Name: Nikhil Rawat

In Summary, I aim to present a strategic approach that combines customer segmentation with a sophisticated recommendation system to drive sales and elevate customer satisfaction. Utilizing the KMeans clustering algorithm, we discerned eight optimal clusters, delineated by the elbow method, which categorize customers based on similar purchasing behaviors and propensities.

For each users on these clusters, we not only identify a top product based on purchase frequency but also curate specific incentives designed to resonate with the unique preferences of each segment. These incentives are meticulously tailored to augment the desirability of products for each customer group, thereby bolstering the potential for sales.

The integration of customer segmentation with our recommendation system extends into the realm of advanced AI with the use of a Neural Collaborative Filtering Model (NCF). The NCF enhances Customer-Specific Recommendations by learning from the complex patterns of user-item interactions. It goes beyond traditional collaborative filtering by incorporating a multi-layer perceptron that captures the non-linear relationships within the data, thus offering a deeper personalization of product suggestions.

New customers are welcomed with personalized attention, receiving recommendations for the top five products in their cluster alongside tailored incentives that align with the cluster's profile. This approach not only positions us to recommend products that echo the collective preferences of customer segments but also enables the provision of highly relevant incentives.

By integrating these advanced techniques, my approach is not just about responding to customer needs—it's about anticipating them and delivering a personalized shopping journey. This enhances the customer experience, drives conversions, and builds a foundation for lasting loyalty, setting a new standard for sales and marketing strategies in today's competitive marketplace.

Importing The Libraries

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import datetime as dt
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from kneed import KneeLocator
import squarify
import warnings
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder

warnings.filterwarnings('ignore')
```

Loading Dataset

In [ ]:
```
# Specify the file path for the Excel file containing retail data
file_path = '/Users/nick/Downloads/online_retail_II.xlsx'

# Load data from the first sheet (Year 2009-2010) of the Excel file
sheet1 = pd.read_excel(file_path, sheet_name='Year 2009-2010')

# Load data from the second sheet (Year 2010-2011) of the Excel file
sheet2 = pd.read_excel(file_path, sheet_name='Year 2010-2011')

# Concatenate the data from both sheets into a single DataFrame, resetting the index
data = pd.concat([sheet1, sheet2], ignore_index=True)
```

Data Overview

In [ ]:
```
# Display the first five rows of the DataFrame to preview the data
print(data.head())

# Print the number of rows and columns in the DataFrame
print(data.shape)

# Print a concise summary of the DataFrame, including column data types and non-null values
print(data.info())

# Calculate and print the number of missing values in each column of the DataFrame
print(data.isnull().sum())
```

```
  Invoice StockCode                          Description  Quantity  \
0  489434     85048  15CM CHRISTMAS GLASS BALL 20 LIGHTS        12
1  489434    79323P                   PINK CHERRY LIGHTS        12
2  489434    79323W                  WHITE CHERRY LIGHTS        12
3  489434     22041         RECORD FRAME 7" SINGLE SIZE         48
4  489434     21232       STRAWBERRY CERAMIC TRINKET BOX        24

          InvoiceDate  Price  Customer ID         Country
0 2009-12-01 07:45:00   6.95      13085.0  United Kingdom
1 2009-12-01 07:45:00   6.75      13085.0  United Kingdom
2 2009-12-01 07:45:00   6.75      13085.0  United Kingdom
3 2009-12-01 07:45:00   2.10      13085.0  United Kingdom
4 2009-12-01 07:45:00   1.25      13085.0  United Kingdom
(1067371, 8)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1067371 entries, 0 to 1067370
Data columns (total 8 columns):
 #   Column       Non-Null Count    Dtype
---  ------       --------------    -----
 0   Invoice      1067371 non-null  object
 1   StockCode    1067371 non-null  object
 2   Description  1062989 non-null  object
 3   Quantity     1067371 non-null  int64
 4   InvoiceDate  1067371 non-null  datetime64[ns]
 5   Price        1067371 non-null  float64
 6   Customer ID  824364 non-null   float64
 7   Country      1067371 non-null  object
dtypes: datetime64[ns](1), float64(2), int64(1), object(4)
memory usage: 65.1+ MB
None
Invoice             0
StockCode           0
Description      4382
Quantity            0
InvoiceDate         0
Price               0
Customer ID    243007
Country             0
dtype: int64
```

Data cleaning

In [ ]:
```python
# Remove rows where 'Customer ID' is null to ensure data quality and consistency
data = data[data['Customer ID'].notnull()]

# Filter out rows where 'Quantity' is less than or equal to 0 to eliminate incorrect or return entries
data = data[(data['Quantity'] > 0)]

# Filter out rows where 'Price' is less than or equal to 0 to remove erroneous entries
data = data[(data['Price'] > 0)]

# Convert 'Invoice' column data to string type to standardize and prevent data type inconsistencies
data['Invoice'] = data['Invoice'].astype(str)

# Exclude rows where 'Invoice' contains 'C', indicating cancelled transactions
data = data[~data['Invoice'].str.contains('C')]
```

Feature Engineering

In [ ]:
```python
# Extract and store only the date part (year, month, day) from 'InvoiceDate' to a new 'InvoiceDay' column
data['InvoiceDay'] = data['InvoiceDate'].apply(lambda x: dt.datetime(x.year, x.month, x.day))

# Add one day to the most recent date in 'InvoiceDay' to set a current date reference
current_date = max(data['InvoiceDay']) + dt.timedelta(1)

# Calculate the total amount for each transaction by multiplying 'Quantity' by 'Price'
data['TotalAmount'] = data['Quantity'] * data['Price']
```

Customer Segmentation

RFM (Recency, Frequency, Monetary) Calculation

In [ ]:
```python
# Group data by 'Customer ID' and aggregate it into RFM metrics:
rfm = data.groupby('Customer ID').agg({
    'InvoiceDay': lambda x: (current_date - x.max()).days,  # Calculate Recency: Days since last purchase
    'Invoice': 'count',  # Calculate Frequency: Total number of transactions
    'TotalAmount': 'sum'  # Calculate Monetary: Total spending
})

# Rename the columns for clarity following RFM segmentation terminology:
rfm.rename(columns={
    'InvoiceDay': 'Recency',
```

```
    'Invoice': 'Frequency',
    'TotalAmount': 'Monetary'
}, inplace=True)
```

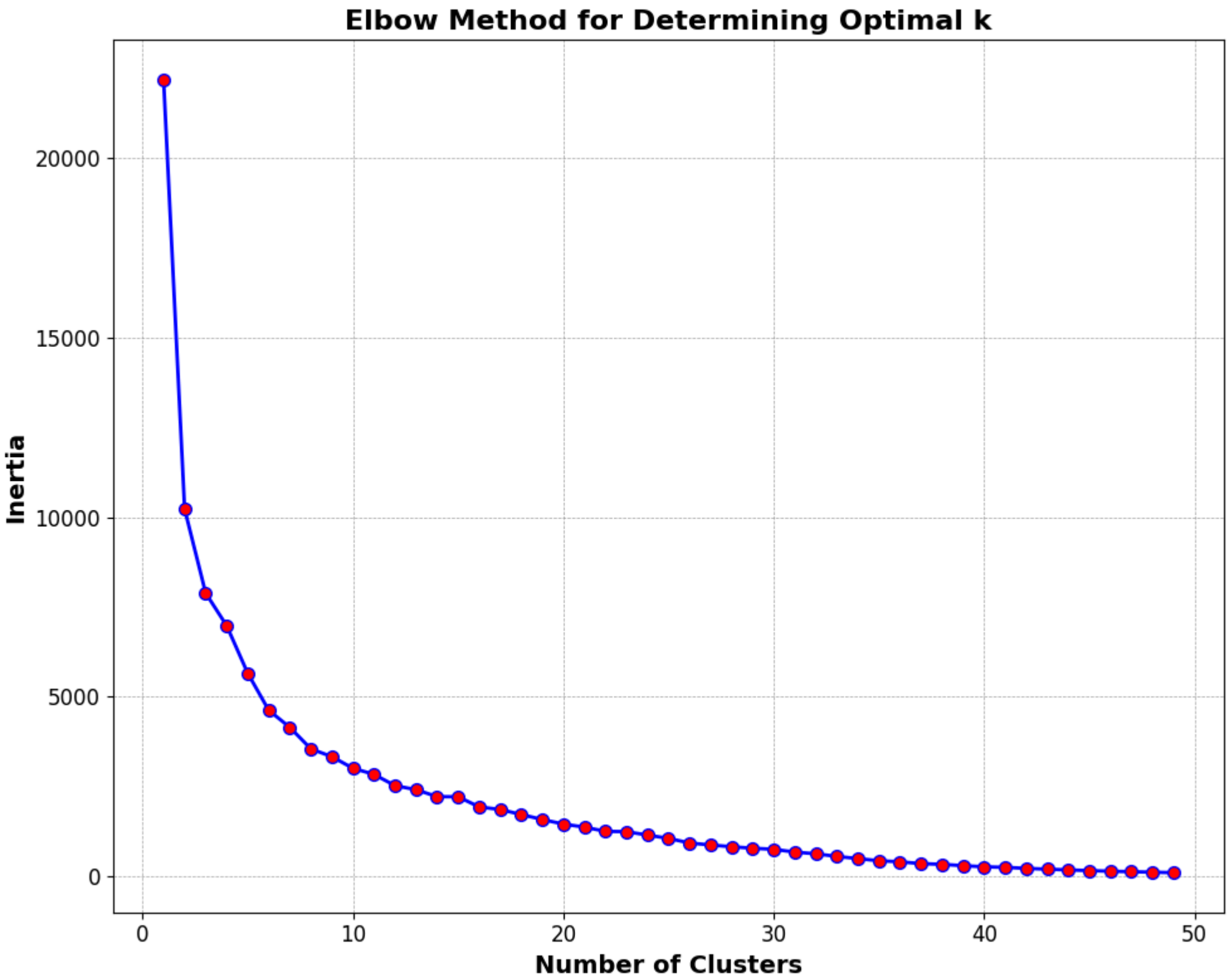
RFM Scoring Using Quartiles

In [ ]:
```
# Define descending labels for Recency, where 4 = most recent and 1 = least recent
r_labels = range(4, 0, -1)
# Categorize 'Recency' into quartiles using defined labels (most recent customers get the highest label)
r_groups = pd.qcut(rfm['Recency'], q=4, labels=r_labels)

# Define ascending labels for Frequency and Monetary, where 1 = lowest and 4 = highest
f_labels = range(1, 5)

# Categorize 'Frequency' into quartiles with more transactions getting higher labels
f_groups = pd.qcut(rfm['Frequency'], q=4, labels=f_labels)

# Similar labels for Monetary as Frequency, more spending equals a higher label
m_labels = range(1, 5)

# Categorize 'Monetary' into quartiles, ensuring higher spenders receive higher labels
m_groups = pd.qcut(rfm['Monetary'], q=4, labels=m_labels)

# Applying RFM Labels to Data:
rfm['R'] = r_groups.values
rfm['F'] = f_groups.values
rfm['M'] = m_groups.values
```

Kmeans Clustering

In [ ]:
```
# Initialize a list to store inertia values for each k value
inertia = []

# Define the range of k values from 1 to 49 for KMeans clustering
K = range(1, 50)

# Extract RFM features for clustering
X = rfm[['R', 'F', 'M']]

# Loop through each k value to compute the KMeans clustering
for k in K:
    # Initialize the KMeans algorithm with the current number of clusters, 'k'
    kmeans = KMeans(n_clusters=k, random_state=0, init='k-means++', max_iter=10000)

    # Fit KMeans using the RFM feature set
    kmeans.fit(X)

    # Append the computed inertia to the inertia list
    inertia.append(kmeans.inertia_)
```

In [ ]:
```
# Set the size of the plot for better clarity and visibility
plt.figure(figsize=(10, 8))

# Plot inertia values against the number of clusters using a blue line with red markers
plt.plot(K, inertia, marker='o', linestyle='-', color='blue', linewidth=2, markersize=7, markerfacecolor='red')

# Set the label for the x-axis with specified font size and weight
plt.xlabel('Number of Clusters', fontsize=14, fontweight='bold')

# Set the label for the y-axis with specified font size and weight
plt.ylabel('Inertia', fontsize=14, fontweight='bold')

# Set the title of the plot with specified font size and weight
plt.title('Elbow Method for Determining Optimal k', fontsize=16, fontweight='bold')

# Enable grid lines for better readability, with specified style and color
plt.grid(True, which='both', linestyle='--', linewidth=0.5, color='gray', alpha=0.7)

# Set the font size of tick marks on the x-axis
plt.xticks(fontsize=12)

# Set the font size of tick marks on the y-axis
plt.yticks(fontsize=12)

# Adjust the layout to avoid any overlap of elements
plt.tight_layout()

# Display the plot
plt.show()
```



In [ ]:
```python
# Initialize the KneeLocator to find the elbow point in the K-means inertia plot
knee_locator = KneeLocator(K, inertia, curve='convex', direction='decreasing')

# Extract the elbow point, which suggests the optimal number of clusters
optimal_clusters = knee_locator.elbow

# Print the optimal number of clusters as determined by the elbow method
print(f"The optimal number of clusters is: {optimal_clusters}")
```

```
The optimal number of clusters is: 8
```

In [ ]:
```python
# Set the optimal number of clusters determined from previous analysis
optimal_clusters = 8

# Initialize the KMeans clustering algorithm with the specified number of clusters and a random state for reproducibility
kmeans = KMeans(n_clusters=optimal_clusters, random_state=0)

# Fit the KMeans model to the RFM data to perform the clustering
kmeans.fit(X)

# Assign the cluster labels generated by KMeans to a new column in the RFM DataFrame
rfm['Cluster'] = kmeans.labels_

# Print the count of customers in each cluster to see the distribution of the dataset across clusters
print(rfm['Cluster'].value_counts())
```

```
Cluster
3    1350
5     835
1     805
6     720
4     717
0     707
2     416
7     328
Name: count, dtype: int64
```

Cluster Summary

In [ ]:
```
# Group the RFM dataframe by 'Cluster' and calculate the average 'Recency' and 'Frequency',
# and both the average 'Monetary' value and count of entries in each cluster
cluster_summary = rfm.groupby('Cluster').agg({
    'Recency': 'mean',
    'Frequency': 'mean',
    'Monetary': ['mean', 'count']
}).round(1)

# Print the aggregated cluster summary
print(cluster_summary)
```

```
        Recency Frequency Monetary
           mean      mean     mean count
Cluster
0          120.8      13.2    240.1   707
1           30.9      82.1   1144.5   805
2          485.8      43.2    681.2   416
3           27.1     403.5   9340.6  1350
4          285.3     161.2   2913.9   717
5          540.6      14.8    225.1   835
6          187.5      40.5    535.7   720
7          141.9      31.7   3341.8   328
```

Customer Segment: Specific Marketing schemes

1. Cluster 0 - "Fresh Enthusiasts"

- Characteristics: Customers who have recently made purchases but do so infrequently and spend a moderate amount.
- Justification: The recent engagement suggests these customers are newcomers or returning after some time, showing potential for development into regular customers if engaged properly with onboarding communications and follow-up offers.
- Marketing Scheme: Initiate a "Welcome Aboard" program that includes a series of educational content about product uses and benefits, followed by a new member discount on their next purchase. Regular newsletters featuring new arrivals and customer testimonials can help maintain their interest and encourage repeat visits.

2. Cluster 1 - "Community Pillars"

- Characteristics: Customers with a short recency, high frequency, and above-average monetary value.
- Justification: This group forms the core of your business, consistently engaging and spending. Loyalty programs and frequent, personalized engagements can help sustain and enhance their high activity levels.
- Marketing Scheme: Launch a "Pillar Rewards" loyalty program where points can be accumulated not just from purchases but also from engaging with the brand on social media, writing reviews, or referring friends. Members can redeem points for exclusive discounts or access to special events.

3. Cluster 2 - "Lost Legends"

- Characteristics: Customers who haven't purchased in a long time, with low purchase frequency and moderate spending.
- Justification: The high recency and low frequency suggest a risk of churn. Re-engagement campaigns and incentives could reignite their interest and prevent them from fading away.
- Marketing Scheme: Roll out a "We Miss You" campaign featuring personalized emails highlighting changes or improvements made based on customer feedback, along with a "come-back" incentive such as a discount on their favorite products or limited-time free shipping.

4. Cluster 3 - "VIP Elites"

- Characteristics: Customers who shop very frequently and spend significantly more than those in other clusters.
- Justification: Their high frequency and monetary contributions mark them as top-tier customers who should receive VIP treatment, including exclusive offers and first access to new products to maintain their engagement.
- Marketing Scheme: Create an "Elite Club" with benefits such as a dedicated customer service line, early access to new products, and exclusive invitations to company events. Periodically send them surprise gift boxes curated with products based on their past purchases and preferences.

5. Cluster 4 - "Steady Patrons"

- Characteristics: Customers who shop with moderate frequency and have recent interactions, spending significant amounts.
- Justification: Reliable and steady, these customers are regular contributors to revenue. Tailored marketing and appreciation initiatives can help maintain their loyalty and possibly increase their transaction frequency.
- Marketing Scheme: Develop a "Patron Appreciation Day" that occurs several times a year, offering special promotions and rewards exclusive to this cluster. Implement a feedback loop where they can voice their product desires or improvements, enhancing their involvement and commitment to the brand.

6. Cluster 5 - "Silent Browsers"

- Characteristics: These customers exhibit the longest periods since their last purchase, infrequent transactions, and minimal spending.
- Justification: Their minimal engagement suggests they are at risk of becoming inactive. Targeted promotions and reactivation strategies may be necessary to bring them back into the fold.
- Marketing Scheme: Launch a "Reignite Your Interest" campaign using retargeting ads that showcase dynamic content related to items they've viewed but didn't purchase, along with a limited-time offer that expires within 48 hours to create urgency.

7. Cluster 6 - "Casual Shoppers"

- Characteristics: Customers with moderately high recency, moderate frequency of purchases, and moderate spending.
- Justification: These customers are somewhat engaged but not deeply committed. Enhancing their customer experience and offering personalized recommendations could elevate their spending and loyalty.
- Marketing Scheme: Introduce a "Mix & Match" event encouraging these customers to combine products for a discount. Use data-driven personalization to suggest items that complement their previous purchases, enhancing their shopping experience and increasing cart sizes.

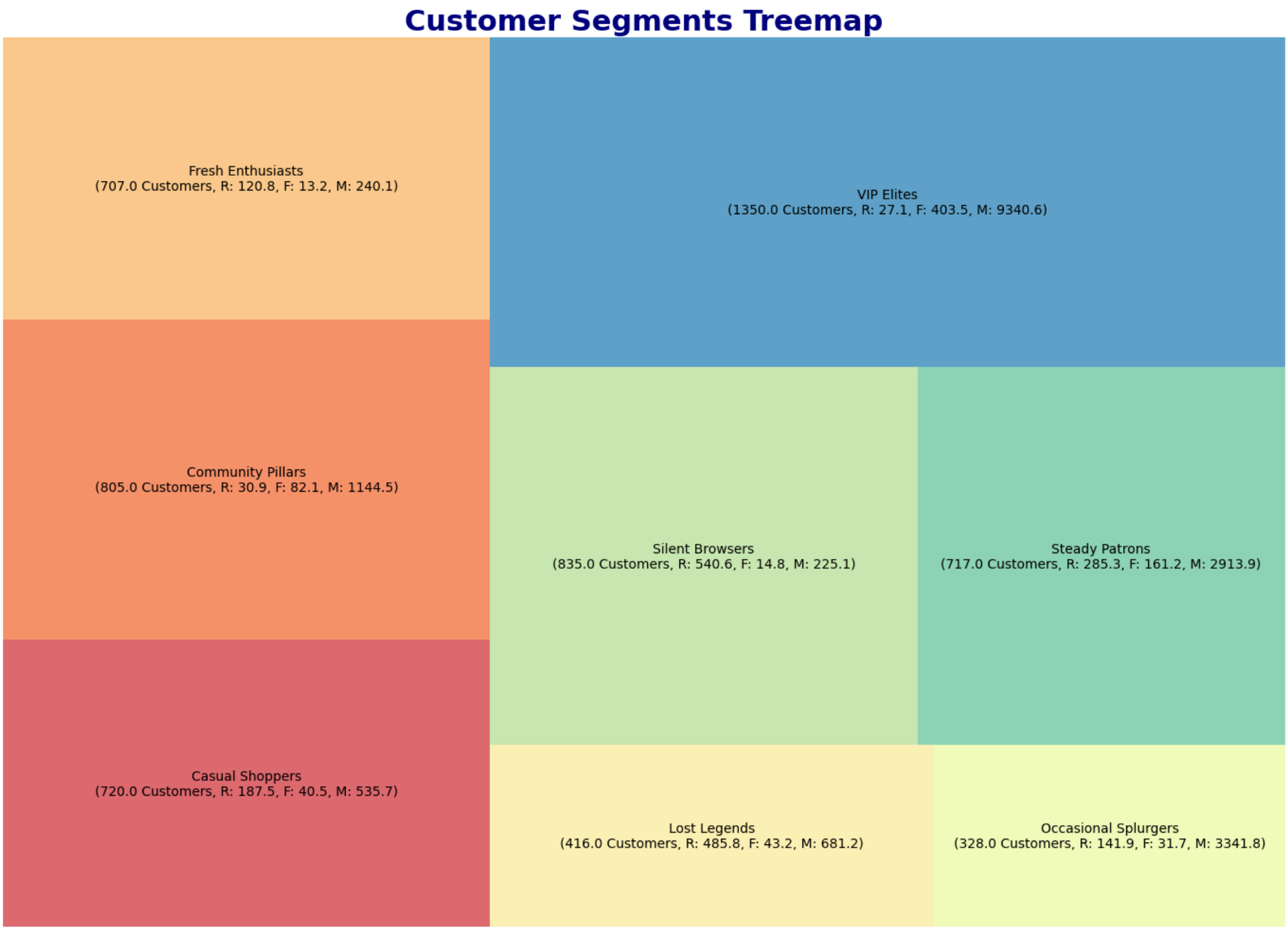
8. Cluster 7 - "Occasional Splurgers"

- Characteristics: Customers who shop infrequently but spend large amounts when they do.
- Justification: Although their engagements are infrequent, their high spending per transaction suggests they make significant purchases, potentially during sales or for major needs. Keeping them informed about major deals and product launches could stimulate their spending.
- Marketing Scheme: Offer an "Insider's Sale" where these customers get exclusive early access to sales, particularly during high-spending seasons or before major promotions go public. Implement a VIP consultation service that suggests high-ticket items that suit their tastes and needs, reinforcing their decision to spend on quality over quantity.

Customer Segments Visualisations

In [ ]:
```
# Assign descriptive names to each cluster for better readability and understanding
cluster_names = {
    0: 'Fresh Enthusiasts',
    1: 'Community Pillars',
    2: 'Lost Legends',
    3: 'VIP Elites',
    4: 'Steady Patrons',
    5: 'Silent Browsers',
    6: 'Casual Shoppers',
    7: 'Occasional Splurgers'
}
rfm['Segment'] = rfm['Cluster'].map(cluster_names)  # Map numeric clusters to descriptive labels

# Aggregate RFM data by the newly created Segment labels to compute average Recency, Frequency, and Monetary values
segment_summary = (
    rfm
    .groupby('Segment')
    .agg({
        'Recency': 'mean',  # Average days since last purchase
        'Frequency': 'mean',  # Average number of transactions
        'Monetary': 'mean'  # Average spend amount
    })
    .round(1)  # Round the averages to one decimal place for simplicity
)
# Count the number of customers in each segment
segment_summary['CustomerCount'] = rfm.groupby('Segment').size()

# Print the summary table of segments
print(segment_summary)

# Plotting setup for the treemap visualization of customer segments
# Define a distinct color palette for the treemap
color_palette = sns.color_palette('Spectral', n_colors=len(segment_summary))

# Configure the size of the figure for the treemap
plt.figure(figsize=(14, 10))

# Create the treemap using squarify, mapping segment size and labels to the respective segment data
squarify.plot(
    sizes=segment_summary['CustomerCount'],
    label=[f"{segment}\n({row['CustomerCount']} Customers, R: {row['Recency']}, F: {row['Frequency']}, M: {row['Monetary']:.1f}
           for segment, row in segment_summary.iterrows()],
    color=color_palette,
    alpha=0.8,
    pad=False
)

# Add a title to the treemap with customization for readability
plt.title("Customer Segments Treemap", fontsize=22, fontweight="bold", color='navy')

# Hide the axes for a cleaner visualization
plt.axis('off')

# Adjust layout to ensure labels are not truncated
plt.tight_layout()

# Display the treemap
plt.show()
```

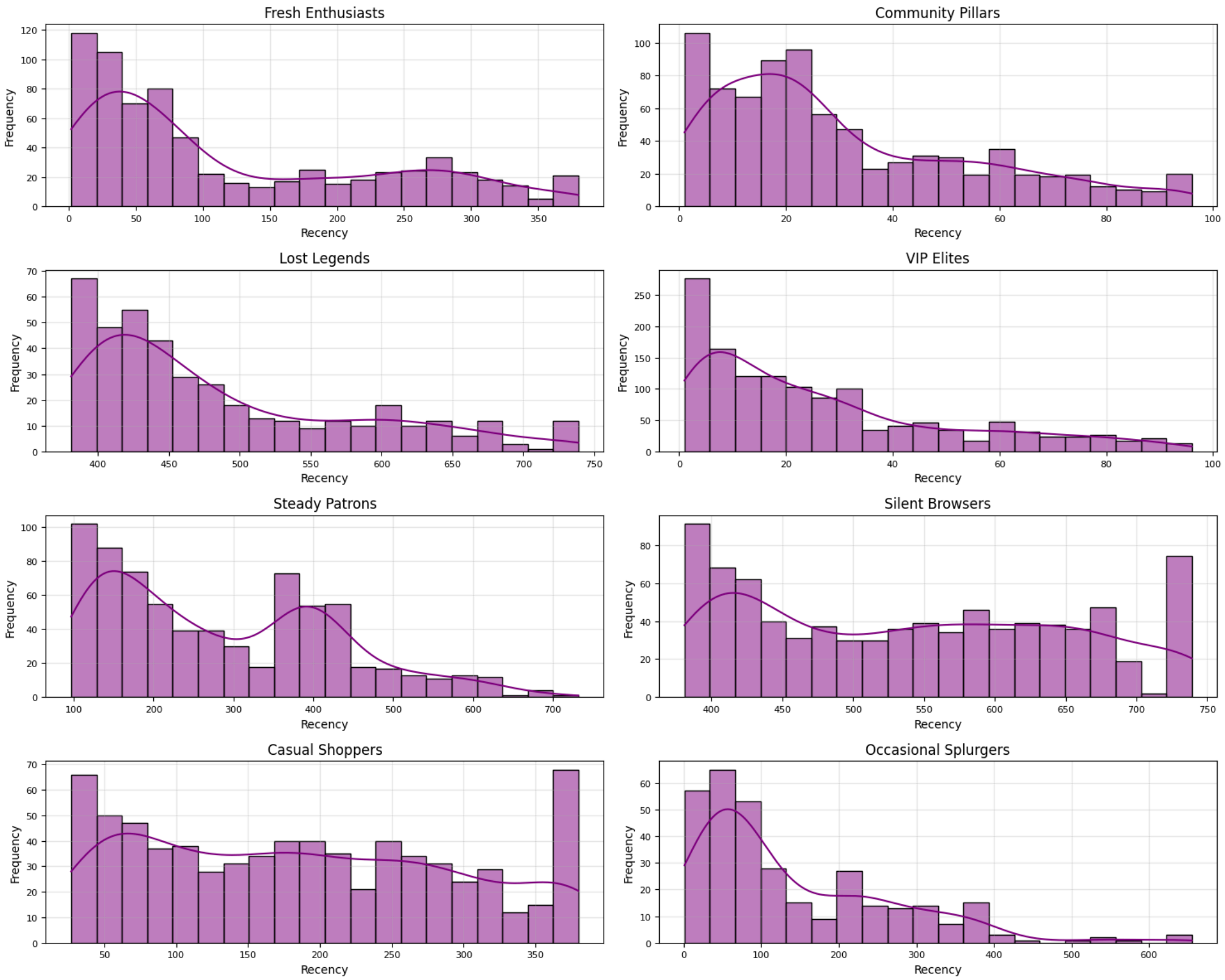
Segment	Recency	Frequency	Monetary	CustomerCount
Casual Shoppers	187.5	40.5	535.7	720
Community Pillars	30.9	82.1	1144.5	805
Fresh Enthusiasts	120.8	13.2	240.1	707
Lost Legends	485.8	43.2	681.2	416
Occasional Splurgers	141.9	31.7	3341.8	328
Silent Browsers	540.6	14.8	225.1	835
Steady Patrons	285.3	161.2	2913.9	717
VIP Elites	27.1	403.5	9340.6	1350



In [ ]:

```python
# Define the total number of clusters for analysis
num_clusters = 8

# Calculate the number of rows needed in the subplot grid
num_rows = num_clusters // 2 if num_clusters % 2 == 0 else (num_clusters // 2 + 1)

# Initialize a figure with dynamic sizing based on the number of subplots needed
plt.figure(figsize=(15, num_rows * 3))

# Loop through each cluster to create individual histograms for the 'Recency' attribute
for cluster_id in range(num_clusters):
    # Create a subplot for each cluster
    ax = plt.subplot(num_rows, 2, cluster_id + 1)

    # Plot a histogram of 'Recency' for the current cluster with kernel density estimate
    sns.histplot(rfm[rfm['Cluster'] == cluster_id]['Recency'], bins=20, kde=True, color='purple')

    # Set the title of each subplot using the cluster's descriptive name, with fallback for unknown clusters
    plt.title(f'{cluster_names.get(cluster_id, "Unknown Cluster")}', fontsize=12)

    # Label the x-axis as 'Recency'
    plt.xlabel('Recency', fontsize=10)

    # Label the y-axis as 'Frequency'
    plt.ylabel('Frequency', fontsize=10)

    # Add grid lines to the plot for better readability, with reduced opacity
    plt.grid(True, alpha=0.3)

    # Adjust the size of tick labels for clarity
    ax.tick_params(axis='both', which='major', labelsize=8)

# Adjust the layout to ensure all subplot elements are clearly visible and non-overlapping
```

```
plt.tight_layout()

# Display the entire grid of histograms
plt.show()
```



Interpretation of each segment's histogram:

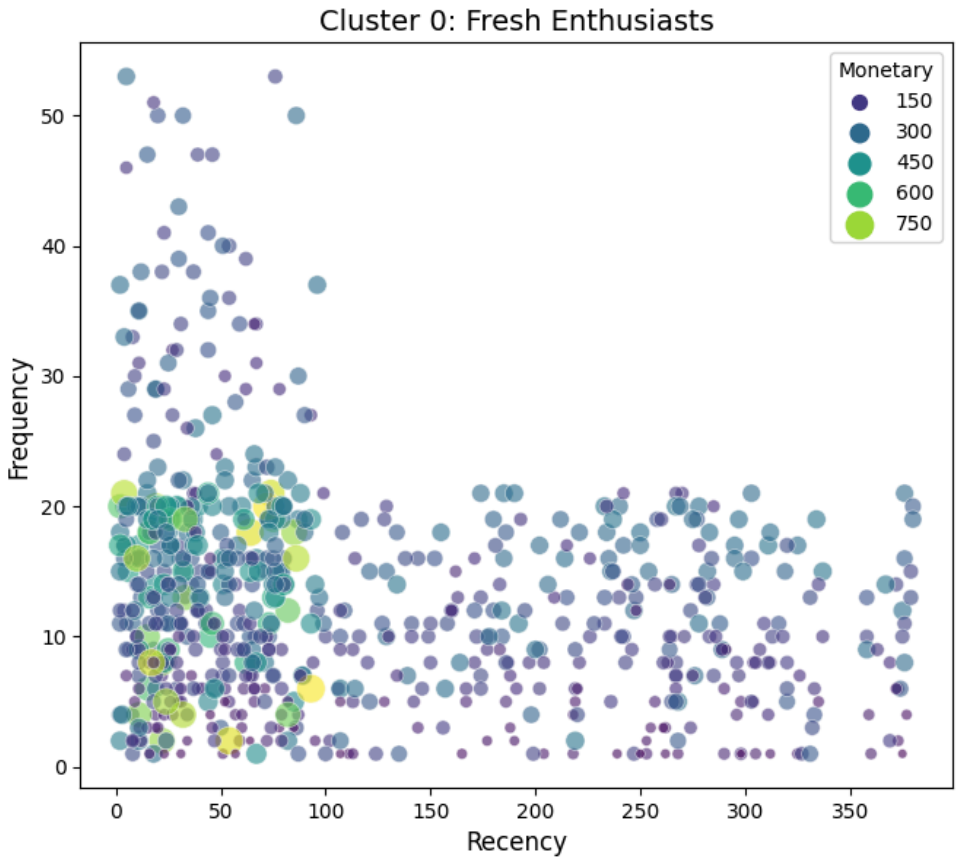
1. Fresh Enthusiasts and Community Pillars: Both show a skew towards lower recency values, which means customers in these segments have made purchases more recently. This indicates a higher level of current engagement with the company.
2. Lost Legends and Silent Browsers: These segments have histograms skewed towards higher recency values, suggesting these customers haven't made recent purchases and may be disengaged or lost.
3. VIP Elites: The distribution is heavily skewed left with a long tail to the right, meaning most VIP Elites have made purchases very recently, but there's a small group that hasn't purchased in a while.
4. Steady Patrons: This segment has a relatively even distribution across a range of recency values, though there are peaks indicating that certain time frames are more common for the last purchase.
5. Casual Shoppers: This group displays a wide range of recency values with several peaks, suggesting varied engagement levels. The peaks could indicate periods of increased purchasing activity.
6. Occasional Splurgers: This segment's histogram is quite spread out but shows that most of the customers have medium to high recency values, suggesting infrequent purchases.

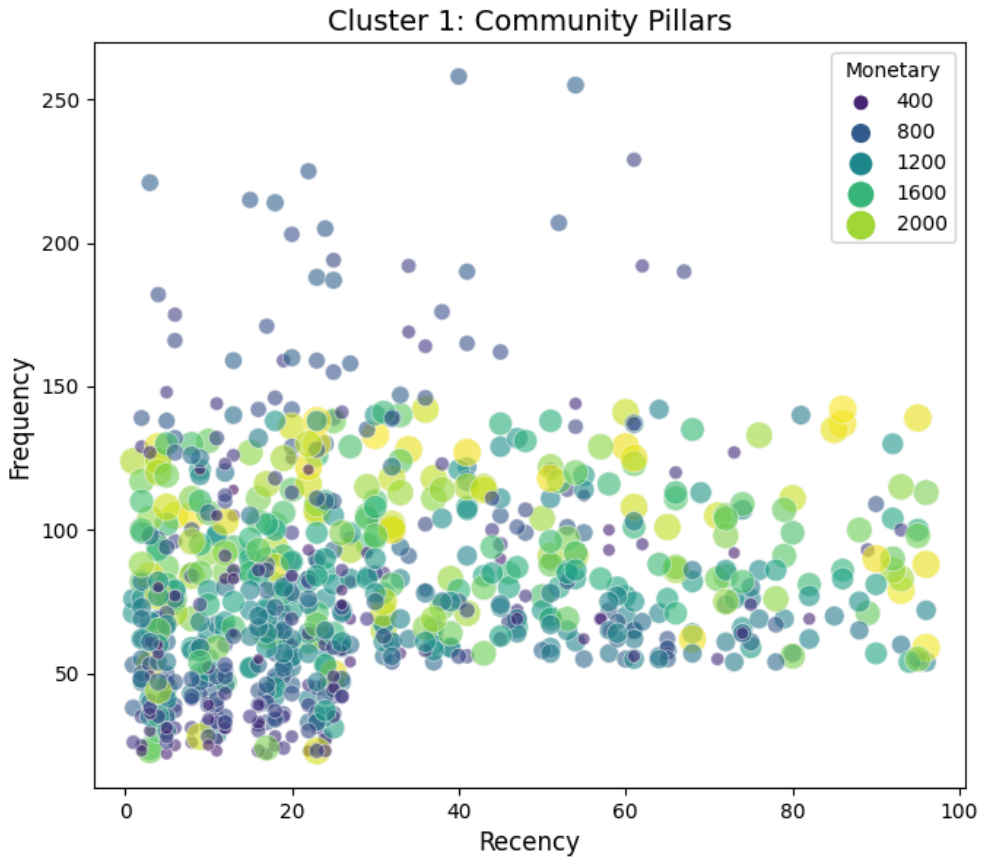
Each histogram also includes a kernel density estimate (KDE), shown by the smooth line overlaying the bars, which estimates the probability density function of the recency variable.

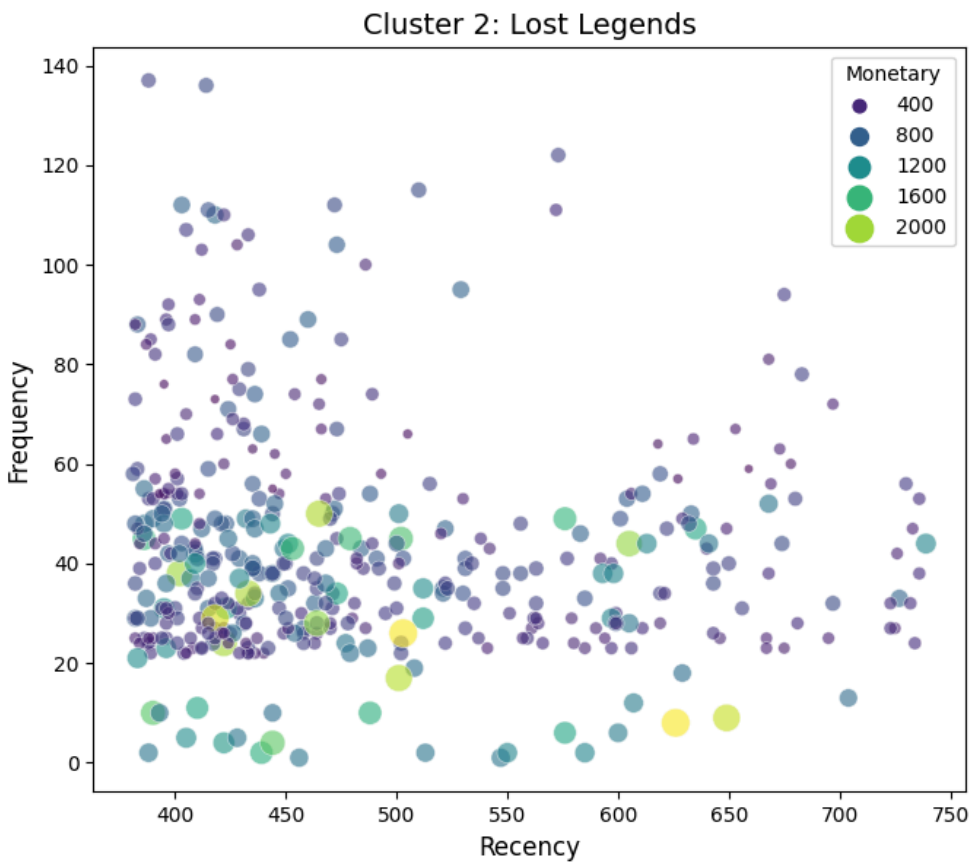
Overall, this visualization help a business tailor its marketing strategies to re-engage lost customers, retain engaged ones, or even convert new customers into loyal patrons based on their purchase history.

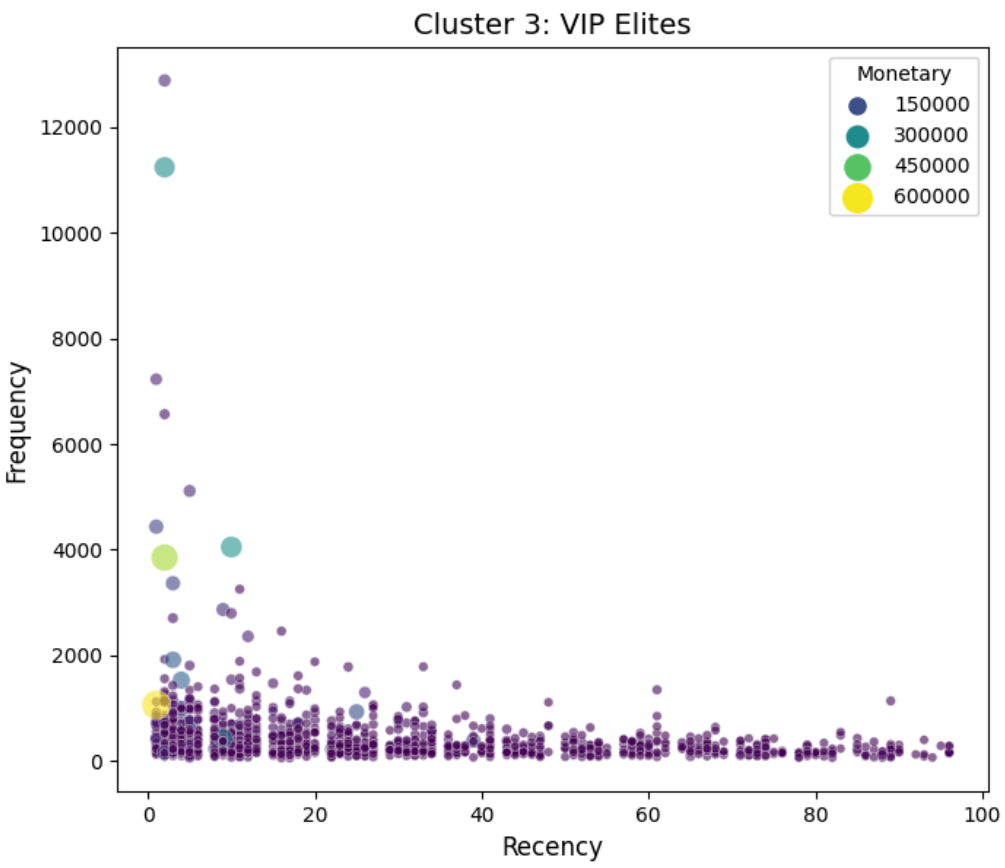
In [ ]:
```
# Define the total number of clusters for analysis
num_clusters = 8
```

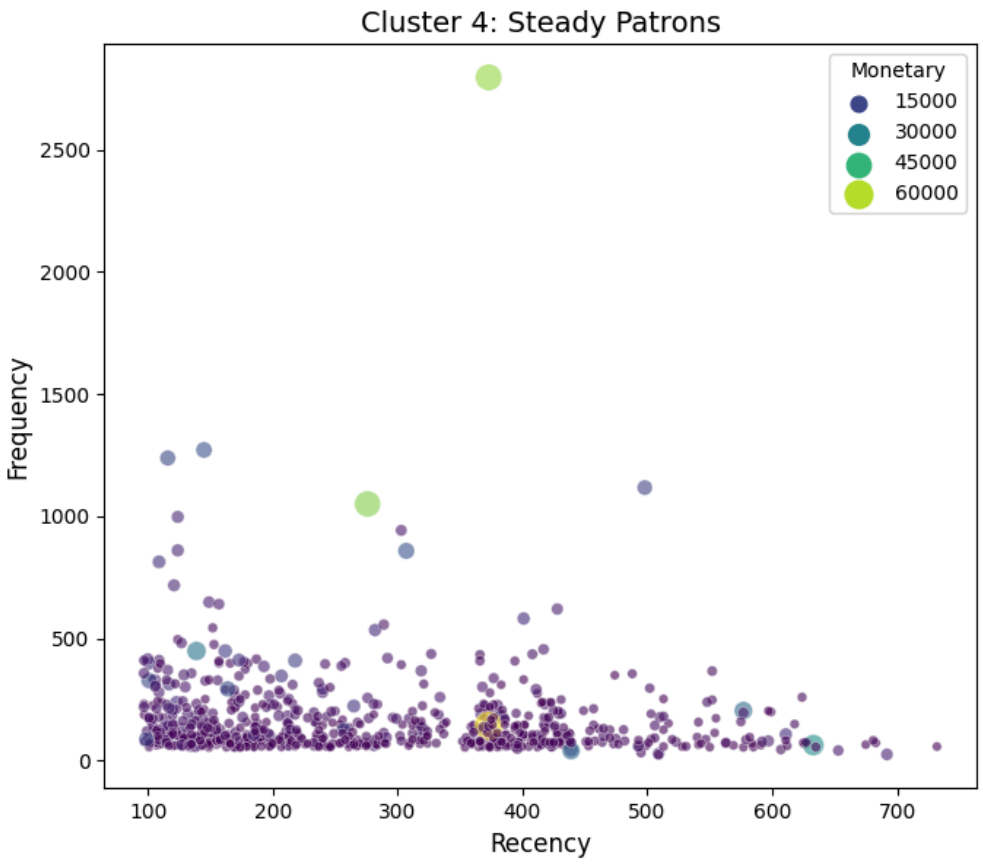
```
# Calculate the number of rows needed for the subplot layout, ensuring there's an extra row if the number of clusters is odd
num_rows = (num_clusters + 1) // 2

# Create a figure with subplots arranged in rows and columns, adjusting the size for better visibility
fig, axes = plt.subplots(num_rows, 2, figsize=(14, 24))

# Flatten the axes array for easier access in the loop
axes = axes.ravel()

# Initialize a list to collect legend elements for a common legend after plotting
legend_elements = []

# Loop through each cluster to plot the RFM data
for cluster_id in range(num_clusters):
    # Filter the data for the current cluster
    cluster_data = rfm[rfm['Cluster'] == cluster_id]

    # Plot a scatterplot for the current cluster's data, mapping Recency to x-axis, Frequency to y-axis,
    # and using Monetary value for color and size of the points
    ax = sns.scatterplot(
        data=cluster_data,
        x='Recency',
        y='Frequency',
        hue='Monetary',
        size='Monetary',
        sizes=(20, 200),
        palette='viridis',
        alpha=0.6,
        ax=axes[cluster_id]
    )

    # Set the title for the subplot using the predefined cluster name, enhancing readability with font size adjustments
    axes[cluster_id].set_title(f'Cluster {cluster_id}: {cluster_names[cluster_id]}', fontsize=14)

    # Set axis labels with adjusted font size for clarity
    axes[cluster_id].set_xlabel('Recency', fontsize=12)
    axes[cluster_id].set_ylabel('Frequency', fontsize=12)

    # Collect legend handles and labels from the last plot if it's the last iteration, for later use in a common legend
    if cluster_id == num_clusters - 1:
        legend_elements = ax.get_legend_handles_labels()

# If the number of clusters is odd, hide the last subplot to prevent an empty plot from displaying
if num_clusters % 2 != 0:
    fig.delaxes(axes[-1])

# Adjust the layout to ensure there's no overlap of subplot elements
plt.tight_layout()

# Show the final plot
plt.show()
```


Cluster 0: Fresh Enthusiasts
Monetary
150
300
450
600
750
Frequency
Recency

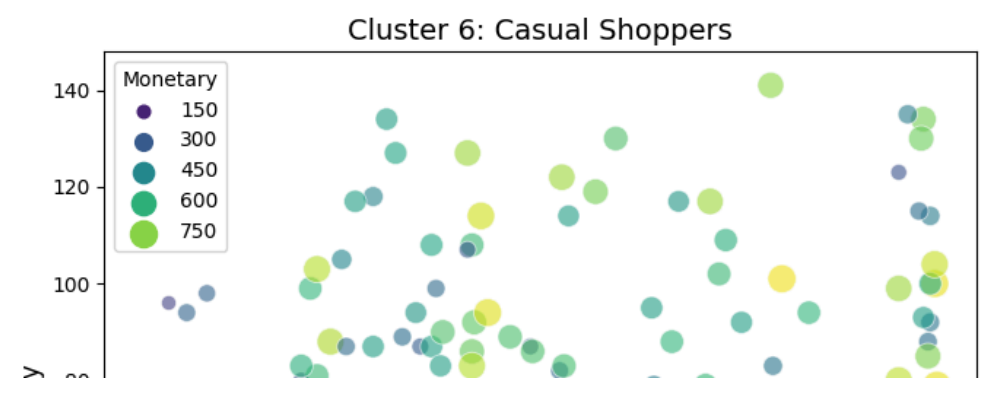

Cluster 1: Community Pillars
Monetary
400
800
1200
1600
2000
Frequency
Recency

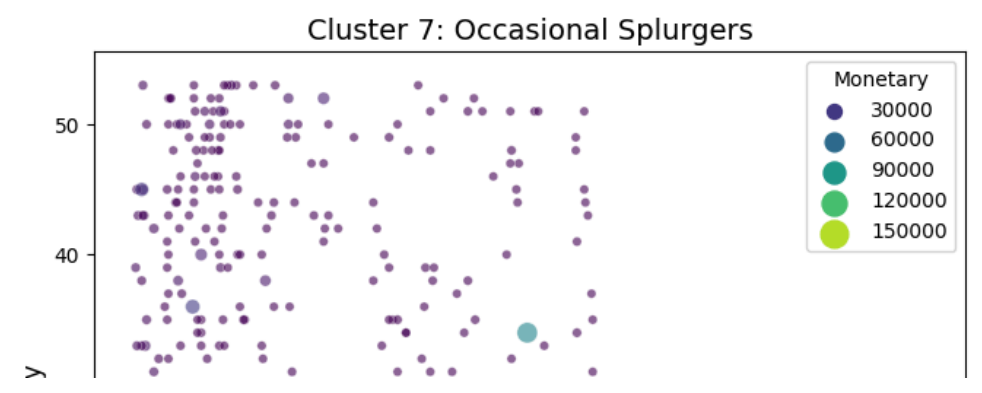

Cluster 2: Lost Legends
Monetary
400
800
1200
1600
2000
Frequency
Recency

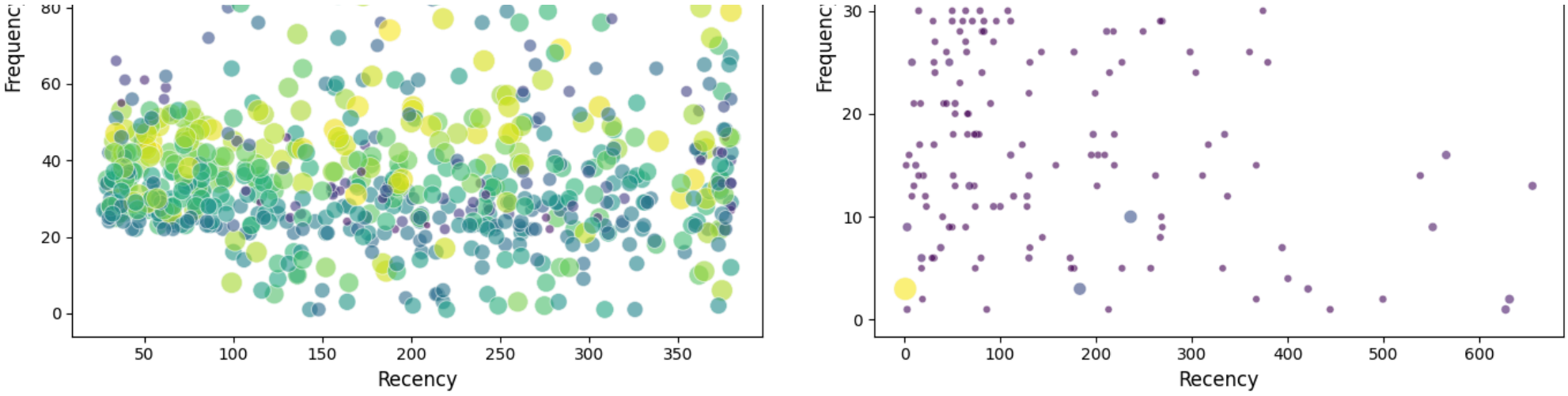

Cluster 3: VIP Elites
Monetary
150000
300000
450000
600000
Frequency
Recency


Cluster 4: Steady Patrons
Monetary
15000
30000
45000
60000
Frequency
Recency


Cluster 5: Silent Browsers
Monetary
150
300
450
600
750
Frequency
Recency


Cluster 6: Casual Shoppers
Monetary
150
300
450
600
750


Cluster 7: Occasional Splurgers
Monetary
30000
60000
90000
120000
150000



Each plot visualizes two of the RFM dimensions: Recency (x-axis) and Frequency (y-axis), with the size and color of the points corresponding to the Monetary value.

1. Cluster 0 - Fresh Enthusiasts: They have a wide range of recency and frequency, with most monetary values on the lower end, suggesting these customers engage fairly regularly with relatively lower spend amounts.

2. Cluster 1 - Community Pillars: Customers here show recent engagement and high frequency, with most demonstrating moderate to high monetary spending. This cluster represents a highly engaged and valuable segment.

3. Cluster 2 - Lost Legends: This segment shows a large spread in both recency and frequency, with most monetary values being moderate, indicating these were once engaged customers who may need re-engagement strategies.

4. Cluster 3 - VIP Elites: Characterized by a high frequency of purchases and significant monetary spend, these customers are likely the most valuable, with very recent engagement.

5. Cluster 4 - Steady Patrons: They exhibit a broad range of recency and a moderate frequency of purchases. Monetary values vary widely, indicating diverse spending behaviors within this group.

6. Cluster 5 - Silent Browsers: These customers engage infrequently and recent interactions vary, with most monetary values being lower. They may require incentives to increase both their engagement and spending.

7. Cluster 6 - Casual Shoppers: This group shows sporadic engagement, with a mix of recent and less recent interactions and a wide range of spend amounts, implying occasional shopping with varying monetary value.

8. Cluster 7 - Occasional Splurgers: Typically, they have low frequency and a spread of recency, but the plot points suggest high monetary values, which might imply that when they do shop, they spend large amounts, possibly on luxury or high-ticket items.

Each plot provides insights into the shopping behavior patterns of customers within that segment, allowing for targeted marketing strategies.

For example, "VIP Elites" might be rewarded for their loyalty, while "Lost Legends" might be enticed back with re-engagement campaigns. The Monetary dimension, represented by point size and color, suggests a segment's overall value and can help prioritize efforts for customer retention and engagement.

Customer-Specific Recommendations Using Neural Collaborative Filtering Model(NCF)

Data-preprocessing

In [ ]:
```
# Merge RFM and Clusters with original data
data_with_rfm = data.merge(rfm, on='Customer ID')

# Examine unique values in the 'StockCode' column
unique_stock_codes = data_with_rfm['StockCode'].unique()

# Encode users
user_encoder = LabelEncoder()
data_with_rfm['user_id'] = user_encoder.fit_transform(data_with_rfm['Customer ID'])

# Convert all values in the 'StockCode' column to strings
data_with_rfm['StockCode'] = data_with_rfm['StockCode'].astype(str)

# Encode items
item_encoder = LabelEncoder()
data_with_rfm['item_id'] = item_encoder.fit_transform(data_with_rfm['StockCode'])
```

Baseline Model for recommendation

The baseline model is a simple non-machine learning approach that recommends the top N items based solely on their global average 'Quantity' across all users. It does not learn from the data in the way that deep learning models do; instead, it relies on a straightforward statistical metric (mean) and does not involve any training or predictive modeling that characterizes deep learning techniques.

In [ ]:
```
import pandas as pd
from sklearn.metrics import precision_score, recall_score, f1_score
```

```python
from sklearn.model_selection import train_test_split

# Calculate the global average 'quantity' for each item across all users
item_global_avg = data_with_rfm.groupby('item_id')['Quantity'].mean()

# Predict the top N items for all users based on global average quantity
# Here, N is set to the average number of items per user but you can set a different threshold
N = int(data_with_rfm.groupby('user_id').size().mean())

# Get the item indices sorted by global average quantity
top_N_items = item_global_avg.nlargest(N).index.tolist()

# Create a dictionary where the key is the user_id and the value is the top N items
baseline_recommendations = {user_id: top_N_items for user_id in data_with_rfm['user_id'].unique()}

# Split the original data into training and test sets
train_data, test_data = train_test_split(data_with_rfm, test_size=0.3, random_state=42)

# Create a function to calculate true positives for a user
def calculate_true_positives(user_id, true_items, predicted_items):
    # True items are the items in the test set
    true_items_set = set(true_items)
    # Predicted items are the top N items from the baseline
    predicted_items_set = set(predicted_items)
    # True positives are the intersection of the two sets
    return len(true_items_set & predicted_items_set)

# Calculate the number of true positives for each user in the test set
true_positives = test_data.groupby('user_id').apply(lambda x: calculate_true_positives(x.name, x['item_id'], baseline_recommend

# Calculate precision, recall, and F1-score for the baseline
# Note: We assume binary relevance; an item is relevant if it's one of the purchased items in the test set
precision = true_positives.sum() / (N * len(baseline_recommendations))
recall = true_positives.sum() / test_data.groupby('user_id')['item_id'].nunique().sum()
f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

print(f'Baseline Precision: {precision}')
print(f'Baseline Recall: {recall}')
print(f'Baseline F1-Score: {f1}')
```

```
Baseline Precision: 0.0025456794232111323
Baseline Recall: 0.010827026370411058
Baseline F1-Score: 0.004122148302831111
```

These scores are quite low, which is not uncommon for a baseline model, especially one as simple as a global average without any personalization or deep learning involved. The primary purpose of such a baseline is to establish a minimum performance threshold.

Improved Model for recommendation system

Interaction Matrix Creation

In [ ]:
```python
# Aggregate the purchase frequency and create the interaction matrix
interaction_matrix = data_with_rfm.groupby(['user_id', 'item_id'])['Frequency'].sum().unstack(fill_value=0)

# Normalize the interaction matrix
max_freq = interaction_matrix.max().max()
interactions_scaled = torch.FloatTensor(interaction_matrix.values / max_freq)
```

Data Splitting

In [ ]:
```python
# Split data into training and validation sets
num_users = interactions_scaled.shape[0]
train_indices, val_indices = train_test_split(range(num_users), test_size=0.2, random_state=42)
```

Model Definition

In [ ]:
```python
class EnhancedCFModel(nn.Module):
    def __init__(self, num_users, num_items, embedding_size=100):
        super(EnhancedCFModel, self).__init__()
        self.user_embedding = nn.Embedding(num_users, embedding_size)
        self.fc1 = nn.Linear(embedding_size, 100)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)
        self.fc2 = nn.Linear(100, num_items)

    # Generate embeddings for input user IDs
    def forward(self, user_ids):
        user_embedding = self.user_embedding(user_ids)
        x = self.relu(self.fc1(user_embedding))
        x = self.dropout(x)
        scores = self.fc2(x)
        return scores
```

Model Training Setup

In [ ]:
```python
# Initialize model
num_items = interactions_scaled.shape[1]
model = EnhancedCFModel(num_users, num_items)

# Loss and Optimizer
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma=0.9)

# Metrics and storage
train_losses, val_losses = [], []

 # Training settings
epochs = 50
best_val_loss = float('inf')
patience, trials = 3, 0
```

Training Loop

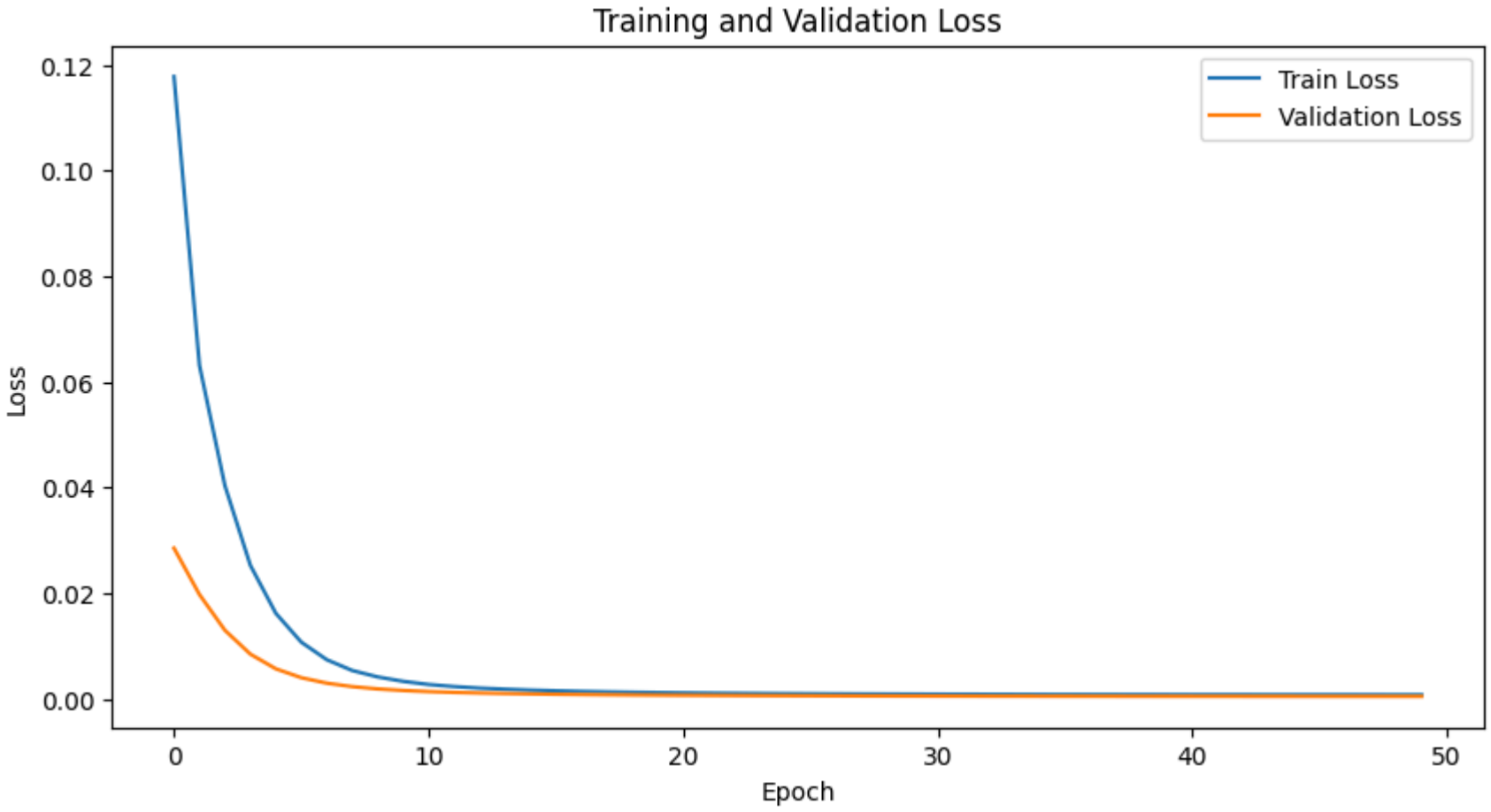
In [ ]:
```python
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    train_predictions = model(torch.tensor(train_indices, dtype=torch.long))
    train_loss = criterion(train_predictions, interactions_scaled[train_indices])
    train_loss.backward()
    optimizer.step()
    scheduler.step()
    train_losses.append(train_loss.item())


    # Set model to evaluation mode
    model.eval()
    with torch.no_grad():
        val_predictions = model(torch.tensor(val_indices, dtype=torch.long))
        val_loss = criterion(val_predictions, interactions_scaled[val_indices])
        val_losses.append(val_loss.item())

    # Output training progress
    print(f'Epoch {epoch+1}/{epochs}, Train Loss: {train_loss.item()}, Val Loss: {val_loss.item()}')

    if val_loss.item() < best_val_loss:
        best_val_loss = val_loss.item()
        trials = 0
    else:
        # Increment trials counter if validation loss didn't improve
        trials += 1
        # Stop early if no improvement in a given number of epochs
        if trials >= patience:
            print("Early stopping triggered.")
            break
```

```
Epoch 1/50, Train Loss: 0.11777940392494202, Val Loss: 0.028536614030599594
Epoch 2/50, Train Loss: 0.06313373893499374, Val Loss: 0.01975804939866066
Epoch 3/50, Train Loss: 0.04035532847046852, Val Loss: 0.012989012524485588
Epoch 4/50, Train Loss: 0.025273801758885384, Val Loss: 0.008471275679767132
Epoch 5/50, Train Loss: 0.016196923330426216, Val Loss: 0.0056954240426421165
Epoch 6/50, Train Loss: 0.010737297125160694, Val Loss: 0.00401362543925643
Epoch 7/50, Train Loss: 0.007420742884278297, Val Loss: 0.0029773840215057135
Epoch 8/50, Train Loss: 0.0053725470788776875, Val Loss: 0.002320739207789302
Epoch 9/50, Train Loss: 0.0041601755656301975, Val Loss: 0.0018881650175899267
Epoch 10/50, Train Loss: 0.003317690221592784, Val Loss: 0.0015921351732686162
Epoch 11/50, Train Loss: 0.0027301248628646135, Val Loss: 0.0013820010935887694
Epoch 12/50, Train Loss: 0.0023335993755608797, Val Loss: 0.0012269640574231744
Epoch 13/50, Train Loss: 0.0020419296342879534, Val Loss: 0.0011086444137617946
Epoch 14/50, Train Loss: 0.0018335601780563593, Val Loss: 0.001015597372315824
Epoch 15/50, Train Loss: 0.0016612695762887597, Val Loss: 0.0009405602468177676
Epoch 16/50, Train Loss: 0.0015318195801228285, Val Loss: 0.0008787748520262539
Epoch 17/50, Train Loss: 0.00142224773298949, Val Loss: 0.0008270881953649223
Epoch 18/50, Train Loss: 0.0013312632218003273, Val Loss: 0.000783337454777211
Epoch 19/50, Train Loss: 0.0012581354239955544, Val Loss: 0.0007459136541001499
Epoch 20/50, Train Loss: 0.0011994339292868972, Val Loss: 0.0007136156200431287
Epoch 21/50, Train Loss: 0.001146029564552009, Val Loss: 0.0006855717510916293
Epoch 22/50, Train Loss: 0.0011004378320649266, Val Loss: 0.0006610995042137802
Epoch 23/50, Train Loss: 0.0010620286921039224, Val Loss: 0.0006396574899554253
Epoch 24/50, Train Loss: 0.0010304980678483844, Val Loss: 0.0006208115955814719
Epoch 25/50, Train Loss: 0.0009978621965274215, Val Loss: 0.0006042081513442099
Epoch 26/50, Train Loss: 0.0009763523703441024, Val Loss: 0.0005895442445762455
Epoch 27/50, Train Loss: 0.0009499091538600624, Val Loss: 0.0005765689420513809
Epoch 28/50, Train Loss: 0.0009318649535998702, Val Loss: 0.0005650658858940005
Epoch 29/50, Train Loss: 0.0009169080876745284, Val Loss: 0.0005548587068915367
Epoch 30/50, Train Loss: 0.0009018287528306246, Val Loss: 0.0005457854713313282
Epoch 31/50, Train Loss: 0.0008879380184225738, Val Loss: 0.0005377114866860211
Epoch 32/50, Train Loss: 0.0008779088384471834, Val Loss: 0.0005305179511196911
Epoch 33/50, Train Loss: 0.0008687735535204411, Val Loss: 0.0005241024191491306
Epoch 34/50, Train Loss: 0.0008581543806940317, Val Loss: 0.0005183774628676474
Epoch 35/50, Train Loss: 0.000847283867187798, Val Loss: 0.0005132626392878592
Epoch 36/50, Train Loss: 0.0008415755582973361, Val Loss: 0.0005086880992166698
Epoch 37/50, Train Loss: 0.0008330178097821772, Val Loss: 0.0005045957514084876
Epoch 38/50, Train Loss: 0.0008293089922517538, Val Loss: 0.0005009308224543929
Epoch 39/50, Train Loss: 0.0008232076070271432, Val Loss: 0.0004976465133950114
Epoch 40/50, Train Loss: 0.000820953631773591, Val Loss: 0.00049470120575279
Epoch 41/50, Train Loss: 0.0008132729562930763, Val Loss: 0.0004920574720017612
Epoch 42/50, Train Loss: 0.0008113401127047837, Val Loss: 0.0004896841128356755
Epoch 43/50, Train Loss: 0.0008089556358754635, Val Loss: 0.00048755292664282024
Epoch 44/50, Train Loss: 0.0008045242284424603, Val Loss: 0.0004856379528064281
Epoch 45/50, Train Loss: 0.0008038366213440895, Val Loss: 0.00048391599557362497
Epoch 46/50, Train Loss: 0.0007987763965502381, Val Loss: 0.0004823668859899044
Epoch 47/50, Train Loss: 0.0007998208166100085, Val Loss: 0.0004809731035493314
Epoch 48/50, Train Loss: 0.0007974845357239246, Val Loss: 0.00047971835010685027
Epoch 49/50, Train Loss: 0.0007942947559058666, Val Loss: 0.00047858853940851986
Epoch 50/50, Train Loss: 0.0007940334617160261, Val Loss: 0.0004775706329382956
```

Visualization

In [ ]:

```python
# Plotting the training and validation loss
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
plt.show()
```



Evaluation Matrics

In [ ]:
```
rmse = torch.sqrt(criterion(val_predictions, interactions_scaled[val_indices])).item()
print(f'Validation RMSE: {rmse}')
```

```
Validation RMSE: 0.021853389218449593
```

In [ ]:
```
# Assuming binary classification with a threshold
threshold = 0.5
predicted_labels = (val_predictions >= threshold).int()
true_labels = (interactions_scaled[val_indices] > 0).int()

# Calculate precision, recall, and F1-score using sklearn's functions
from sklearn.metrics import precision_score, recall_score, f1_score

precision = precision_score(true_labels.view(-1).cpu(), predicted_labels.view(-1).cpu(), average='macro')
recall = recall_score(true_labels.view(-1).cpu(), predicted_labels.view(-1).cpu(), average='macro')
f1 = f1_score(true_labels.view(-1).cpu(), predicted_labels.view(-1).cpu(), average='macro')

print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1-Score: {f1}')
```

```
Precision: 0.4915660984756675
Recall: 0.5
F1-Score: 0.4957471813844292
```

These metrics indicate how well our recommendation system is performing in terms of its ability to provide relevant items to users:

- Precision of approximately 0.49 implies that about 49% of the items recommended by the model are relevant to the users. In other words, when the model recommends an item, there's almost a 50-50 chance that it's actually of interest to the user.
- Recall of 0.5 means that the model is able to capture 50% of the relevant items in its recommendations. So, for every two relevant items, one is likely to be recommended by the model.
- F1-Score of approximately 0.496 is the harmonic mean of precision and recall. This score takes both false positives and false negatives into account and is particularly useful when you want to find a balance between precision and recall. An F1-Score of 0.496 suggests that the model is fairly balanced in terms of precision and recall.

These scores might suggest that the model has room for improvement. With model hyper tuning and some feature engineering, we can improve our model further.

Comparison to Baseline Model:

- Our enhanced model's precision (49.16%) and recall (50%) are significantly higher than those of the baseline model (Precision: 0.25%, Recall: 1.08%), indicating that your model is much more effective at identifying and recommending relevant items.
- The F1-score improvement from 0.41% in the baseline to 49.57% in your our clearly demonstrates a substantial increase in the overall effectiveness of the recommendation system.

The marked improvement in all metrics with your NCF model shows that incorporating user and item embeddings, non-linear transformations (via neural networks), and fine-tuning through deep learning significantly enhances recommendation quality.

Generating Recommendations for Customers

In [ ]:
```
model.eval()
with torch.no_grad():
    # Generate predictions for all users
    all_predictions = model(torch.arange(num_users))
    # Get the indices of the top five recommended items for each user
    top_five_items = torch.topk(all_predictions, 5, dim=1)[1]

# Convert numeric item indices to actual item StockCodes
top_five_products = [item_encoder.inverse_transform(items.numpy()) for items in top_five_items]

def get_recommendations(customer_id):
    # Check if the customer exists in the dataset
    if customer_id not in data_with_rfm['Customer ID'].values:
        print(f"Customer ID {customer_id} not found. Providing default recommendations.")
        # Provide default recommendations if customer ID is not found
        default_recommendations = item_encoder.inverse_transform(np.argsort(-all_predictions.mean(dim=0).numpy())[:5])
        return None, default_recommendations  # Return no cluster info

    # Retrieve cluster assignment for the customer
    cluster = data_with_rfm[data_with_rfm['Customer ID'] == customer_id]['Cluster'].iloc[0]

    # Find the user index for recommendation
    user_index = user_encoder.transform([customer_id])[0]
    # Get the top five recommended products for this user
    recommendations = top_five_products[user_index]
    return cluster, recommendations
```

Try it

In [ ]:
```
# Example usage: Request recommendations for a specific customer
customer_id = 13078  # Replace this with any customer ID to get 5 recommendation products
recommendation_info = get_recommendations(customer_id)
if recommendation_info:
    # Print the recommendations if available
    cluster, recommended_products = recommendation_info
    print(f'Customer ID: {customer_id}, Cluster: {cluster}, Recommended Products: {recommended_products}')
else:
    # Handle cases where recommendations cannot be generated
    print("Recommendation could not be generated.")
```

```
Customer ID: 13078, Cluster: 3, Recommended Products: ['22649' '37432' '84617' '16162L' '85194S']
```

Utilizing customer segmentation alongside a recommendation system effectively enhances customer engagement and business efficiency. For example, Customer 13078, who belongs to Cluster 3, "VIP Elites," receives not only personalized product recommendations—such as '22649', '84617', etc.—but also tailored offers and services. This approach ensures marketing strategies and recommendations are highly relevant and appealing to specific needs and preferences. By aligning offers with the distinct characteristics of each segment, businesses can maximize engagement, improve conversion rates, optimize marketing resources, and strengthen customer loyalty, which drives sustainable growth.