluator` was used with metric `"silhouette"` to evaluate clustering quality. A higher silhouette score indicates better-defined clusters.

The following Python code illustrates this process:

```python
for k in range(2, 10):
    kmeans = KMeans(featuresCol="scaled_features",
                    predictionCol="cluster", k=k, seed=1)
    model = kmeans.fit(final_scaled)
    transformed = model.transform(final_scaled)

    evaluator = ClusteringEvaluator(
        featuresCol="scaled_features",
        predictionCol="cluster",
        metricName="silhouette"
    )
    silhouette = evaluator.evaluate(transformed)
    print(f"With k= {k} → Silhouette Score = {silhouette}")
```

Resulting Silhouette Scores:

```
With k=2 -> Silhouette Score = 0.8776
With k=3 -> Silhouette Score = 0.9418
With k=4 -> Silhouette Score = 0.8958
With k=5 -> Silhouette Score = 0.8765
With k=6 -> Silhouette Score = 0.8359
With k=7 -> Silhouette Score = 0.7935
With k=8 -> Silhouette Score = 0.7884
With k=9 -> Silhouette Score = 0.7884
```

Conclusion: The highest silhouette score was achieved with $k = 3$ (score = **0.9418**), making it the optimal number of clusters for this dataset.

4.4.4 Train Final K-Means Model

Based on the evaluation of silhouette scores from the iterative process, the optimal number of clusters was determined to be $k_{\text{best}} = 3$. A final `KMeans` model was then trained using this selected value of k, with the same parameters for `featuresCol`, `predictionCol`, and `seed` as before.

The model, denoted as `km_model`, was fitted on the standardized features (`final_scaled`). After training, it was used to transform the dataset and generate cluster predictions. The resulting DataFrame, `final_pred`, contains the original features along with an additional `"cluster"` column indicating each customer's assigned cluster.

```python
k_best = 3
kmeans = KMeans(featuresCol="scaled_features",
                predictionCol="cluster", k=k_best, seed=1)

km_model = kmeans.fit(final_scaled)
final_pred = km_model.transform(final_scaled)
final_pred.show()
```

4.5 Implementation of system

In this project, Streamlit is used to create a basic web that enables user to management the analyst of retail transaction and customer behaviours.

4.5.1 Setup streamlit and environment

First we create the directory of streamlit app.

```
os.makedirs("path_to_streamlit_app", exist_ok=True)
with open("path_to_streamlit_app/tabs/__init__.py", "w") as f:
    pass
```

Next we need to import some needed libraries to implement all function.

```
    import streamlit as st
    from pyspark.sql import SparkSession
    from pyspark.sql.functions import *
    from pyspark.sql.types import TimestampType
    from functools import reduce
    import plotly.express as px
    import pandas as pd
    import altair as alt
    from pyspark.sql.functions import (
    avg, col, countDistinct, desc, when, sum as _sum, month, year,
                                lag, round, countDistinct, percentile_approx)
    from pyspark.sql.window import Window
    from pyspark.ml.linalg import Vectors
    from pyspark.ml.feature import VectorAssembler
    from pyspark.ml.clustering import KMeansModel
    from pyspark.ml.evaluation import ClusteringEvaluator
    from pyspark.ml.feature import StandardScalerModel

```

4.5.2 Retail Data Overiew

We create the function that implement the page ”Retail Data Overiew” as the code below:

```
def show_data(df):
  st.title("Retail Transaction Dataset Summary")
  st.write("This is a .....") <- information of dataset
  with st.expander("Online Retail Data Preview", expanded=False):
    st.dataframe(df.limit(5).toPandas())
  # count the number of rows and columns in the dataset
  st.subheader("Data Dimensions")
  # FE Get the max date of the dataset
```

```
    max_date = df.agg(max("InvoiceDate")).collect()[0][0]
    st.write(f"The last retail is : {max_date}")

    # Define the table data
    data = {
      "Schema ( Variables Name)",
      "Role",
      "Type",
      "Description"
    }

    # Create a DataFrame
    df_descrip = pd.DataFrame(data)

    # Streamlit app title
    st.subheader("Key Dataset Metrics")
    # Display the table
    st.dataframe(df_descrip)

```

4.5.3 Retail Analytics

This page will present the charts that present the relationship between all factors which help to analyze retail data. This page can be implement in the pseudo code in below. Just clarify that this code will present all the result in the below section, so, we just give pseudo code to prevent redundant.

4.5.4 Customer Churn

```
def show_customer_churn(df):
    show_rfm_averages(df) # The function to show averages of dataframe
    st.markdown("### The below RMF analysis follows the 3Ps-G framework")
    show_product_analysis(df) # The function to show analysis of product aspect
    show_people_analysis(df)# The function to show analysis of people aspect
    show_place_analysis(df)# The function to show analysis of place aspect
```

4.5.5 Customer Segmentation

Algorithm 1 Show General Dashboard

```
1: procedure ShowGeneralDashboard(df)
2:     Display Title: "Overall Analysis By Other Factors"
3:     — Revenue by Country —
4:     Calculate Revenue = Quantity × UnitPrice
5:     Group by Country and sum Revenue as Total_Revenue
6:     Order by Total_Revenue descending
7:     Convert to Pandas DataFrame
8:     Plot bar chart of Revenue per Country using Plotly
9:     — Revenue per Year and Country —
10:    Extract Year from InvoiceDate
11:    Group by Country and Year, sum Monetary as YearlyRevenue
12:    Define window partitioned by Country, ordered by Year
13:    Calculate PreviousRevenue using lag function
14:    Compute YoY_Growth
15:    Order by Country and Year
16:    Display YoY Revenue Growth in a table
17:    — Top Countries Revenue Chart —
18:    Get Top N from slider
19:    Get sort order (ascending or descending)
20:    Sort and limit to Top N countries
21:    Convert to Pandas DataFrame and sort accordingly
22:    Plot horizontal bar chart of Top N countries by revenue
23:    — Monthly Revenue Trend for Top Countries —
24:    Extract Top N countries from previous step
25:    Filter original df by Top Countries
26:    Add Year and Month columns from InvoiceDate
27:    Group by Year, Month, Country, and sum Revenue
28:    Convert to Pandas DataFrame and create YearMonth column
29:    Sort by Year and Month
30:    Plot line chart of Monthly Revenue Trend using Plotly
31:    — Top Products by Quantity —
32:    Get Top N Products from slider
33:    Get sort order (ascending or descending)
34:    Group by Description and sum Quantity
35:    Order by TotalQuantity and limit to Top N
36:    Convert to Pandas and plot horizontal bar chart
37:    — Top Products by YoY Revenue Growth —
38:    Extract Year from InvoiceDate
39:    Group by StockCode, Description, Year and sum Revenue
40:    Define window partitioned by StockCode, ordered by Year
41:    Calculate PreviousYearRevenue using lag
42:    Compute YoY_Growth
43:    Filter where PreviousYearRevenue is not null
44:    Order by YoY_Growth descending and limit to top 5
45:    Display result in table
46: end procedure
```

Algorithm 2 Customer Segmentation

```
 1: function CustomerSegmentation(df)
 2:     Display title: "Customer Segmentation"
 3:     model ← LoadModel
 4:     data ← LoadClusterData(df)
 5:     if model is None or data is None then
 6:         return
 7:     end if
 8:     Apply model to data → cluster_data
 9:     Rename prediction column to cluster
10:     Evaluate clustering performance
11:     Compute silhouette score with ClusteringEvaluator
12:     Remove intermediate columns: features, scaled_features
13:     Select relevant columns and remove duplicates
14:     Display clustered data and silhouette score
15:     Visualize cluster distribution
16:     Get number of clusters from model
17:     Generate cluster labels: "Cluster 1", "Cluster 2", ...
18:     Filter data for each cluster and count entries
19:     Plot pie chart showing customer distribution per cluster
20:     Interpret cluster segments
21:     Create interpretation table with:
          • Cluster IDs
          • Labels (e.g., "Churned Low-Spenders", etc.)
          • Key Traits (e.g., purchase behavior)
22:     Display interpretation table
23: end function
```

Chapter 5
RESULT

5.1 Evaluate model

Following the training of the K-Means model and the assignment of clusters to each data point, the subsequent steps focused on analyzing the characteristics of the identified clusters and visualizing their distribution. This phase is crucial for understanding the distinct segments discovered by the clustering algorithm.

```
cluster_counts = final_pred.groupBy("cluster").count()
cluster_counts_pd = cluster_counts.toPandas()

plt.figure(figsize=(8, 6))
plt.bar(cluster_counts_pd['cluster'],
        cluster_counts_pd['count'], color='darkblue')
plt.xlabel('Cluster')
plt.ylabel('Number of Customers')
plt.title('Distribution of Customers in Each Cluster')
plt.xticks(cluster_counts_pd['cluster'])
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```

By executing the code above, we can see from 5.1 that Cluster 0 contains the majority of customers, suggesting a segment with relatively lower engagement or spending behavior. Cluster 1 represents a moderately sized group, likely comprising customers with average purchase activity. Cluster 2, though the smallest in number, appears to consist of the most valuable customers based on higher Recency, Frequency, and Monetary scores.

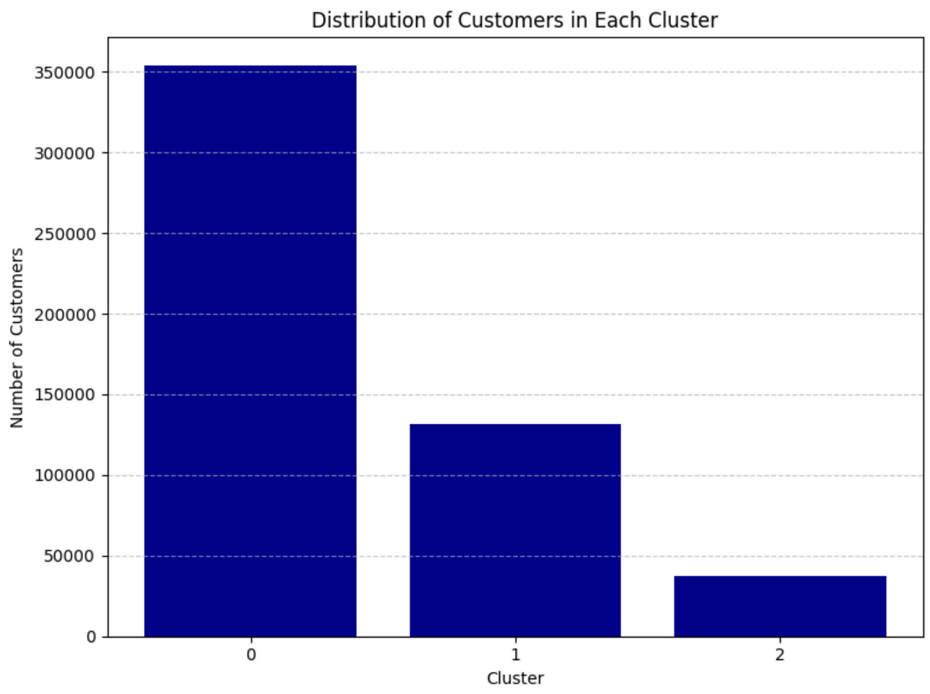


Figure 5.1: Distribution of customers in each cluster

This distribution provides actionable insight: businesses should consider prioritizing Cluster 2 for loyalty or reward programs due to their high value, while Cluster 1 may benefit from tailored engagement strategies. Meanwhile, efforts to reactivate or re-engage customers in Cluster 0 could be valuable for improving overall customer retention.

The cluster centers were extracted from the trained K-Means model using the following code:

```python
centers = km_model.clusterCenters()
print("Cluster Centers:")
for i, c in enumerate(centers):
    print(f"Cluster {i}: {c}")
```

The output cluster centers (scaled features) are:

Table 5.1: Cluster Centers (Scaled Features)

Cluster	**Recency**	**Frequency**	**Monetary**
0	0.3741	0.0345	0.0189
1	0.0000	2.3333	2.3203
2	3.6180	0.0046	0.0016

- **Cluster 0:**
  - Moderate **Recency** value (approximately 0.37) indicates customers purchased somewhat recently but not very recently.

  - Very low **Frequency** (around 0.03) and **Monetary** (around 0.02) values suggest these customers make purchases infrequently and spend relatively little.
  - These could be occasional buyers or low engagement customers.

- **Cluster 1:**
  - Near-zero **Recency** means these customers bought very recently.
  - Very high **Frequency** (around 2.33) and **Monetary** (around 2.32) values indicate frequent and high-value purchases.
  - This cluster likely represents the **best customers**: loyal, frequent, and high spenders.

- **Cluster 2:**
  - Very high **Recency** (approximately 3.62) shows these customers haven't purchased for a long time.
  - Extremely low **Frequency** (around 0.005) and **Monetary** (around 0.002) values suggest they rarely purchase and spend very little.
  - This cluster is probably made up of **inactive or churned customers** who may need re-engagement efforts.

To gain a deeper understanding of each customer segment formed by the K-Means clustering algorithm, statistical summaries of the original RFM features—Recency, Frequency, and Monetary—were computed for each cluster. This analysis involved calculating the minimum, mean, median, and maximum values for each feature within every cluster.

```python
features = ["Recency", "Frequency", "Monetary"]
agg_exprs = []
for col in features:
    agg_exprs.extend([
        F.min(col).alias(f"{col}_min"),
        F.avg(col).alias(f"{col}_mean"),
        F.expr(f'percentile({col}, 0.5)').alias(f"{col}_median"),
        F.max(col).alias(f"{col}_max")
    ])

summary_df = final_pred.groupBy("cluster").agg(*agg_exprs)
summary_df.orderBy("cluster").show(truncate=False)
```

After run the code we get a Table 5.2 presents descriptive statistics for each cluster shows distinct behavioral patterns.

Table 5.2: Cluster Summary Statistics (Recency, Frequency, Monetary)

Metric (Statistic)	Cluster 0	Cluster 1	Cluster 2
Recency (Min)	0	0	119
Recency (Mean)	22.26	0.00	215.31
Recency (Median)	12.0	0.0	196.0
Recency (Max)	117	0	373
Frequency (Min)	1	1371	1
Frequency (Mean)	20.30	1371.0	2.73
Frequency (Median)	8.0	1371.0	2.0
Frequency (Max)	206	1371	34
Monetary (Min)	5.90	1509994.23	3.75
Monetary (Mean)	12276.93	1509994.23	1018.38
Monetary (Median)	2993.10	1509994.23	610.01
Monetary (Max)	279138.02	1509994.23	77183.60

The overall summary in Table 5.2 highlights key differences in customer behavior across clusters. To gain a deeper understanding, we examine each cluster individually—starting with Cluster 0. **Cluster 0:**

Table 5.3: Cluster 0 Summary Statistics and Center (Scaled Features)

Metric	Min	Mean	Median	Max	Center (Scaled)
Recency	0	22.33	12.00	119.00	0.38
Frequency	1	20.31	8.00	206.00	0.03
Monetary	30	12,288.85	2,984.49	279,138.02	0.02

From the Table 5.3, we could draw the following insights:

- **Low Recency:** Customers purchased recently, with a median of **12 days** ago and an average around **23 days**.

- **Medium Frequency:** Average of **20 purchases** with some customers buying up to **206 times**.

- **Monetary:** Moderate spending, averaging about **12,000** with a median of **3,000**, and some spending as high as **279,000**.

Interpretation: *Active Medium-Value Customers* — regular customers with decent purchase frequency and moderate spending.

While Cluster 0 represents active, mid-value customers with steady engagement, Cluster 1 displays a very different pattern. **Cluster 1:**

Table 5.4: Cluster 1 Summary Statistics and Center (Scaled Features)

Metric	Min	Mean	Median	Max	Center (Scaled)
Recency	0.00	0.00	0.00	0.00	0.00
Frequency	1371.0	1371.00	1371.00	1371.00	2.33
Monetary	1,500,000	1,500,000	1,500,000	1,500,000	2.32

From the Table 5.4, we got:

- **Recency = 0:** Purchased very recently (today).
- **Extremely High Frequency:** Exactly 1371 purchases.
- **Extremely High Monetary:** Spending exceeding 1.5 million.

Interpretation: *Ultra VIP Customers* — a rare but critical segment representing loyal, frequent, and very high-value customers.

In contrast to the high-value, highly engaged customers in Cluster 1, Cluster 2 reveals a group with minimal activity. **Cluster 2:**

Table 5.5: Cluster 2 Summary Statistics and Center (Scaled Features)

Metric	Min	Mean	Median	Max	Center (Scaled)
Recency	120	215.53	196.00	373.00	3.63
Frequency	1	2.74	2.00	34.00	0.00
Monetary	3.75	1,032.45	609.40	44,534.30	0.00

From the Table 5.5, we could see that:

- **High Recency:** Customers haven't purchased for a long time, with recency ranging from **120 to 373 days**, average **215 days**.
- **Low Frequency:** Very infrequent purchasers, averaging **2-3 purchases**.
- **Low Monetary:** Low average spending around **1,000**, but with some variance.

Interpretation: *Churned Low-Value Customers* — customers likely inactive or churned, with low engagement and spending.

Table 5.6: Cluster Labels and Key Traits

Cluster	Label	Key Traits
0	Active Medium-Value Customers	Regular, recent purchasers with moderate to high spending.
1	Ultra VIP Customers	Purchased very recently, with extremely high purchase frequency and monetary value.
2	Churned Low-Value Customers	Long inactive customers with rare purchases and low spending.

Customer segmentation groups customers based on their buying behavior, such as how recently and frequently they purchase and how much they spend. This helps businesses target each group effectively: loyal, high-value customers receive rewards and personalized offers to keep them engaged, while inactive or low-value customers are reached with win-back campaigns. Segmentation also helps prioritize marketing resources and tailor products to meet customer needs. Overall, it improves customer experience, reduces churn, and increases profitability by enabling focused and relevant marketing actions.

5.2 System Result

The results of data processing, analysis, and clustering have been integrated into a user-friendly application developed using Streamlit. This application serves as an interactive platform that visualizes all implemented functionalities discussed in the previous chapter.

As outlined in Figure 5.2, the application is organized into multiple tabs, each dedicated to a specific analytical aspect. This modular design facilitates ease of navigation and allows users to focus on distinct dimensions of the dataset. Users can interact with each section to explore independent themes without being overwhelmed by extraneous information.

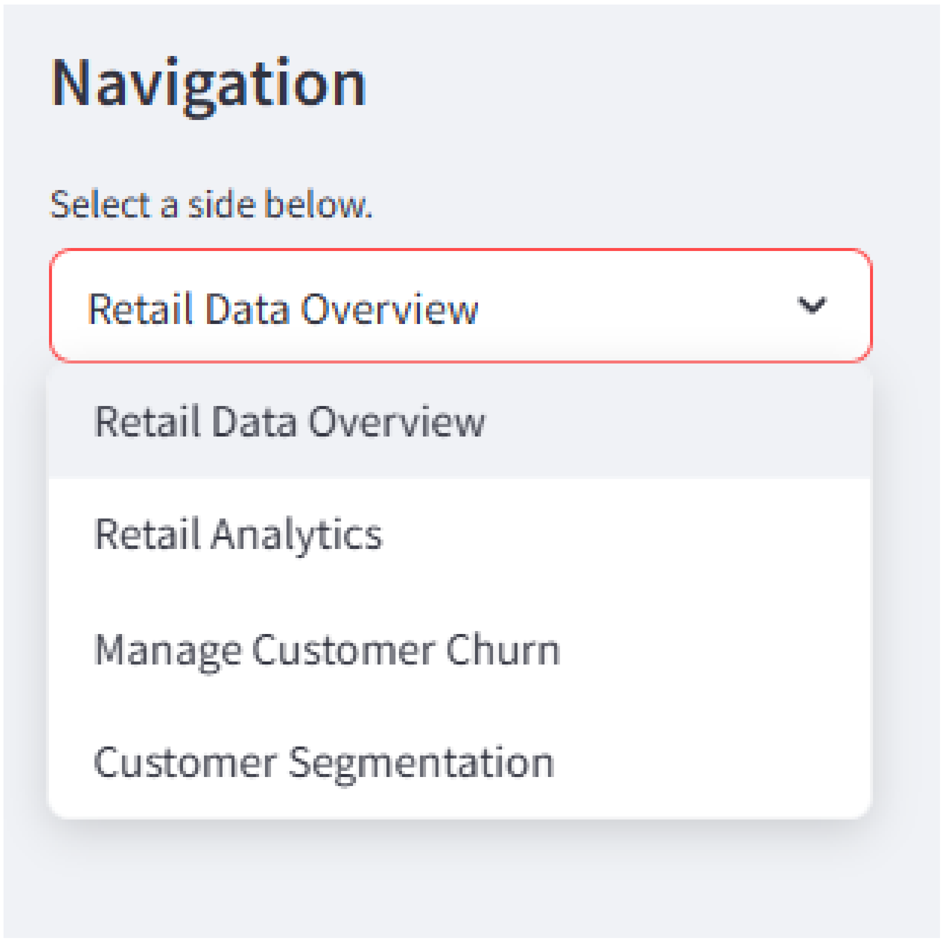
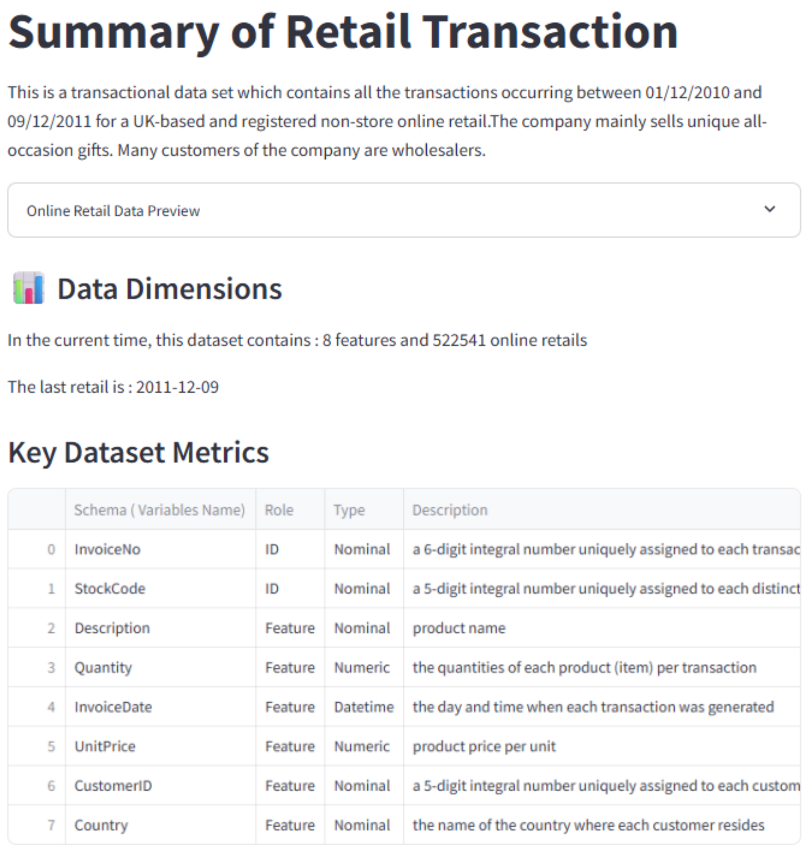


Figure 5.2: Side bar

The first tab (in Figure 5.3) presents a general overview of the dataset, which has been cleaned and preprocessed using Apache Spark. This section provides insight into the structure, composition, and content of the dataset. Users can examine what data are included and gain an understanding of the dataset's scope and characteristics.



Summary of Retail Transaction

This is a transactional data set which contains all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail.The company mainly sells unique all-occasion gifts. Many customers of the company are wholesalers.

Online Retail Data Preview

📊 Data Dimensions

In the current time, this dataset contains : 8 features and 522541 online retails

The last retail is : 2011-12-09

Key Dataset Metrics

	Schema (Variables Name)	Role	Type	Description
0	InvoiceNo	ID	Nominal	a 6-digit integral number uniquely assigned to each transac
1	StockCode	ID	Nominal	a 5-digit integral number uniquely assigned to each distinct
2	Description	Feature	Nominal	product name
3	Quantity	Feature	Numeric	the quantities of each product (item) per transaction
4	InvoiceDate	Feature	Datetime	the day and time when each transaction was generated
5	UnitPrice	Feature	Numeric	product price per unit
6	CustomerID	Feature	Nominal	a 5-digit integral number uniquely assigned to each custom
7	Country	Feature	Nominal	the name of the country where each customer resides

Figure 5.3: Retail data overview

The second tab visualizes various factors influencing transactional data. Through a series of interactive charts, relationships between key variables are illustrated. For instance, Figure 5.4 displays the correlation between revenue and country, offering insights into how geographical location affects revenue generation.

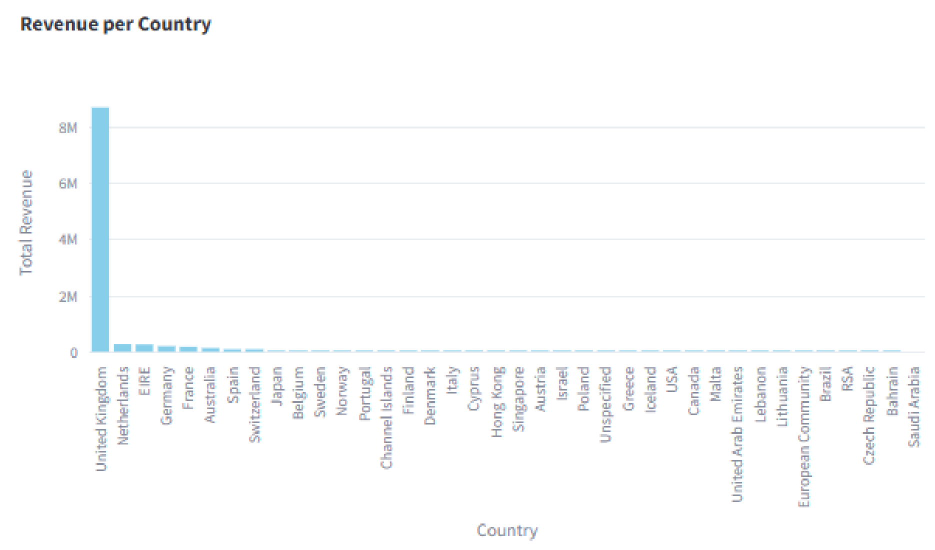


Figure 5.4: Bar chart present relationship between Revenue and Country

The Figure 5.4 show the relationship between revenue and country, it help user to see easy the relationship and effect of revenue per each country.

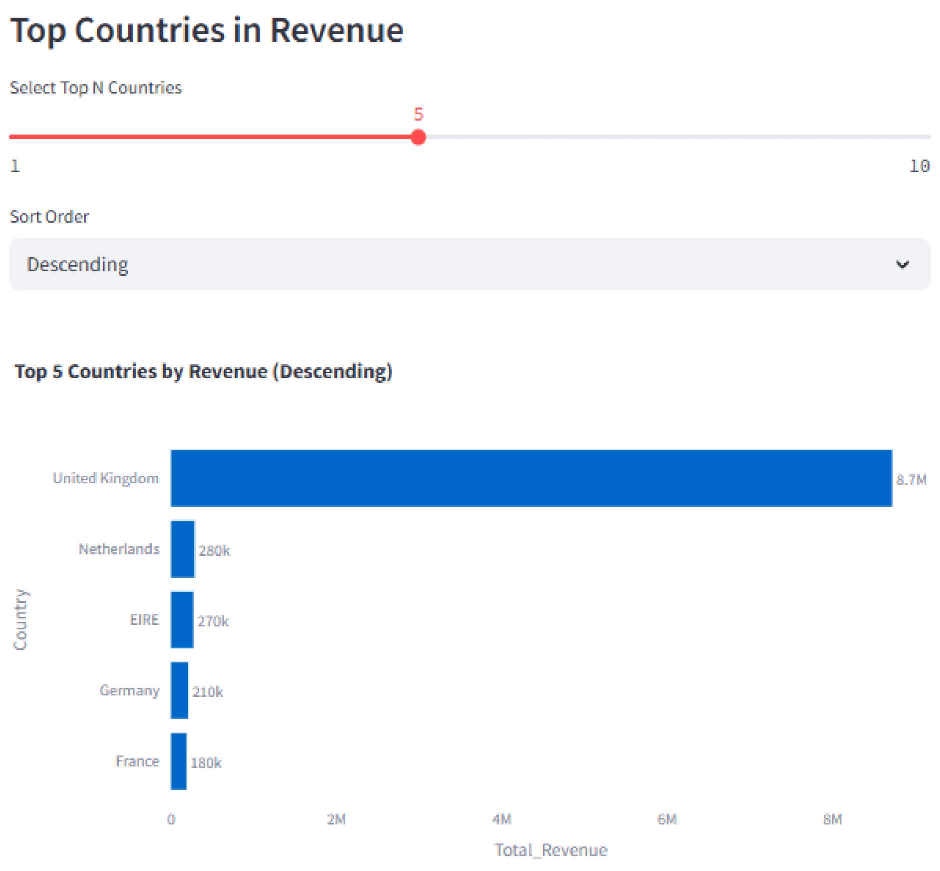
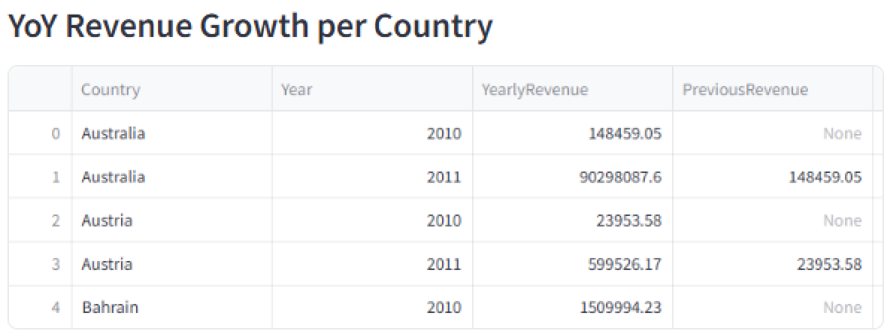


Figure 5.5: The rate Revenue by country

Similarly, Figure 5.5 highlights countries with the highest and lowest revenue contributions, enabling users to identify geographic areas of opportunity or concern.



YoY Revenue Growth per Country

	Country	Year	YearlyRevenue	PreviousRevenue
0	Australia	2010	148459.05	None
1	Australia	2011	90298087.6	148459.05
2	Austria	2010	23953.58	None
3	Austria	2011	599526.17	23953.58
4	Bahrain	2010	1509994.23	None

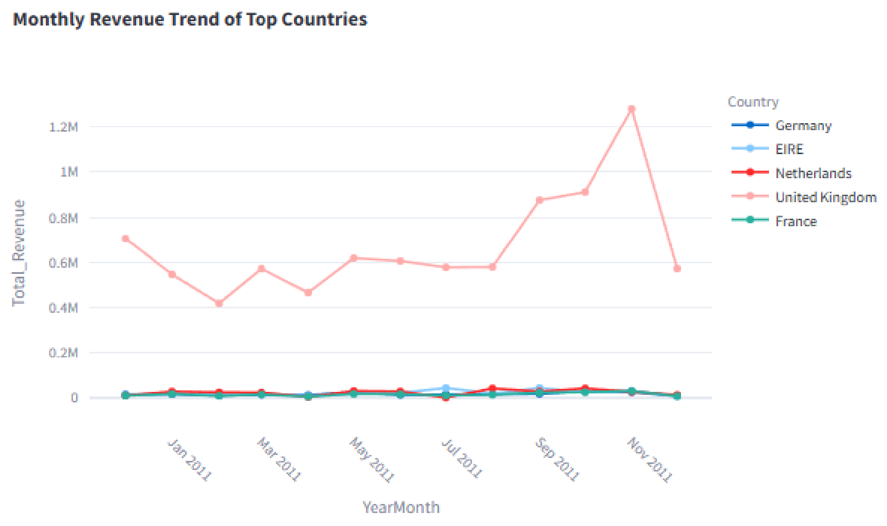


Figure 5.6: The YoY Grow chart and Monthly Revenue of country

Additional visualizations depict year-over-year (YoY) revenue growth and monthly revenue trends across different countries.

Top Products by Quantity

Top N Products by Quantity

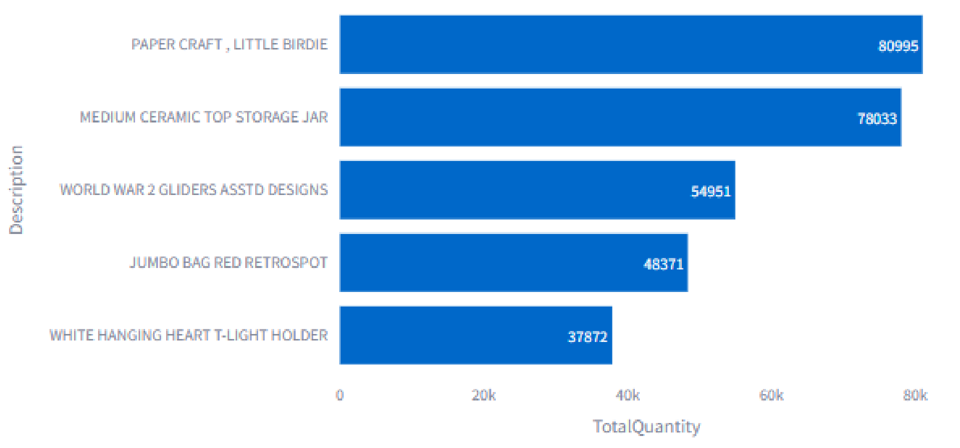
5

1 10

Product Sort Order

Descending

Top 5 Products by Quantity (Descending)



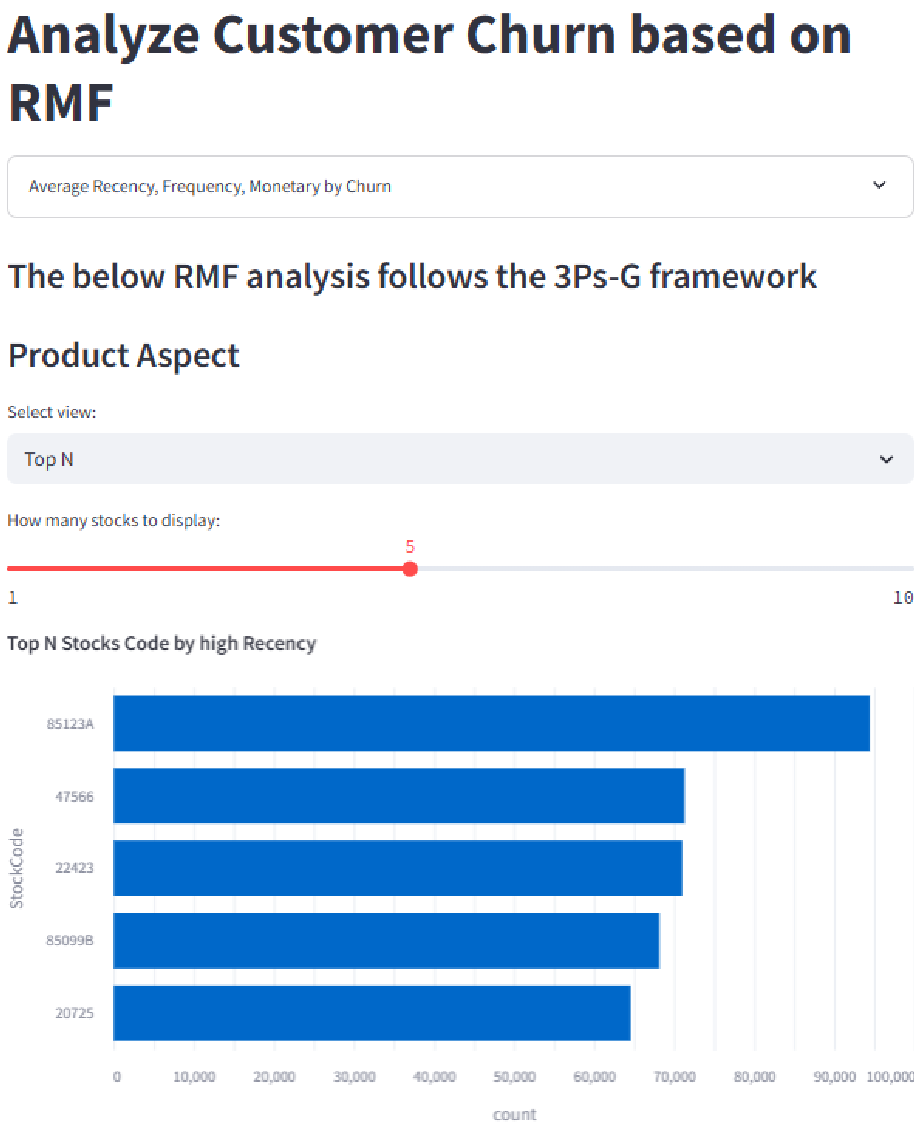
Products with Highest YoY Revenue Growth

	StockCode	Description	Year	YearlyRevenue	PreviousYearRevenue	YoY_Gr
0	22148	EASTER CRAFT 4 CHICKS	2011	3913.24	4.21	92851.
1	75049L	LARGE CIRCULAR MIRROR MOBILE	2011	6805.45	7.53	9027
2	22698	PINK REGENCY TEACUP AND SAUCER	2011	17424.9	20.66	84241.
3	21090	SET/6 COLLAGE PAPER PLATES	2011	1272.49	1.7	74752.
4	22952	60 CAKE CASES VINTAGE CHRISTMAS	2011	8616.03	13.2	65172.

Figure 5.7: The relationship between Products and Quantity

In terms of product performance, Figure 5.7 illustrates the relationship between product types and sales volume, providing a foundation for product-based strategy development.

The third tab focuses on customer churn analysis, employing the RMF (Recency, Monetary, Frequency) model in combination with the 3Ps-G metric. This section allows business managers to identify the primary factors influencing customer attrition without needing to examine large volumes of raw transaction data. The analysis is further divided into three perspectives: product, people, and place. Each sub-section provides targeted visualizations that facilitate the interpretation of churn-related patterns. For example, managers can pinpoint which products are most associated with churn or understand which customer segments and geographic locations exhibit higher churn rates.


Analyze Customer Churn based on RMF

Average Recency, Frequency, Monetary by Churn

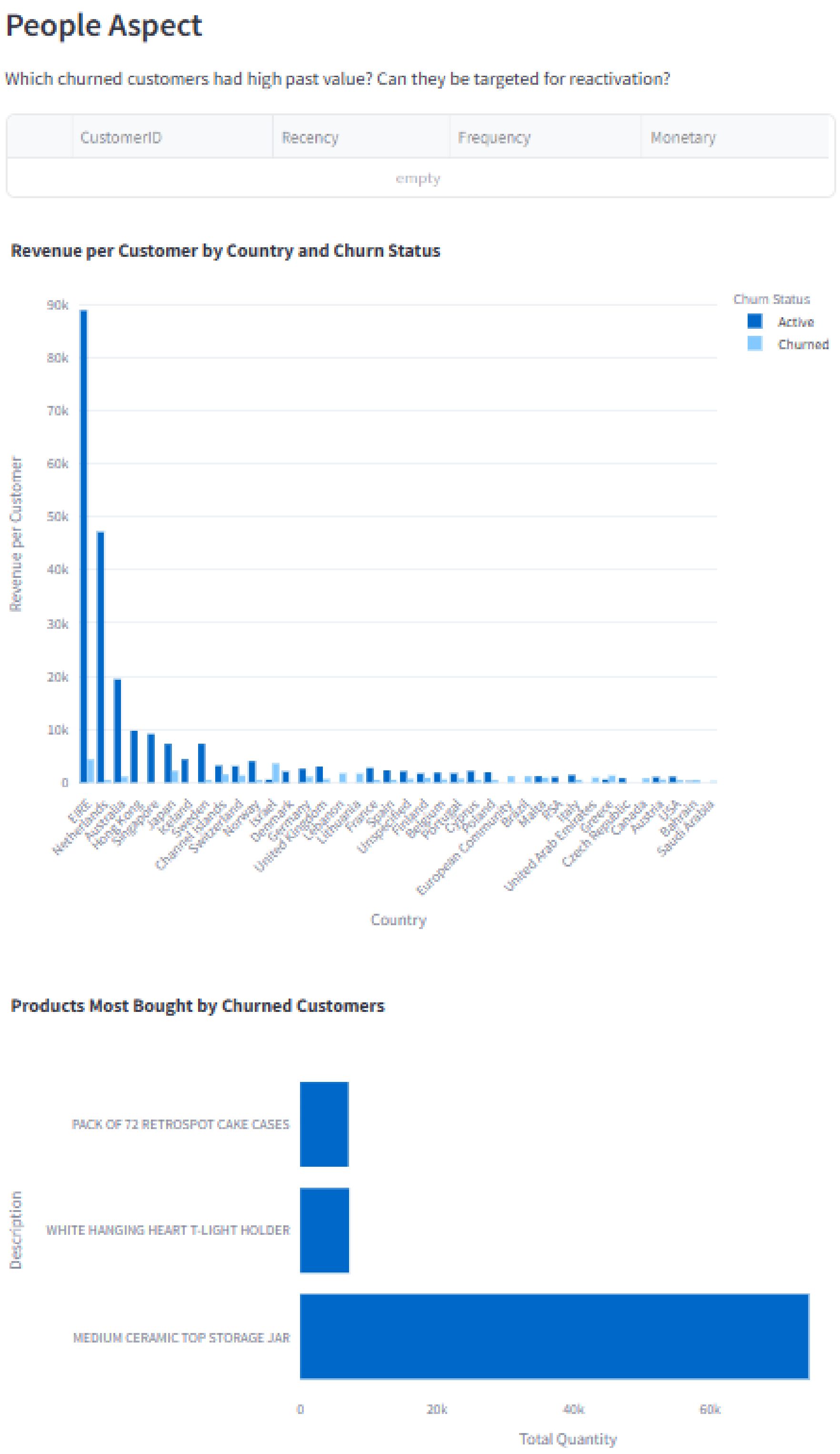
The below RMF analysis follows the 3Ps-G framework

Product Aspect

Select view: Top N

How many stocks to display: 5 (1–10)

Figure 5.8: Analyzing customer churn through product aspect



People Aspect

Which churned customers had high past value? Can they be targeted for reactivation?

	CustomerID	Recency	Frequency	Monetary
		empty		

Revenue per Customer by Country and Churn Status

Products Most Bought by Churned Customers

Description
PACK OF 72 RETROSPOT CAKE CASES
WHITE HANGING HEART T-LIGHT HOLDER
MEDIUM CERAMIC TOP STORAGE JAR
0
20k
40k
60k
Total Quantity

Figure 5.9: Analyzing customer churn through people aspect

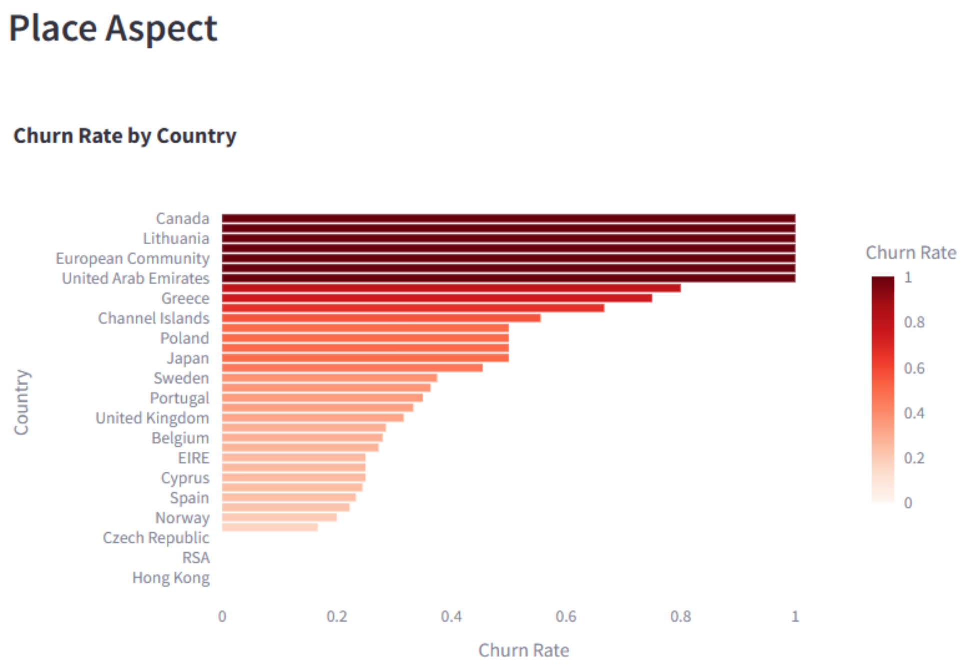
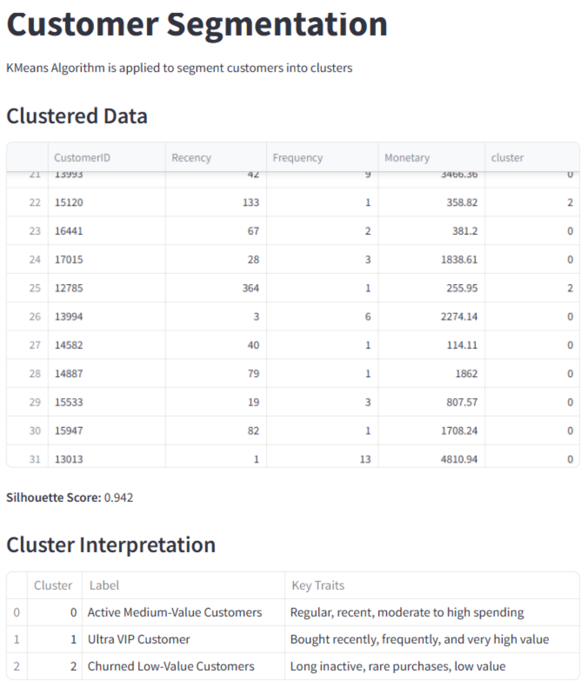


Figure 5.10: Analyzing customer churn through place aspect

The final tab features customer segmentation based on clustering techniques. Customers are automatically grouped into distinct clusters, and each group is assigned a label based on shared characteristics. Detailed descriptions of each cluster are provided, including average scores and defining features. This segmentation supports the development of targeted marketing strategies. For example, groups identified as having low value can be targeted with promotional offers or discounts to encourage re-engagement. The clustering results are visualized and include summary statistics that make the segmentation process transparent and interpretable.



Customer Segmentation

KMeans Algorithm is applied to segment customers into clusters

Clustered Data

	CustomerID	Recency	Frequency	Monetary	cluster
21	13993	42	9	3466.36	0
22	15120	133	1	358.82	2
23	16441	67	2	381.2	0
24	17015	28	3	1838.61	0
25	12785	364	1	255.95	2
26	13994	3	6	2274.14	0
27	14582	40	1	114.11	0
28	14887	79	1	1862	0
29	15533	19	3	807.57	0
30	15947	82	1	1708.24	0
31	13013	1	13	4810.94	0

Silhouette Score: 0.942

Cluster Interpretation

	Cluster	Label	Key Traits
0	0	Active Medium-Value Customers	Regular, recent, moderate to high spending
1	1	Ultra VIP Customer	Bought recently, frequently, and very high value
2	2	Churned Low-Value Customers	Long inactive, rare purchases, low value

Figure 5.11: The result of KMeans model & label of clusters

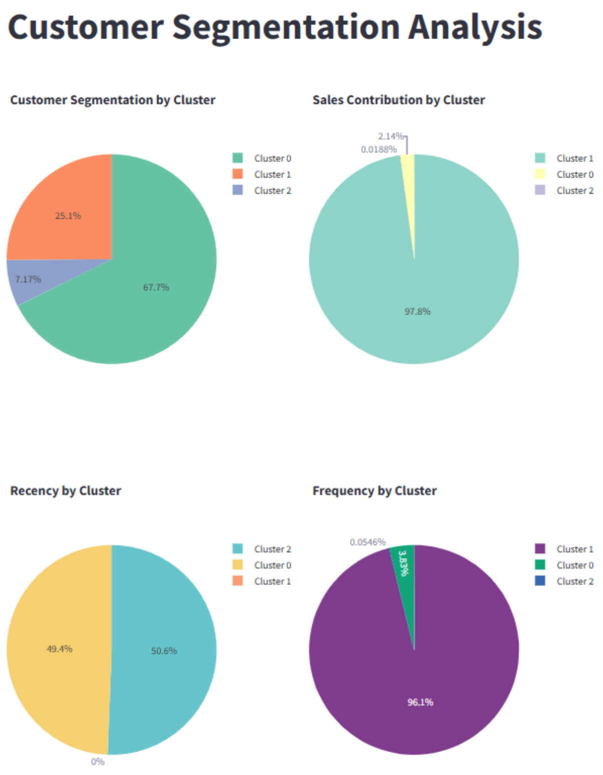


Figure 5.12: Visualization customer behaviour through many aspects

In summary, the developed application functions as an interactive dashboard that synthesizes the results of the entire project. It transforms complex transaction data into an accessible format, enabling stakeholders to extract insights efficiently and make data-driven decisions.

Chapter 6
CONCLUSION

6.1 Conclusion

In conclusion, this project has demonstrated the practical application of various data processing and analytical techniques within the context of a real-world business scenario. By working with large-scale transaction data, the project showcased the capabilities of Apache Spark and its associated libraries, particularly Spark SQL, for efficient big data processing.

Spark SQL was employed to manage, clean, and analyze extensive transactional datasets, highlighting its strength in handling complex queries across distributed systems. The analysis encompassed multiple dimensions, including key factors influencing transaction behavior and customer churn. The ability to process and analyze such data at scale underscores the effectiveness of Spark in addressing modern business intelligence challenges.

In addition, customer segmentation was achieved through clustering techniques implemented directly within the Spark ecosystem. This enabled the identification of distinct customer groups based on behavioral metrics, which is critical for developing targeted marketing and retention strategies.

The analytical process culminated in the development of an interactive web application, designed using Streamlit, to present the results in a clear and accessible format. This application transforms complex data into actionable insights through intuitive visualizations and interactive features, thereby empowering users to make informed business decisions without the need to sift through vast amounts of raw data.

Overall, the project illustrates the integration of big data tools, statistical modeling, and user-centric design to create a comprehensive data analytics solution. It bridges theoretical knowledge with practical implementation, offering valuable experience in end-to-end data-driven application development.

6.2 Limitations & Future work

While this project successfully demonstrates the application of Apache Spark for batch processing of large-scale transactional data, several limitations remain. The current scope is primarily focused on static data analysis using Spark SQL and clustering techniques. As such, more advanced functionalities of the Spark ecosystem, such as Spark Streaming for real-time data processing and structured streaming architectures, have not been explored.

Additionally, the system is designed for offline analysis and does not yet incorporate mechanisms for handling continuous data flows or real-time decision-making. Integrating streaming capabilities would allow the application to support real-time monitoring, anomaly detection, and dynamic customer behavior tracking—key requirements for modern data-driven businesses.

Future work could address these limitations by extending the architecture to include real-time data ingestion and analysis, possibly integrating with message brokers such as Apache Kafka. Moreover, enhancements in model deployment, performance optimization, and user experience design would further increase the system's applicability in operational environments.

References

1. Streamlit Docs. (n.d.). https://docs.streamlit.io/

2. Streamlit – Posit Connect Documentation Version 2025.04.0. (n.d.). https://docs.posit.co/connect/user/streamlit/

3. Merced, A. (2024, September 23). Deep Dive into Data Apps with Streamlit. DEV Community. https://dev.to/alexmercedcoder/deep-dive-into-data-apps-with-streamlit-3e40

4. Sharma, P. (2024, October 17). Deploying machine learning models using Streamlit – An introductory guide to Model Deployment. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/12/deploying-machine-learning-models-using-streamlit-an-introductory-guide-to-model-deployment

5. Opendatasoft. (n.d.). *Dataset schema – Definition*. Retrieved from `https://www.opendatasoft.com/en/glossary/dataset-schema/`

6. Wikipedia contributors. (n.d.). *Missing data*. Wikipedia. Retrieved from `https://en.wikipedia.org/wiki/Missing_data`

7. Müller, A. C., & Guido, S. (2017). *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, pp. 168–181.

8. Arthur, D., & Vassilvitskii, S. (2007). k-means++: The Advantages of Careful Seeding. *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, pp. 1027–1035.

9. Jain, A. K. (2010). Data Clustering: 50 Years Beyond K-Means. *Pattern Recognition Letters*, 31(8), 651–666.

10. Rousseeuw, P. J. (1987). Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *Journal of Computational and Applied Mathematics*, 20, 53–65.