Student's Full Name: Meghana Maringanti, Li Mi
Team Name: Data Duo
Course Number and Title: INFO 531- Data Warehousing and Analytics in the Cloud
Term and year: Spring 2025
Submission Week: Week 16 Final Project
Instructor's Name: Nayem Rahman
Date of Submission: 15/05/2025

---

1. Data Preparation

The data used in this project comes from "<source>". We created the database "OLIST" in a Snowflake data warehouse based on the data. The preparation phase involved creating tables in the "OLIST" database, loading data into them, and then securely connecting to Snowflake using SQLAlchemy using a Python script in Jupyter Notebook, fetching data from eight interrelated tables, cleaning each dataset, and merging them into a single, unified DataFrame suitable for unsupervised machine learning.

1.1 Data Sources

1. We downloaded the data required for the project from the link: [Brazilian E-Commerce Public Dataset by Olist]

2. For this project, we used the Snowflake environment and created a database named "OLIST"

3. Then the following tables were created, and the data downloaded is loaded into the Snowflake `OLIST.PUBLIC` schema:

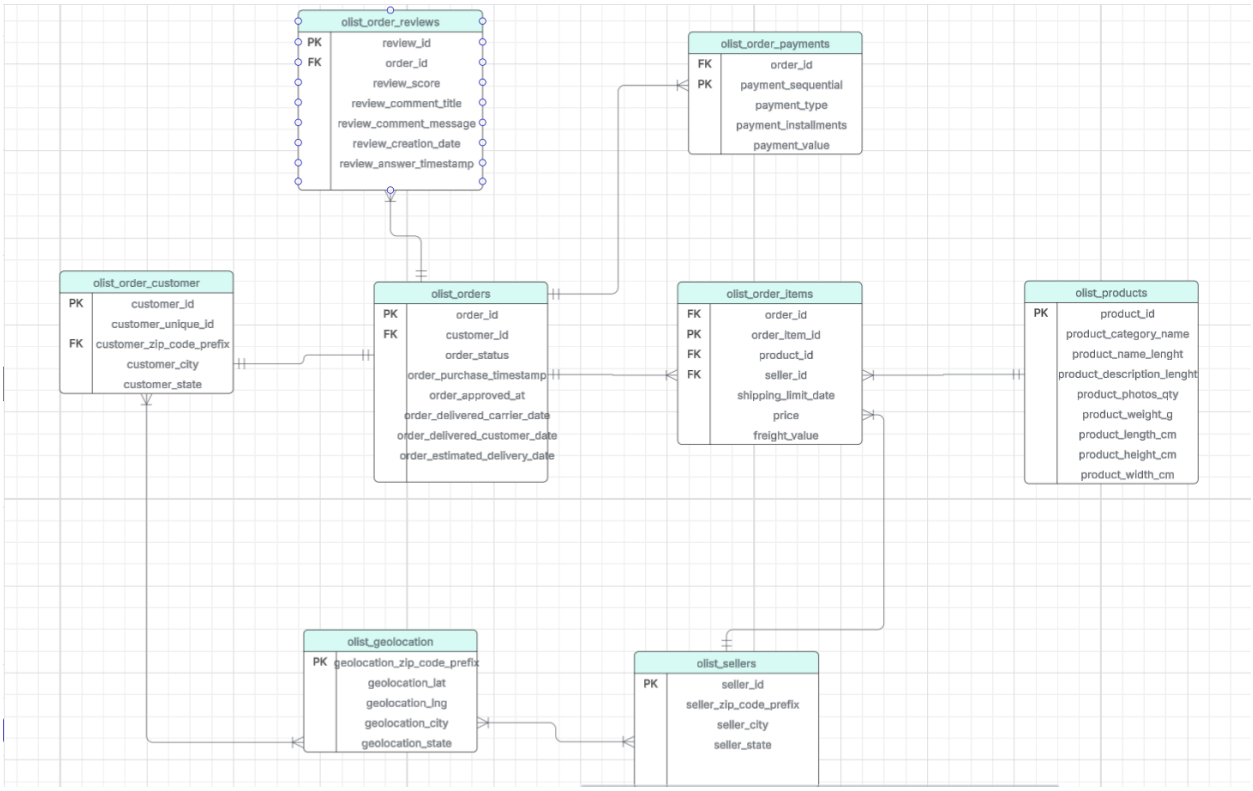
- `olist_orders`
- `olist_order_customer`
- `olist_geolocation`
- `olist_products`
- `olist_sellers`
- `olist_order_items`
- `olist_order_payments`
- `Olist_order_reviews`

Then, we used the diagram to illustrate the schema of the "OLIST" e-commerce database, outlining eight key tables and their connections. The tables represent core aspects of the e-commerce process: **orders**, **customers**, **geolocation**, **order items**, **products**, **sellers**, **payments**, and **reviews**.

We used the ER plot to demonstrate the relationship between each tables. The relationship is described below:

- olist_orders is the central fact table, linked one-to-many to olist_order_items, olist_order_payments, and olist_order_reviews by order_id.

- olist_orders.customer_id references olist_order_customer.customer_id, modeling each order's association to one customer.
- olist_order_customer.customer_zip_code_prefix and olist_sellers.seller_zip_code_prefix both reference olist_geolocation.geolocation_zip_code_prefix .
- olist_order_items carries a composite primary key (order_id, order_item_id) and foreign keys to olist_products (product_id) and olist_sellers (seller_id)
- olist_order_payments likewise uses (order_id, payment_sequential) as its composite PK, linking back to olist_orders to record multiple payment events per order.
- olist_order_reviews uses review_id as its primary key and a foreign key on order_id.



Each table represents a key component of the e-commerce transaction lifecycle. For example, `olist_orders` provides order-level information, while `olist_order_items` includes product-level details for each order. The `olist_geolocation` table adds location-based features. Finished those processes, we wrote the codes in the snowflake workshop, the codes showing below:

```
CREATE DATABASE Olist;
USE DATABASE Olist;
USE SCHEMA PUBLIC
```

```sql
-- 1. olist_orders table
CREATE OR REPLACE TABLE olist_orders (
  order_id                      VARCHAR   NOT NULL,
  customer_id                   VARCHAR   NOT NULL,
  order_status                  VARCHAR,
  order_purchase_timestamp      TIMESTAMP,
  order_approved_at             TIMESTAMP,
  order_delivered_carrier_date  TIMESTAMP,
  order_delivered_customer_date TIMESTAMP,
  order_estimated_delivery_date TIMESTAMP,
  PRIMARY KEY (order_id));
```

```sql
-- 2. olist_order_customer table
CREATE OR REPLACE TABLE olist_order_customer (
  customer_id           VARCHAR   NOT NULL,
  customer_unique_id    VARCHAR,
  customer_zip_code_prefix  NUMBER,
  customer_city         VARCHAR,
  customer_state        VARCHAR,
  PRIMARY KEY (customer_id));
-- order table references customer table
ALTER TABLE olist_orders
  ADD FOREIGN KEY (customer_id)
  REFERENCES olist_order_customer (customer_id);
```

```sql
-- 3. olist_geolocation table
CREATE OR REPLACE TABLE olist_geolocation (
  geolocation_zip_code_prefix  NUMBER   NOT NULL,
  geolocation_lat              FLOAT,
  geolocation_lng              FLOAT,
  geolocation_city             VARCHAR,
  geolocation_state            VARCHAR,
  PRIMARY KEY (geolocation_zip_code_prefix));
-- Customer table references geolocation
ALTER TABLE olist_order_customer
  ADD FOREIGN KEY (customer_zip_code_prefix)
  REFERENCES olist_geolocation (geolocation_zip_code_prefix);
```

```sql
-- 4. olist_products table
CREATE OR REPLACE TABLE olist_products (
  product_id                  VARCHAR   NOT NULL,
  product_category_name       VARCHAR,
  product_name_lenght         NUMBER,
  product_description_lenght  NUMBER,
  product_photos_qty          NUMBER,
  product_weight_g            NUMBER,
  product_length_cm           NUMBER,
  product_height_cm           NUMBER,
  product_width_cm            NUMBER,
  PRIMARY KEY (product_id));
```

```sql
-- 5. olist_sellers table
CREATE OR REPLACE TABLE olist_sellers (
  seller_id            VARCHAR   NOT NULL,
  seller_zip_code_prefix  NUMBER,
  seller_city          VARCHAR,
  seller_state         VARCHAR,
  PRIMARY KEY (seller_id));
-- sellers tables reference geolocation
ALTER TABLE olist_sellers
  ADD FOREIGN KEY (seller_zip_code_prefix)
  REFERENCES olist_geolocation (geolocation_zip_code_prefix);
```

```sql
-- 6. olist_order_items table
CREATE OR REPLACE TABLE olist_order_items (
  order_id            VARCHAR   NOT NULL,
  order_item_id       NUMBER    NOT NULL,
  product_id          VARCHAR   NOT NULL,
  seller_id           VARCHAR   NOT NULL,
  shipping_limit_date TIMESTAMP,
  price               FLOAT,
  freight_value       FLOAT,
  PRIMARY KEY (order_id, order_item_id),
  FOREIGN KEY (order_id)   REFERENCES olist_orders  (order_id),
  FOREIGN KEY (product_id) REFERENCES olist_products(product_id),
  FOREIGN KEY (seller_id)  REFERENCES olist_sellers (seller_id));
```

```sql
-- 8. olist_order_reviews table
CREATE OR REPLACE TABLE olist_order_reviews (
  review_id               VARCHAR   NOT NULL,
  order_id                VARCHAR   NOT NULL,
  review_score            NUMBER,
  review_comment_title    VARCHAR,
  review_comment_message  VARCHAR,
  review_creation_date    TIMESTAMP,
  review_answer_timestamp TIMESTAMP,
  PRIMARY KEY (review_id),
  FOREIGN KEY (order_id) REFERENCES olist_orders (order_id));

-- 7. olist_order_payments table
CREATE OR REPLACE TABLE olist_order_payments (
  order_id             VARCHAR   NOT NULL,
  payment_sequential   NUMBER    NOT NULL,
  payment_type         VARCHAR,
  payment_installments NUMBER,
  payment_value        FLOAT,
  PRIMARY KEY (order_id, payment_sequential),
  FOREIGN KEY (order_id) REFERENCES olist_orders (order_id));
```

After creating these tables, we then uploaded the original CSV datasets to those tables, and then checked the counts for each table, to make sure that the upload has been successful.

```sql
-- Check
SELECT COUNT(*) FROM olist_orders;
SELECT COUNT(*) FROM olist_order_items;
SELECT COUNT(*) FROM olist_order_payments;
SELECT COUNT(*) FROM olist_order_reviews;
SELECT COUNT(*) FROM olist_order_customer;
SELECT COUNT(*) FROM olist_products;
SELECT COUNT(*) FROM olist_sellers;
SELECT COUNT(*) FROM olist_geolocation;
```

Results | Chart

	COUNT(*)
1	1000163

4. Next, we connected Snowflake to Jupyter Notebook in local environment for the next steps. For the connection we are using the SQL Alchemy package. And then we are fetching the data from tables in snowflake database and are storing them as dataframes.

```
user = 'LiMi686'
password = getpass.getpass('Enter your Snowflake password: ')
account = 'TVRZIHA-IFB54421'
warehouse = 'COMPUTE_WH'
database = 'OLIST'
schema = 'PUBLIC'

# Create SQLAlchemy engine
engine = create_engine(
    f'snowflake://{user}:{password}@{account}/{database}/{schema}?warehouse={warehouse}'
)

# Fetch data from tables
def fetch_table(table_name):
    query = f'SELECT * FROM {table_name};'
    with engine.connect() as conn:
        return pd.read_sql(query, conn)

# Load all required tables
orders_df = fetch_table('olist_orders')
customers_df = fetch_table('olist_order_customer')
geolocation_df = fetch_table('olist_geolocation')
products_df = fetch_table('olist_products')
sellers_df = fetch_table('olist_sellers')
order_items_df = fetch_table('olist_order_items')
payments_df = fetch_table('olist_order_payments')
reviews_df = fetch_table('olist_order_reviews')
```

```
Enter your Snowflake password:  ........
```

1.2 Initial Data Overview

Before cleaning, the shape of each table was printed to understand the scale Here's a snapshot of the data shapes.

```
print("Before cleaning:")
print(f"Shape of orders table/df: {orders_df.shape}")
print(f"Shape of customers table/df: {customers_df.shape}")
print(f"Shape of geolocation table/df: {geolocation_df.shape}")
print(f"Shape of products table/df: {products_df.shape}")
print(f"Shape of sellers table/df: {sellers_df.shape}")
print(f"Shape of order_items table/df: {order_items_df.shape}")
print(f"Shape of payments table/df: {payments_df.shape}")
print(f"Shape of reviews table/df: {reviews_df.shape}")
```

```
Before cleaning:
Shape of orders table/df: (99441, 8)
Shape of customers table/df: (99441, 5)
Shape of geolocation table/df: (1000163, 5)
Shape of products table/df: (32951, 9)
Shape of sellers table/df: (3095, 4)
Shape of order_items table/df: (112650, 7)
Shape of payments table/df: (103886, 5)
Shape of reviews table/df: (99224, 7)
```

1.3 Data Cleaning

Each table underwent the following steps:

- Delete Duplicates: Ensures the uniqueness of entities

- Handle missing values: Dropped rows with null values to maintain integrity during joins. These steps significantly improve the reliability of the downstream merged dataset, reducing potential noise and inconsistency

```
def clean_df(df, name):
    df = df.drop_duplicates()
    df = df.dropna()
    print(f"After cleaning: {name} -> {df.shape}")
    return df

orders_df = clean_df(orders_df, "orders")
customers_df = clean_df(customers_df, "customers")
geolocation_df = clean_df(geolocation_df, "geolocation")
products_df = clean_df(products_df, "products")
sellers_df = clean_df(sellers_df, "sellers")
order_items_df = clean_df(order_items_df, "order_items")
payments_df = clean_df(payments_df, "payments")
reviews_df = clean_df(reviews_df, "reviews")
```

```
After cleaning: orders -> (96461, 8)
After cleaning: customers -> (99441, 5)
After cleaning: geolocation -> (738332, 5)
After cleaning: products -> (32340, 9)
After cleaning: sellers -> (3095, 4)
After cleaning: order_items -> (112650, 7)
After cleaning: payments -> (103886, 5)
After cleaning: reviews -> (9837, 7)
```

1.4 Merging Tables

A series of inner and left joins were used to combine the datasets into a single DataFrame named orders_full

1. Orders + Customers (on customer_id)
2. + geolocation (on customer_zip_code_prefix)
3. + Order Items (on order_id)
4. + products (on product id
5. +sellers (on seller_id)
6. + payments (on order_id)
7. +Reviews (on order_id)

```python
# Merge 1: Orders + Customers
orders_customers = pd.merge(orders_df, customers_df, on='customer_id', how='inner')

# Merge 2: + Geolocation (via zip prefix)
orders_customers_geo = pd.merge(
    orders_customers,
    geolocation_df,
    left_on='customer_zip_code_prefix',
    right_on='geolocation_zip_code_prefix',
    how='left'
)

# Merge 3: + Order Items
orders_full = pd.merge(orders_customers_geo, order_items_df, on='order_id', how='inner')

# Merge 4: + Products
orders_full = pd.merge(orders_full, products_df, on='product_id', how='left')

# Merge 5: + Sellers
orders_full = pd.merge(orders_full, sellers_df, on='seller_id', how='left')

# Merge 6: + Payments
orders_full = pd.merge(orders_full, payments_df, on='order_id', how='left')

# Merge 7: + Reviews
orders_full = pd.merge(orders_full, reviews_df, on='order_id', how='left')

# Result
print(f"Unified dataset shape: {orders_full.shape}")
orders_full.head()

# orders_full.to_csv('Full_data.csv', index=False)
```

```
Unified dataset shape: (12030992, 44)
```

2. Feature Selection and Construction

This project focuses on **unsupervised learning**, particularly clustering and RFM analysis, no response or target variable is required. Instead, meaningful features were created and selected to represent customer behavior and order characteristics. These features form the foundation for identifying patterns in customer activity and product ordering trends.

2.1 Temporal and Ratio-Based Features

Several new features were created from date and transaction-related fields.

- **Delivery Time**: Number of days between the order purchase and actual delivery to the customer.
- **Estimated Delay**: Difference in days between actual and estimated delivery dates. A positive value indicates a late delivery.
- **Approval Delay**: Number of hours between order placement and approval, indicating how quickly the system processed transactions.
- **Freight-to-Price Ratio**: The ratio of shipping cost to product cost, highlighting potentially expensive deliveries.

To ensure data quality, infinite or missing ratio values were removed.

```python
# Convert dates to datetime
date_cols = [
    'order_purchase_timestamp', 'order_approved_at',
    'order_delivered_carrier_date', 'order_delivered_customer_date',
    'order_estimated_delivery_date'
]

for col in date_cols:
    orders_full[col] = pd.to_datetime(orders_full[col])

# Feature 1: Delivery Time
orders_full['delivery_time'] = (orders_full['order_delivered_customer_date'] - orders_full['order_purchase_timestamp']).dt.days

# Feature 2: Estimated Delay (output is positive if late)
orders_full['estimated_delay'] = (orders_full['order_delivered_customer_date'] - orders_full['order_estimated_delivery_date']).dt.days

# Feature 3: Approval Delay
orders_full['approval_delay'] = (orders_full['order_approved_at'] - orders_full['order_purchase_timestamp']).dt.total_seconds() / 3600  # (time in hours)

# Feature 4: Freight-to-Price Ratio
orders_full['freight_ratio'] = orders_full['freight_value'] / orders_full['price']

# Clean infinite/NaN ratios
orders_full['freight_ratio'] = orders_full['freight_ratio'].replace([float('inf'), -float('inf')], pd.NA)
orders_full.dropna(subset=['freight_ratio'], inplace=True)

# Preview
orders_full[['delivery_time', 'estimated_delay', 'approval_delay', 'freight_ratio']].describe()
```

2.2 Aggregated Customer-Level Features

Customer-centric features were created by grouping the data by `customer_id`. These features help in analyzing purchasing patterns and satisfaction:

- **Number of Orders**
- **Average Order Value**
- **Most Frequently Used Payment Type**
- **Average Review Score**
- **Customer State** (geographic location)

```python
# Group by customer_id
customer_features = orders_full.groupby('customer_id').agg({
    'order_id': 'nunique',
    'price': 'mean',
    'payment_type': lambda x: x.mode().iloc[0] if not x.mode().empty else None,
    'review_score': 'mean',
    'customer_state': 'first'
}).reset_index()

customer_features.columns = [
    'customer_id',
    'num_orders',
    'avg_order_value',
    'most_used_payment_type',
    'avg_review_score',
    'customer_state'
]

customer_features.head()
```

	customer_id	num_orders	avg_order_value	most_used_payment_type	avg_review_score	customer_state
0	00012a2ce6f8dcda20d059ce98491703	1	89.80	credit_card	NaN	SP
1	000161a058600d5901f007fab4c27140	1	54.90	credit_card	NaN	MG
2	0001fd6190edaaf884bcaf3d49edf079	1	179.99	credit_card	NaN	ES
3	0002414f95344307404f0ace7a26f1d5	1	149.90	boleto	NaN	MG
4	000379cdec625522490c315e70c7a9fb	1	93.00	boleto	NaN	SP

2.3 Aggregated Order-Level Features

Order-specific metrics were created to capture product diversity and operational metrics:

- **Unique Products per Order**
- **Total Order Value**
- **Total Freight Cost**
- **Number of Items**
- **Average Delivery Time**
- **Average Delay**

```
order_level_features = orders_full.groupby('order_id').agg({
    'customer_id': 'first',
    'product_id': 'nunique',
    'price': 'sum',
    'freight_value': 'sum',
    'order_item_id': 'count',
    'delivery_time': 'mean',
    'estimated_delay': 'mean'
}).reset_index()

order_level_features.columns = [
    'order_id', 'customer_id', 'unique_products', 'total_order_value',
    'total_freight', 'num_items', 'avg_delivery_time', 'avg_estimated_delay'
]
```

```
order_level_features.head()
```

order_id	customer_id	unique_products	total_order_value	total_freight	num_items	avg_delivery_time	avg_estimated_delay
0242fe8c5a6d1ba2dd792cb16214	3ce436f183e68e07877b285a838db11a	1	6714.60	1515.06	114	7.0	-9.0
18f77f2f0320c557190d7a144bdd3	f6dd3ec061db4e3987629fe6b26e5cce	1	74608.90	6198.23	311	16.0	-3.0
229ec398224ef6ca0657da4fc703e	6489ae5e4333f3693df5ad4372dab6d3	1	39203.00	3520.39	197	7.0	-14.0
!4acbcdf0a6daa1e931b038114c75	d4eb9395c8c0431ee92fce09860c5a06	1	220.83	217.43	17	6.0	-6.0
42b26cf59d7ce69dfabb4e55b4fd9	58dbd0b2d70206bf40e62cd34e84d795	1	799.60	72.56	4	25.0	-16.0

2.4 Dataset Creation for Clustering

The final dataset used for clustering was created by merging **customer-level** and **order-level** features. Identifiers such as `order_id` and `customer_id` were dropped to prevent them from influencing the algorithm.

Two groups of features were handled distinctly:

- **Numerical Features**: These are scaled using `StandardScaler`.
- **Categorical Features**: These are Encoded using `OneHotEncoder`.

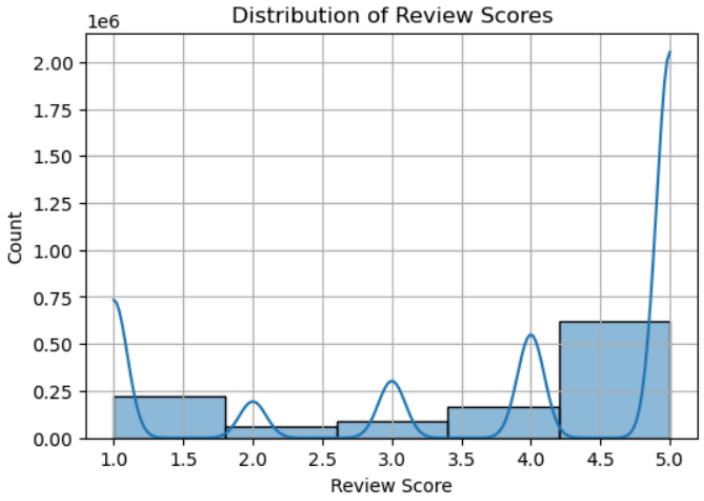
This transformed feature matrix, `clustering_prepared`, serves as input to K-Means clustering.

```
# Combine customer and order-level features
clustering_data = pd.merge(order_level_features, customer_features, on='customer_id', how='left')

# Drop IDs
clustering_data_clean = clustering_data.drop(columns=['order_id', 'customer_id'])

# Define columns
num_cols = ['num_orders', 'avg_order_value', 'avg_review_score', 'unique_products', 'total_order_value',
            'total_freight', 'num_items', 'avg_delivery_time', 'avg_estimated_delay']
cat_cols = ['most_used_payment_type', 'customer_state']

# Build pipeline
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), num_cols),
    ('cat', OneHotEncoder(sparse_output=False, handle_unknown='ignore'), cat_cols)
])

clustering_prepared = preprocessor.fit_transform(clustering_data_clean)

print(f"Final shape for clustering: {clustering_prepared.shape}")
```

```
Final shape for clustering: (96461, 41)
```

2.5 Visualizations

```
plt.figure(figsize=(6, 4))
sns.histplot(orders_full['review_score'].dropna(), bins=5, kde=True)
plt.title('Distribution of Review Scores')
plt.xlabel('Review Score')
plt.ylabel('Count')
plt.grid(True)
plt.show()
```

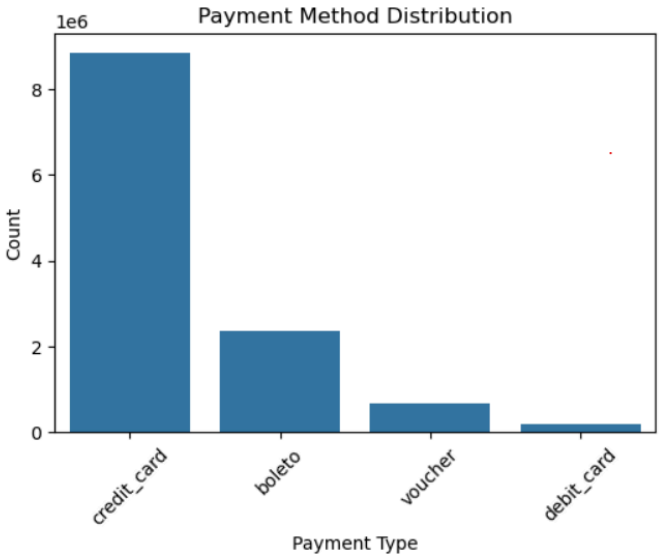


This histogram shows the distribution of review scores in the Brazilian e-commerce dataset. The majority of reviews are concentrated at the highest score of 5, indicating a high level of customer satisfaction. There are smaller peaks at scores of 1, 4, and to a lesser extent, 2 and 3, suggesting some customers had less positive experiences. The distribution is heavily skewed towards positive reviews.

```
avg_order_value = orders_full.groupby('customer_id')['price'].mean()

plt.figure(figsize=(6, 4))
sns.histplot(avg_order_value, bins=30, kde=True)
plt.title('Average Order Value per Customer')
plt.xlabel('Avg. Order Value')
plt.ylabel('Count')
plt.grid(True)
plt.show()
```

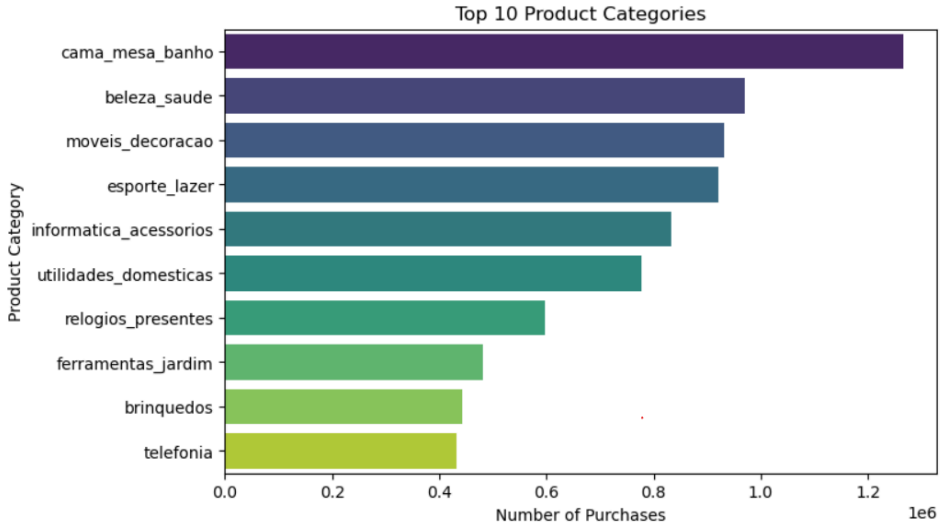
This histogram displays the distribution of the average order value per customer. The vast majority of customers have a relatively low average order value, concentrated towards the lower end of the price range. The distribution is strongly right-skewed, indicating that while most customers spend a small amount on average, there are a few customers with significantly higher average order values.

```
plt.figure(figsize=(6, 4))
sns.countplot(data=orders_full, x='payment_type', order=orders_full['payment_type'].value_counts().index)
plt.title('Payment Method Distribution')
plt.xlabel('Payment Type')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.show()
```



This bar plot illustrates the distribution of payment methods used in the Brazilian e-commerce dataset. Credit card is the overwhelmingly most popular payment method, followed by boleto (a Brazilian payment slip). Voucher and debit card are used less frequently. This indicates a strong preference for credit cards among customers in this dataset.

```
top_cats = orders_full['product_category_name'].value_counts().head(10)

plt.figure(figsize=(8, 5))
sns.barplot(x=top_cats.values, y=top_cats.index, palette='viridis', hue=top_cats.index, legend=False)
plt.title('Top 10 Product Categories')
plt.xlabel('Number of Purchases')
plt.ylabel('Product Category')
plt.show()
```



This horizontal bar plot displays the top 10 product categories by the number of purchases in the Brazilian e-commerce dataset. "Cama_mesa_banho" (bed, table, and bath) is the most purchased category by a significant margin. Following behind are "beleza_saude" (beauty and health) and "moveis_decoracao" (furniture and decoration). The remaining categories show a gradual decrease in the number of purchases, indicating varying levels of demand across these top segments.

```
# Bar plot of num_orders per customer_state
orders_per_state = customer_features.groupby('customer_state')['num_orders'].sum().reset_index()
sns.barplot(x='customer_state', y='num_orders', data=orders_per_state)
plt.title('Number of Orders by State')
plt.xlabel('State')
plt.ylabel('Number of Orders')
plt.xticks(rotation=45)
plt.show()
```

The bar plot shows the total number of e-commerce orders per Brazilian state, revealing significant variation across regions. São Paulo (SP) stands out with a dramatically higher number of orders, indicating a strong concentration of e-commerce activity there. Many other states exhibit considerably lower order volumes, forming a long tail in the distribution, suggesting less prevalent e-commerce activity in those areas.

The top five states by order volume are São Paulo (SP), Rio de Janeiro (RJ), Minas Gerais (MG), Paraná (PR), and Rio Grande do Sul (RS). This distribution likely reflects factors like population density, economic activity, and infrastructure within each state. Overall, the data highlights a clear dominance of a few key states in the Brazilian e-commerce landscape represented in this dataset.

3. ML Techniques Implementation

3.1 RFM (Recency, Frequency, Monetary) Analysis

RFM analysis is a marketing technique used to understand customer behavior by examining three key factors: **Recency**, **Frequency**, and **Monetary value**. It helps identify valuable customer segments and tailor marketing strategies accordingly.

1. **Define the Reference Date:** Created a "snapshot date" variable by taking the latest purchase date in r dataset and adding one day. This serves as a fixed point in time to calculate recency.
2. **Aggregate RFM Values:** Grouped the customer data by customer_id and calculated the following for each customer:
    - **Recency:** The number of days since their last purchase (calculated by subtracting the customer's latest purchase date from the snapshot_date).
    - **Frequency:** The total number of orders they placed.
    - **Monetary:** The total amount of money they spent.
3. **Calculate Descriptive Statistics:** Used the .describe() method to get a statistical overview of the Recency, Frequency, and Monetary values. This helps in understanding the distribution and range of these metrics across r customer base.

```
# Reference date: last purchase date + 1
snapshot_date = orders_full['order_purchase_timestamp'].max() + pd.Timedelta(days=1)

# Aggregate RFM values
rfm = orders_full.groupby('customer_id').agg({
    'order_purchase_timestamp': lambda x: (snapshot_date - x.max()).days,  # Recency
    'order_id': 'count',                                                   # Frequency
    'price': 'sum'                                                         # Monetary
}).reset_index()

rfm.columns = ['customer_id', 'recency', 'frequency', 'monetary']

rfm.describe()
```

	recency	frequency	monetary
count	96461.000000	96461.000000	9.646100e+04
mean	240.104757	124.723899	1.500053e+04
std	152.827594	168.591327	3.819059e+04
min	1.000000	1.000000	5.600000e+00
25%	116.000000	41.000000	2.419000e+03
50%	221.000000	79.000000	6.089130e+03
75%	350.000000	152.000000	1.478400e+04
max	714.000000	9451.000000	2.895372e+06

Descriptive Statistics of RFM Values

- This table provides a statistical summary of the raw Recency, Frequency, and Monetary values across all customers.
- **count:** Shows the total number of customers (96461).
- **mean:** Represents the average value for each metric. On average, a customer's last purchase was about 240 days ago, they placed around 1.25 orders, and spent approximately $15,000.
- **std (standard deviation):** Indicates the spread or variability of the data. Higher standard deviation means more variation in customer behavior.
- **min:** Shows the minimum value for each metric. The most recent purchase was 1 day ago, the minimum number of orders is 1, and the minimum spending is $5.60.
- **25%, 50% (median), 75%:** These are percentiles that divide the data into four equal parts. For example, 25% of customers made a purchase within 116 days, 50% within 221 days, and 75% within 350 days. Similarly for frequency and monetary value.
- **max:** Shows the maximum value for each metric. The longest time since a purchase is 714 days, the maximum number of orders by a single customer is 9451, and the highest spending by a customer is approximately $2.89 million.

4. **Score Each Metric (1-4):** Assigned scores from 1 to 4 to each customer based on their Recency, Frequency, and Monetary values using quantiles:

    - **Recency Score (r_score):** Customers with more recent purchases received a higher score (4 being the most recent, 1 being the least recent). Used “pd.qcut” to divide customers into four equal groups based on their recency.
    - **Frequency Score (f_score):** Customers who placed more orders received a higher score (4 being the highest frequency, 1 being the lowest). Used “pd.qcut” on the rank of the frequency to handle potential ties.
    - **Monetary Score (m_score):** Customers who spent more money received a higher score (4 being the highest spending, 1 being the lowest). Used “pd.qcut” to divide customers into four equal groups based on their monetary value.

5. **Create RFM Segment:** Combined the individual R, F, and M scores into a three-digit string to create RFM segments (e.g., a customer with an r_score of 4, f_score of 2, and m_score of 3 belongs to the "423" segment).

6. **Count Customers in Each Segment:** Counted the number of customers falling into each unique RFM segment.

```
# Score each metric 1-4
rfm['r_score'] = pd.qcut(rfm['recency'], 4, labels=[4, 3, 2, 1]).astype(int)
rfm['f_score'] = pd.qcut(rfm['frequency'].rank(method='first'), 4, labels=[1, 2, 3, 4]).astype(int)
rfm['m_score'] = pd.qcut(rfm['monetary'], 4, labels=[1, 2, 3, 4]).astype(int)

# Create RFM segment
rfm['rfm_segment'] = rfm['r_score'].astype(str) + rfm['f_score'].astype(str) + rfm['m_score'].astype(str)

# Preview
# Count of customers in each RFM segment
rfm['rfm_segment'].value_counts()

rfm.head()
```

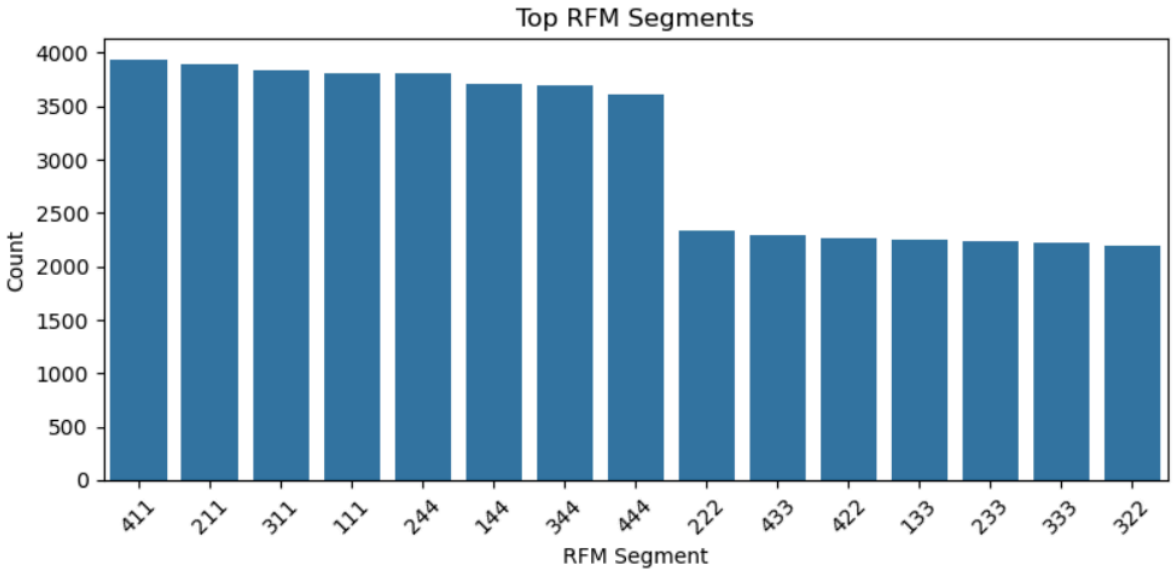
	customer_id	recency	frequency	monetary	r_score	f_score	m_score	rfm_segment
0	00012a2ce6f8dcda20d059ce98491703	288	75	6735.00	2	2	3	223
1	000161a058600d5901f007fab4c27140	410	87	4776.30	1	3	2	132
2	0001fd6190edaaf884bcaf3d49edf079	548	173	31138.27	1	4	4	144
3	0002414f95344307404f0ace7a26f1d5	379	1	149.90	1	1	1	111
4	000379cdec625522490c315e70c7a9fb	150	72	6696.00	3	2	3	323

Sample of RFM Segmented Data

- This table shows the first few rows of r RFM analysis results. Each row represents a unique customer.
- **customer_id:** A unique identifier for each customer.
- **recency, frequency, monetary:** The raw calculated values for each customer.
- **r_score, f_score, m_score:** The assigned scores (1-4), based on the quantiles of the respective raw values.
- **rfm_segment:** The combined RFM segment for each customer. For example, the first customer has a segment "223", meaning their recency is in the second quartile (not the most recent), their frequency is in the second quartile, and their monetary value is in the third quartile.

7. **Visualize Top RFM Segments:** Created a bar chart showing the count of customers in the top 15 most frequent RFM segments.

```
plt.figure(figsize=(8, 4))
sns.countplot(data=rfm, x='rfm_segment', order=rfm['rfm_segment'].value_counts().index[:15])
plt.title('Top RFM Segments')
plt.xlabel('RFM Segment')
plt.ylabel('Count')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Bar Chart of Top RFM Segments

- This chart visually represents the distribution of customers across different RFM segments, focusing on the segments with the highest number of customers.
- The x-axis shows the RFM segments (e.g., "411", "211", "311"). The first digit represents the Recency score, the second represents the Frequency score, and the third represents the Monetary score.
- The y-axis shows the number of customers in each segment (Count).

- The tallest bars indicate the RFM segments with the largest customer populations.
- The segment "411" has the highest number of customers among the top 15. This signifies a large group of customers who purchased most recently (Recency score of 4) but have the lowest frequency (Frequency score of 1) and lowest monetary value (Monetary score of 1) compared to other segments.
- By examining the top segments, we can identify the most common customer behavior patterns.

By understanding these segments, we can target specific groups of customers with tailored marketing campaigns. For instance, customers in the "444" segment (not shown in the top 5 but would be the ideal segment) are the most valuable as they purchased recently, frequently, and spent the most. Conversely, segments with low recency scores (e.g., starting with "1") represent customers at risk of churning.

3.2 K-Means Clustering

K-Means clustering algorithm aims to group data points into distinct clusters based on their similarity. The algorithm iteratively assigns each data point to the cluster whose centroid (mean of all points in that cluster) is closest. The goal is to minimize the within-cluster variance, meaning the data points within each cluster should be as similar to each other as possible.

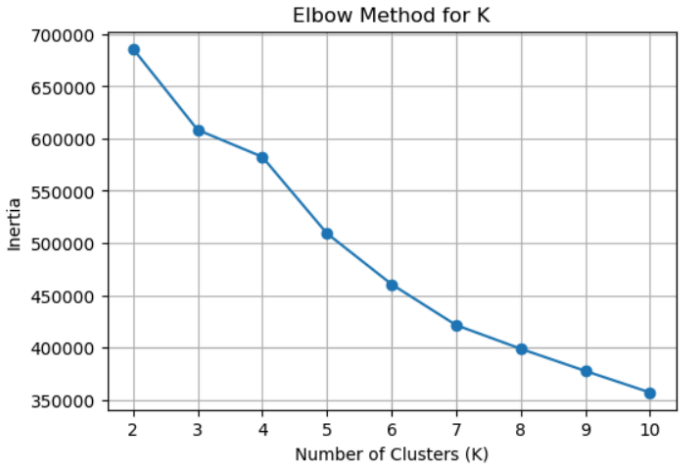
```
from sklearn.cluster import KMeans
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy='mean')
clustering_prepared = imputer.fit_transform(clustering_prepared)
#Try K from 2 to 10
inertia = []
K_range = range(2, 11)

for k in K_range:
    km = KMeans(n_clusters=k, random_state=42, n_init='auto')
    km.fit(clustering_prepared)
    inertia.append(km.inertia_)

# Plot
plt.figure(figsize=(6, 4))
plt.plot(K_range, inertia, marker='o')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Inertia')
plt.title('Elbow Method for K')
plt.grid(True)
plt.show()
```

1. **Import Libraries:** Imported necessary libraries: KMeans for the clustering algorithm and SimpleImputer to handle any missing values in r data.

2. **Handle Missing Values**: Used SimpleImputer with the mean strategy to fill any missing values in r clustering_prepared data with the mean of the respective column. This ensures that the clustering algorithm can process all data points.

3. Determine the Optimal Number of Clusters (K) using the Elbow Method:

    - Initialized an empty list called inertia. Inertia measures the within-cluster sum of squares, representing how far the data points are from their cluster's centroid. Lower inertia indicates denser, better-separated clusters.
    - Iterated through a range of possible cluster numbers (K) from 2 to 10.
    - For each K:
        - Created a KMeans model with the specified number of clusters (n_clusters=k), a fixed random state for reproducibility (random_state=42), and set n_init='auto' to let scikit-learn handle the number of initializations.
        - Fit the K-Means model to r prepared data (clustering_prepared).
        - Appended the calculated inertia of the fitted model to the inertia list.
    - Then plotted the inertia values against the corresponding number of clusters (K). The "elbow" point in this plot, where the rate of decrease in inertia starts to slow down, is often considered a good estimate for the optimal number of clusters.



- This plot shows the inertia (within-cluster sum of squares) on the y-axis against the number of clusters (K) on the x-axis.
- The goal is to find an "elbow" point where the decrease in inertia starts to level off, indicating that adding more clusters beyond this point may not significantly reduce the within-cluster variance.
- In this plot, the decrease in inertia is quite steep from K=2 to K=4. After K=4, the rate of decrease became less pronounced, suggesting that K=4 might be a

reasonable choice for the optimal number of clusters. This is the basis for choosing n_clusters=4 in the subsequent K-Means step.

```
kmeans = KMeans(n_clusters=4, random_state=42, n_init='auto')
clusters = kmeans.fit_predict(clustering_prepared)

# Add cluster labels to original DataFrame
clustering_data['cluster'] = clusters

# Check cluster distribution
clustering_data['cluster'].value_counts().sort_index()
```

```
cluster
0    19239
1    62406
2    10243
3     4573
Name: count, dtype: int64
```

4. **Perform K-Means Clustering with the Chosen Number of Clusters:**
    - Based on the elbow method (which suggests 4 clusters from Image 1), created a KMeans model with n_clusters=4, random_state=42, and n_init='auto'.
    - Fit this model to r prepared data and used .fit_predict() to assign each data point to one of the 4 clusters. The resulting clusters array contains the cluster label (0, 1, 2, or 3) for each data point.
5. **Add Cluster Labels to the Original DataFrame:** Added a new column named 'cluster' to r clustering_data DataFrame, containing the cluster assignment for each data point.

6. **Check Cluster Distribution:** Used .value_counts().sort_index() to see the number of data points belonging to each cluster, providing insight into the size and balance of the clusters.
7. **Calculate Average Values per Cluster (Cluster Profiles):** Grouped r clustering_data by the 'cluster' label and calculated the mean of the other numerical features for each cluster. This cluster_profiles DataFrame shows the average characteristics of the data points within each cluster.

8. **Normalize Cluster Profiles:** To better visualize and compare the characteristics of different clusters in the radar chart, normalized the cluster_profiles. Normalization scales the values of each feature to a range between 0 and 1. This is done by subtracting the minimum value of each feature and dividing by the range (maximum - minimum).

```
# Average values per cluster
cluster_profiles = clustering_data.groupby('cluster').mean(numeric_only=True)
cluster_profiles
```

cluster	unique_products	total_order_value	total_freight	num_items	avg_delivery_time	avg_estimated_delay	num_orders	avg_order_value	avg_review_score
0	1.026769	8777.880844	1831.133512	106.197515	10.681740	-12.811373	1.0	90.586795	4.060247
1	1.033410	10447.735874	1849.199086	104.131077	9.363747	-14.157998	1.0	112.426247	4.066530
2	1.018159	11658.380365	2195.032317	100.025090	31.387484	4.695011	1.0	132.078069	2.601050
3	1.204024	110796.123011	13994.258979	539.010934	12.070195	-13.907501	1.0	430.379948	3.517647

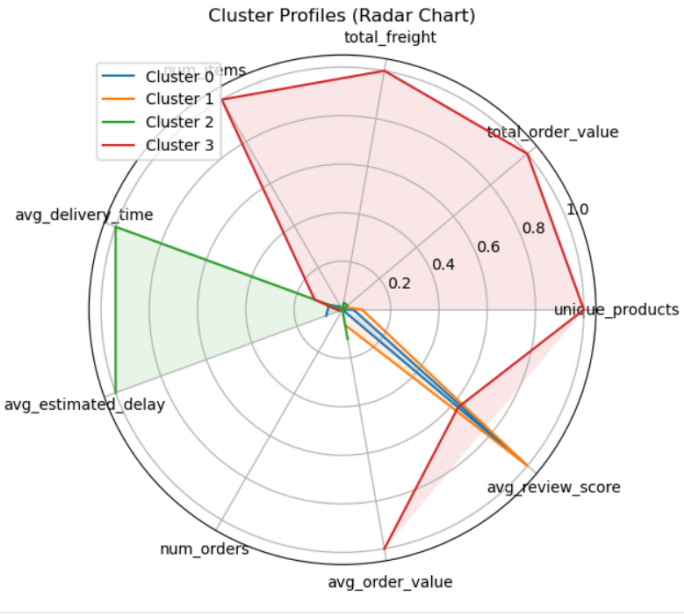
Average Values per Cluster

- This table shows the average values of different features for each of the four identified clusters (0, 1, 2, and 3). Each row represents a cluster, and each column represents a feature.
- By comparing the average values across clusters, we can understand the distinct characteristics of each group

```
# Normalize cluster profiles
profile_norm = cluster_profiles.copy()
profile_norm = (profile_norm - profile_norm.min()) / (profile_norm.max() - profile_norm.min())

# Prepare for radar
labels = profile_norm.columns.tolist()
num_vars = len(labels)

fig, ax = plt.subplots(figsize=(6, 6), subplot_kw=dict(polar=True))

for i in profile_norm.index:
    values = profile_norm.loc[i].tolist()
    values += values[:1]  # loop back to start
    angles = [n / float(num_vars) * 2 * pi for n in range(num_vars)]
    angles += angles[:1]

    ax.plot(angles, values, label=f'Cluster {i}')
    ax.fill(angles, values, alpha=0.1)

ax.set_xticks(angles[:-1])
ax.set_xticklabels(labels)
plt.title('Cluster Profiles (Radar Chart)')
plt.legend()
plt.show()
```

9. **Create a Radar Chart of Cluster Profiles:**
    - Extracted the feature labels (column names) from the normalized cluster profiles.
    - Determined the number of variables (features).

- Created a polar subplot for the radar chart.
- For each cluster:
  - Extracted the normalized values for all features.
  - Appended the first value to the end of the list to close the radar shape.
  - Calculated the angles for each feature on the radar plot.
  - Plotted the values at the corresponding angles, creating a polygon for each cluster, and filled the area with a transparent color.
  - Added a label for each cluster in the legend.
- Set the tick positions and labels for the features around the radar plot.
- Added a title and displayed the legend.



Cluster Profiles (Radar Chart)

- This radar chart provides a visual representation of the normalized average feature values for each cluster. Each axis represents a different feature, and the distance from the center indicates the normalized value (from 0 to 1).
- The different colored polygons represent the profiles of the four clusters:
  - **Cluster 0 (blue):** Shows relatively low values across most spending-related features (total_order_value, total_freight, num_items, avg_order_value) but a moderate avg_review_score and shorter avg_delivery_time.
  - **Cluster 1 (orange)**: Is characterized by moderately high spending features and the highest avg_review_score, along with shorter avg_delivery_time.
  - **Cluster 2 (green):** Stands out with high values for spending-related features but also high avg_delivery_time and avg_estimated_delay, and the lowest avg_review_score.
  - **Cluster 3 (red)**: Is notable for its very high avg_order_value and high total_freight, with moderate values for other spending features and a negative avg_estimated_delay.

Each cluster represents:

- **Cluster 0: "Budget-Conscious & Prompt Delivery" Customers:** These customers tend to place smaller orders with lower overall value and freight costs. They experience relatively quick deliveries, often arriving earlier than estimated, and provide moderate reviews.
- **Cluster 1: "Satisfied Medium Spenders"**: These customers spend a bit more than Cluster 0 and have the highest satisfaction levels based on their review scores. They also benefit from relatively prompt deliveries.
- **Cluster 2: "High-Value, Delayed Delivery" Customers:** This group represents customers who make the largest and most expensive orders with the highest number of items and freight costs. However, they experience the longest delivery times, often with delays, and are the least satisfied based on their reviews.
- **Cluster 3: "High Average Order Value" Customers:** These customers might place fewer items per order but have a very high average spending per order, leading to significant total freight costs. Their deliveries tend to be earlier than estimated, and their satisfaction is moderate.

Understanding these distinct customer profiles can help us tailor marketing strategies, improve delivery processes, and address the concerns of specific customer segments. One insight is that Cluster 2 requires attention to improve delivery times and customer satisfaction.

Summary:

In this project, we are analyzing the e-commerce data to understand customer behavior and order characteristics. The initial phase involved data preparation, where data from Kaggle is stored in multiple tables in a database in Snowflake and this data was extracted using Python programming, cleaned, and merged into a unified DataFrame . Feature engineering played a crucial role in creating meaningful metrics, including temporal aspects like delivery time and estimated delay, ratio-based features such as freight-to-price ratio, and aggregated customer and order-level features like average order value, number of unique products, and average review score. This prepared dataset formed the basis for the next steps. We used unsupervised machine learning techniques.

We employed RFM (Recency, Frequency, Monetary) analysis to segment customers based on their purchase history, revealing valuable insights into customer loyalty and spending habits. Top RFM segments, such as those with recent purchases but low frequency and monetary value, were identified, allowing for targeted marketing strategies. Then we applied K-Means clustering to group customers with similar purchasing patterns and order attributes. The elbow method suggested an optimal number of four clusters, and the resulting cluster profiles highlighted distinct customer segments, including "Budget-Conscious & Prompt Delivery" customers, "Satisfied Medium Spenders," "High-Value, Delayed Delivery" customers, and those with a "High Average Order Value." These profiles offer actionable insights for tailoring customer engagement strategies and improving operational efficiency, particularly in addressing the pain points of the "High-Value, Delayed Delivery" segment.