Workshop2 - K-Means Clustering

June 5, 2025

This notebook demonstrates the application of the **KMeans clustering** algorithm to segment customer data. We'll walk through data preparation, model application, and a crucial method for determining the optimal number of clusters. The goal is to identify distinct customer groups based on their purchasing behavior — **Recency**, **Frequency**, and **MonetaryValue** — to enable targeted business strategies such as personalized marketing, loyalty programs, and churn prevention.

Contributors: Sanuja Vihanga Senadeera, Naveen Karan Krishna, Thi Anh Tram Le, Mayra Geraldine Reinoso Varon, Justin Kyle Pedro

1. Environment Setup: Installing Necessary Libraries

This initial step ensures that all required Python libraries, especially **yellowbrick** for enhanced visualization, are available in our environment. Running this command once sets up our toolbox for the analysis.

In [3]:
```
!pip install yellowbrick
```

```
Requirement already satisfied: yellowbrick in c:\users\tgdd\anaconda3\lib\site-packa
ges (1.5)
Requirement already satisfied: matplotlib!=3.0.0,>=2.0.2 in c:\users\tgdd\anaconda3
\lib\site-packages (from yellowbrick) (3.9.2)
Requirement already satisfied: scipy>=1.0.0 in c:\users\tgdd\anaconda3\lib\site-pack
ages (from yellowbrick) (1.13.1)
Requirement already satisfied: scikit-learn>=1.0.0 in c:\users\tgdd\anaconda3\lib\si
te-packages (from yellowbrick) (1.5.1)
Requirement already satisfied: numpy>=1.16.0 in c:\users\tgdd\anaconda3\lib\site-pac
kages (from yellowbrick) (1.26.4)
Requirement already satisfied: cycler>=0.10.0 in c:\users\tgdd\anaconda3\lib\site-pa
ckages (from yellowbrick) (0.11.0)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\tgdd\anaconda3\lib\site-
packages (from matplotlib!=3.0.0,>=2.0.2->yellowbrick) (1.2.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\tgdd\anaconda3\lib\site
-packages (from matplotlib!=3.0.0,>=2.0.2->yellowbrick) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\tgdd\anaconda3\lib\site
-packages (from matplotlib!=3.0.0,>=2.0.2->yellowbrick) (1.4.4)
Requirement already satisfied: packaging>=20.0 in c:\users\tgdd\anaconda3\lib\site-p
ackages (from matplotlib!=3.0.0,>=2.0.2->yellowbrick) (24.1)
Requirement already satisfied: pillow>=8 in c:\users\tgdd\anaconda3\lib\site-package
s (from matplotlib!=3.0.0,>=2.0.2->yellowbrick) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\tgdd\anaconda3\lib\site-
packages (from matplotlib!=3.0.0,>=2.0.2->yellowbrick) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\tgdd\anaconda3\lib\s
ite-packages (from matplotlib!=3.0.0,>=2.0.2->yellowbrick) (2.9.0.post0)
Requirement already satisfied: joblib>=1.2.0 in c:\users\tgdd\anaconda3\lib\site-pac
kages (from scikit-learn>=1.0.0->yellowbrick) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in c:\users\tgdd\anaconda3\lib\s
ite-packages (from scikit-learn>=1.0.0->yellowbrick) (3.5.0)
Requirement already satisfied: six>=1.5 in c:\users\tgdd\anaconda3\lib\site-packages
(from python-dateutil>=2.7->matplotlib!=3.0.0,>=2.0.2->yellowbrick) (1.16.0)
```

2. Importing Libraries

Here we import the essential Python libraries for data manipulation (**pandas**), plotting (**matplotlib.pyplot**), K-Means clustering (**sklearn.cluster.KMeans**), data scaling (**sklearn.preprocessing.StandardScaler**), and the Elbow Method visualization (**yellowbrick.cluster.KElbowVisualizer**).

In [58]:
```python
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from yellowbrick.cluster import KElbowVisualizer
import warnings
warnings.filterwarnings('ignore')    # ignor warning message
```

3. Data Loading and Initial Inspection

We start by defining our raw customer data and loading it into a **pandas** DataFrame. This allows us to view the initial structure and data types, ensuring it's ready for processing. The `df.info()` command provides a quick summary, confirming non-null counts and data types.

In [8]:
```
data = {
    'CustomerID':[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
    'Segment':['Segment1', 'Segment2', 'Segment1', 'Segment3', 'Segment2', 'Segment
               'Segment2', 'Segment1', 'Segment3', 'Segment2', 'Segment1', 'Segment
               'Segment1', 'Segment3', 'Segment2', 'Segment1', 'Segment3', 'Segment
    'Recency':[10, 5, 15, 3, 8, 20, 2, 6, 18, 1, 4, 12, 2, 7, 16, 1, 3, 9, 2, 5],
    'Frequency':[25, 40, 10, 60, 35, 15, 70, 45, 20, 80, 55, 30, 75, 50, 22, 85, 65
    'MonetaryValue': [500, 1000, 250, 1500, 800, 300, 1800, 900, 400, 2000, 1100, 6
                      350, 2200, 1300, 550, 2400, 1200]
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df.head())
print("\nDataFrame Info:")
df.info()
```

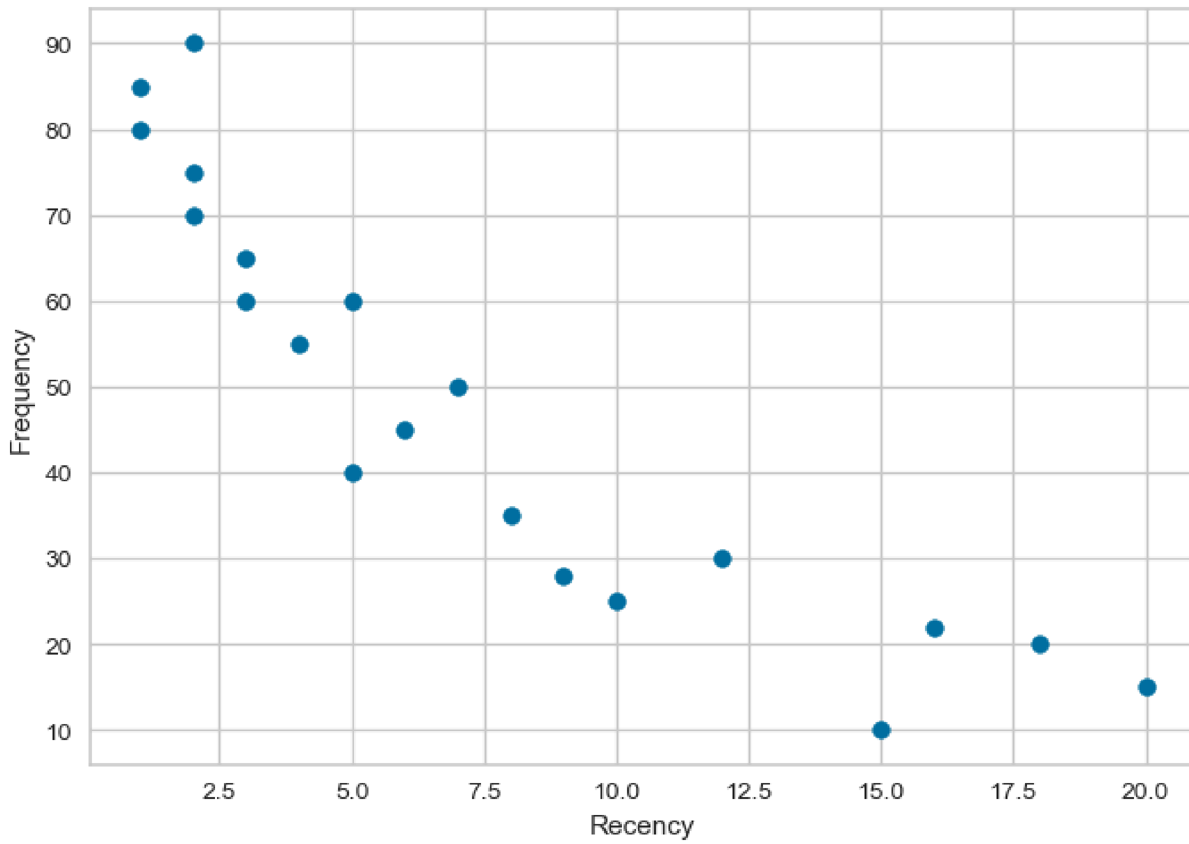
```
Original DataFrame:
   CustomerID   Segment  Recency  Frequency  MonetaryValue
0           1  Segment1       10         25            500
1           2  Segment2        5         40           1000
2           3  Segment1       15         10            250
3           4  Segment3        3         60           1500
4           5  Segment2        8         35            800

DataFrame Info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20 entries, 0 to 19
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype 
---  ------         --------------  ----- 
 0   CustomerID     20 non-null     int64 
 1   Segment        20 non-null     object
 2   Recency        20 non-null     int64 
 3   Frequency      20 non-null     int64 
 4   MonetaryValue  20 non-null     int64 
dtypes: int64(4), object(1)
memory usage: 932.0+ bytes
```

In [9]:
```
# Checking if there is any missing values in the dataframe
df.isnull().sum()
```

Out[9]:
```
CustomerID       0
Segment          0
Recency          0
Frequency        0
MonetaryValue    0
dtype: int64
```

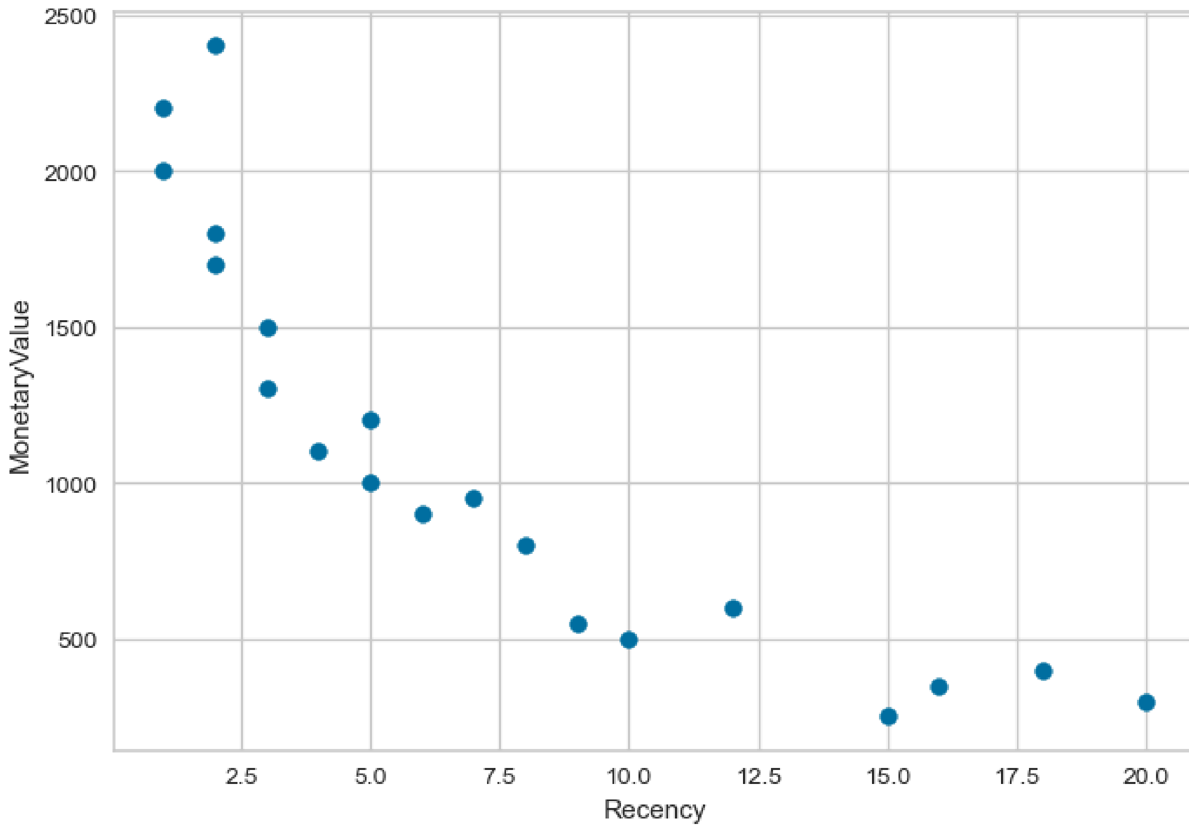
In [13]:
```python
# To observe relationships between Recency and Frequency.
plt.scatter(df["Recency"], df["Frequency"])
plt.xlabel("Recency")
plt.ylabel("Frequency")
```

Out[13]:
```
Text(0, 0.5, 'Frequency')
```



In [15]:
```python
# To observe relationships between Recency and MonetaryValue.
plt.scatter(df["Recency"], df["MonetaryValue"])
plt.xlabel("Recency")
plt.ylabel("MonetaryValue")
```

Out[15]:
```
Text(0, 0.5, 'MonetaryValue')
```

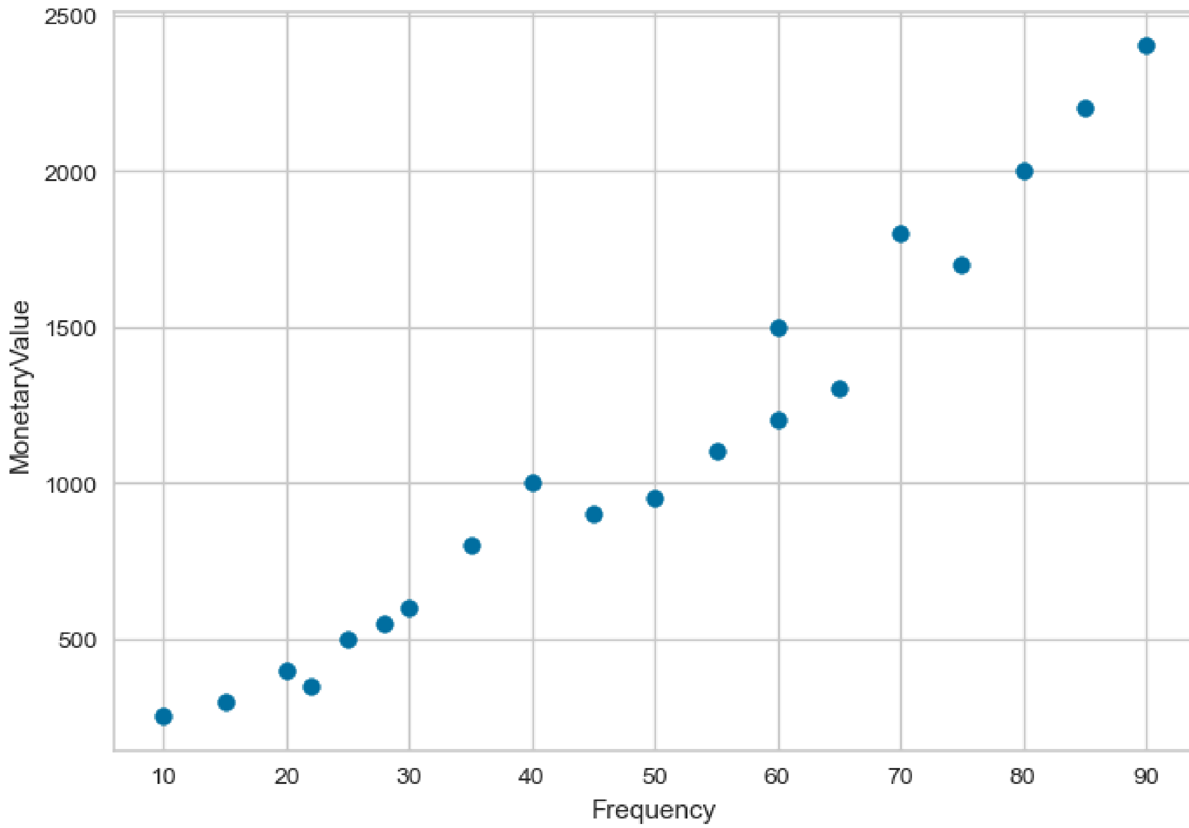


In [17]:

```
# To observe relationships between Frequency and MonetaryValue.
plt.scatter(df["Frequency"], df["MonetaryValue"])
plt.xlabel("Frequency")
plt.ylabel("MonetaryValue")
```

Out[17]:

```
Text(0, 0.5, 'MonetaryValue')
```



4. Feature Selection for Clustering

KMeans is a distance-based algorithm, requiring numerical features. We explicitly select **'Recency'**, **'Frequency'**, and **'MonetaryValue'** (RFM) as our clustering features, excluding **'CustomerID'** (an identifier) and **'Segment'** (a pre-existing categorical label that we will compare our clusters against later).

In [24]:
```
# Select only the numerical features for clustering
X = df[['Recency', 'Frequency', 'MonetaryValue']]
print("\nFeatures for Clustering (X):")
print(X.head())
```

```
Features for Clustering (X):
   Recency  Frequency  MonetaryValue
0       10         25            500
1        5         40           1000
2       15         10            250
3        3         60           1500
4        8         35            800
```

5. Data Standardization: Ensuring Fair Feature Contribution

To prevent features with larger numerical ranges (like **MonetaryValue**) from disproportionately influencing the clustering process, we standardize the data. **StandardScaler** transforms each feature to have a mean of 0 and a standard deviation of 1. This ensures all features contribute equally to the distance calculations within KMeans.

In [26]:
```
# Scale the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
X_scaled_df = pd.DataFrame(X_scaled, columns=X.columns)
print("\nScaled Features (X_scaled_df):")
print(X_scaled_df.head())
```

```
Scaled Features (X_scaled_df):
    Recency  Frequency  MonetaryValue
0  0.441579  -0.959667      -0.918184
1 -0.424262  -0.333797      -0.140062
2  1.307421  -1.585537      -1.307245
3 -0.770599   0.500696       0.638060
4  0.095243  -0.542421      -0.451311
```

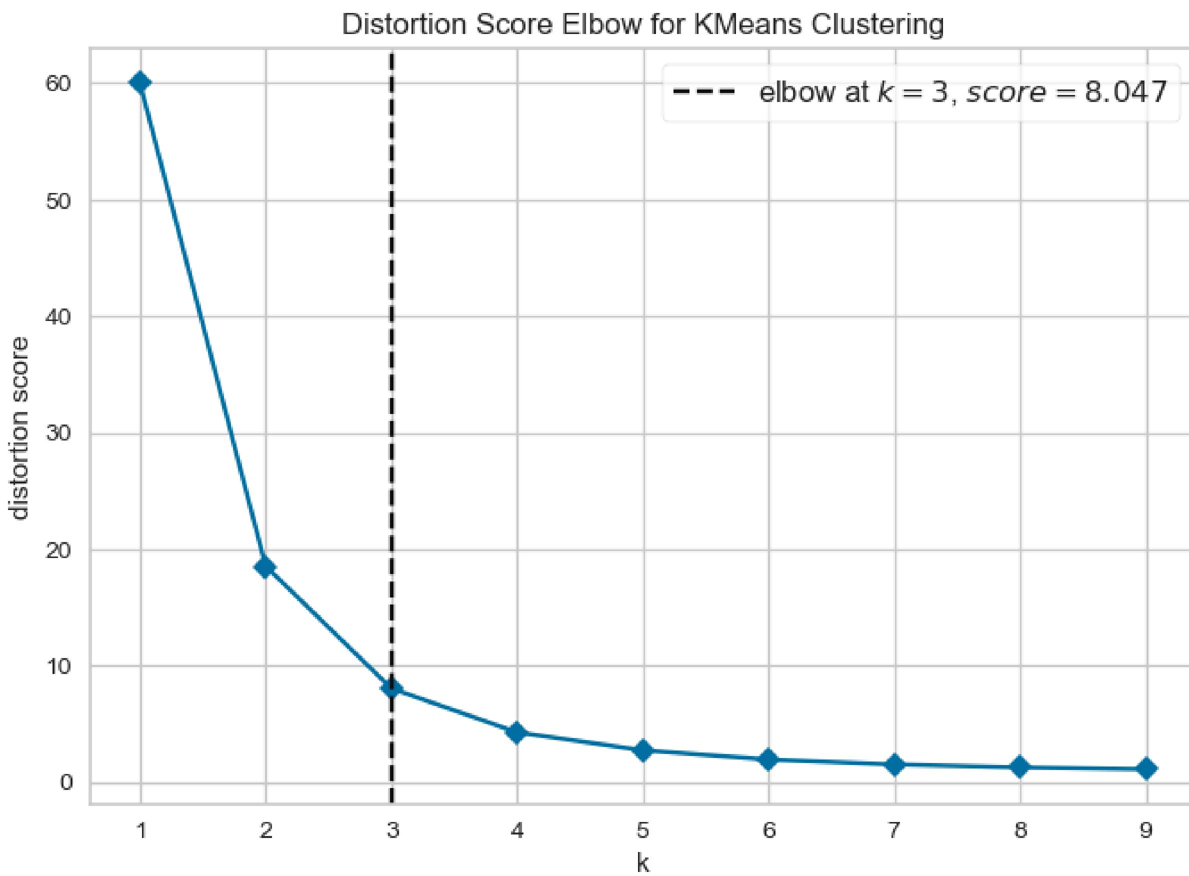
6. Determining the Optimal Number of Clusters: The Elbow Method

The **Elbow Method** is a heuristic technique to find the optimal **k** for KMeans. It plots the **inertia** (Within-Cluster Sum of Squares — WCSS) against the number of clusters. The "elbow" point, where the rate of decrease in inertia sharply changes, is often considered the optimal **k**. We use **yellowbrick.cluster.KElbowVisualizer** for a concise plot, and also demonstrate the manual calculation for conceptual understanding.

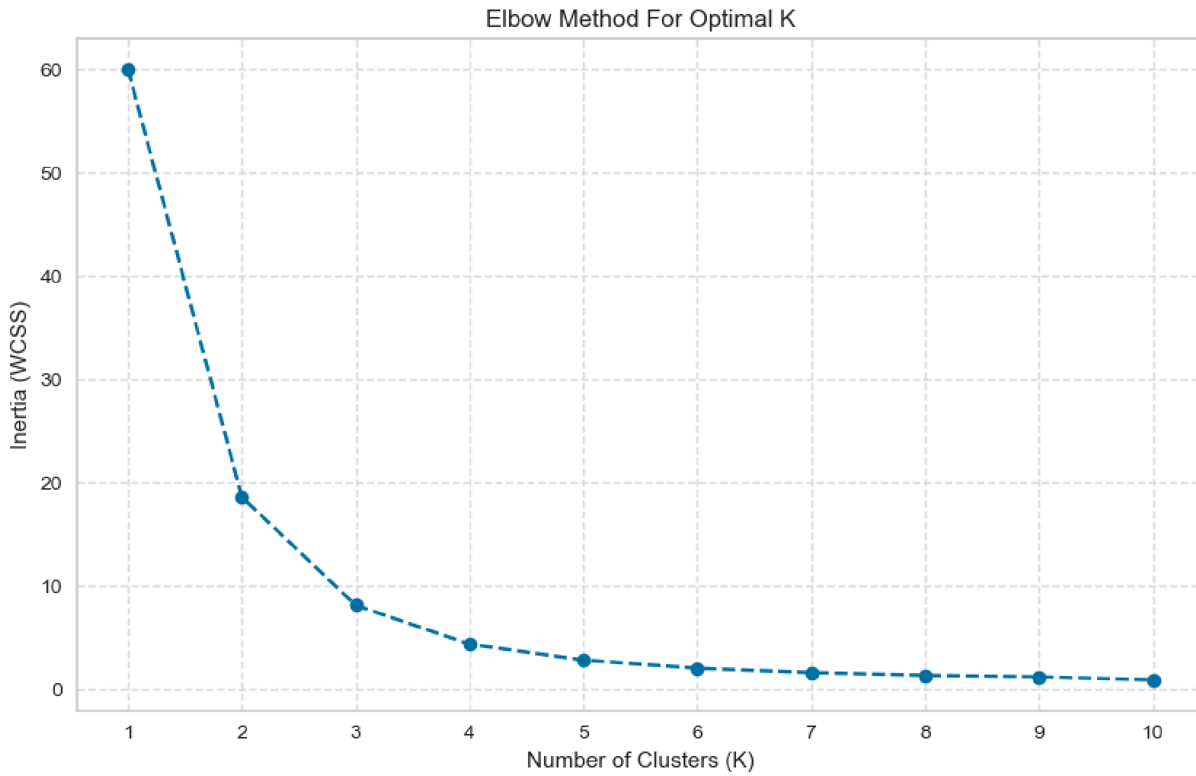
In [68]:
```
# Use the Yellowbrick KElbowVisualizer for a clear visualization
print("\n--- Elbow Method to find Optimal K ---")
model = KMeans(random_state=42, n_init=10) # n_init=10 to avoid warning
visualizer = KElbowVisualizer(model, k=(1,10), metric='distortion', timings=False)

visualizer.fit(X_scaled)        # Fit the data to the visualizer
visualizer.show()               # Finalize and render the figure

# Alternatively, manual calculation (useful for understanding)
wcss = [] # Within-Cluster Sum of Squares (Inertia)
for i in range(1, 11): # Test k from 1 to 10
    kmeans = KMeans(n_clusters=i, random_state=42, n_init=10)
    kmeans.fit(X_scaled)
    wcss.append(kmeans.inertia_)
```

```python
# Run KneeLocator
kneedle = KneeLocator(k_range, inertia, curve='convex', direction='decreasing', int
print(f"Optimal k found by KneeLocator: {kneedle.elbow}")

plt.figure(figsize=(10, 6))
plt.plot(range(1, 11), wcss, marker='o', linestyle='--')
plt.title('Elbow Method For Optimal K')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Inertia (WCSS)')
plt.xticks(range(1, 11))
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```

```
--- Elbow Method to find Optimal K ---
```




```
Optimal k found by KneeLocator: 3
```



7. Final K-Means Clustering with Optimal K and Cluster Analysis

Based on the Elbow Method, **k=3** was identified as the optimal number of clusters. We now apply **KMeans** with this optimal **k**, assign the resulting clusters to a new **'Optimal_Cluster'** column, and perform a detailed analysis of each cluster's characteristics using the unscaled original data.

Cluster Characteristics: We examine the mean of **Recency**, **Frequency**, and **MonetaryValue** for each cluster. This is crucial for interpreting what defines each customer segment.

Comparison with Original Segments: A cross-tabulation using `pd.crosstab` helps us compare the new K-Means clusters with the original **'Segment'** labels. This serves as a powerful validation step.

The results reveal distinct customer groups:

- **Cluster 0:** “Lapsed/Low-Value Customers” — high Recency, low Frequency, low MonetaryValue.
- **Cluster 1:** “VIP/High-Value Customers” — very low Recency, very high Frequency, very high MonetaryValue.
- **Cluster 2:** “Regular/Mid-Value Customers” — medium Recency, medium Frequency, medium MonetaryValue.

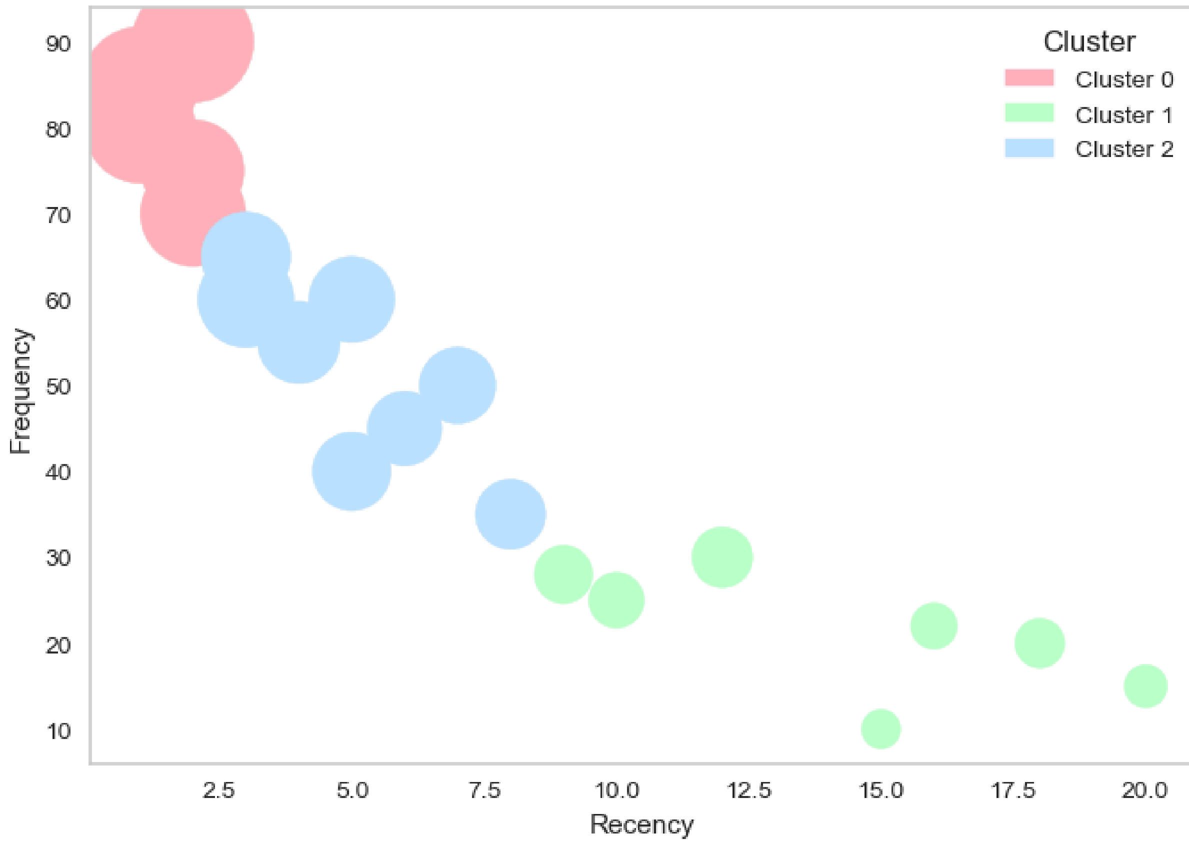
The strong alignment between **Optimal_Cluster** and **Segment** (e.g., Cluster 0 aligns with Segment1, Cluster 1 with Segment3, and Cluster 2 mostly with Segment2) indicates that KMeans has effectively uncovered natural groupings in the data.

In [54]:
```
# Creating KMeans Model for the optimum no of clustering
kmeans_model = KMeans(n_clusters = 3, random_state = 50)
kmeans_model.fit(X_scaled)

# Adding the cluster label to the dataframe
df["Cluster"] = kmeans_model.labels_ # Inserting the label of cluster into the data
df.head()
```

Out[54]:

	CustomerID	Segment	Recency	Frequency	MonetaryValue	Cluster
0	1	Segment1	10	25	500	1
1	2	Segment2	5	40	1000	2
2	3	Segment1	15	10	250	1
3	4	Segment3	3	60	1500	2
4	5	Segment2	8	35	800	2

In [62]:
```
# Plotting the clusters against the features
colors_map = {0: '#FFB3BA', # light pink
              1: '#BAFFC9', # light green
              2: '#BAE1FF'} # light blue
colors = df["Cluster"].map(colors_map)

plt.scatter(df["Recency"],
            df["Frequency"],
            s = df["MonetaryValue"], # size of the bubble/points
            c = colors)
legend_handles = [
    mpatches.Patch(color=color, label=f'Cluster {cluster}') # To show the color leg
    for cluster, color in colors_map.items()
]
plt.legend(handles=legend_handles, title='Cluster')
plt.xlabel('Recency')
plt.ylabel('Frequency')
plt.grid(False)
plt.show()
```



In [64]:
```
# Apply K-means with the optimal k (e.g., k=3 based on elbow)
optimal_k = 3
kmeans_optimal = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)
df['Optimal_Cluster'] = kmeans_optimal.fit_predict(X_scaled)

print(f"\nDataFrame with {optimal_k} Optimal Clusters:")
print(df.head())
print(f"\nCluster counts for optimal k={optimal_k}:")
print(df['Optimal_Cluster'].value_counts())

# Let's inspect the characteristics of each cluster (using unscaled data for interp
cluster_centers_scaled = kmeans_optimal.cluster_centers_
cluster_centers_unscaled = scaler.inverse_transform(cluster_centers_scaled)

cluster_summary_df = pd.DataFrame(cluster_centers_unscaled, columns=X.columns)
cluster_summary_df['Cluster'] = range(optimal_k)
cluster_summary_df = cluster_summary_df.set_index('Cluster')

print(f"\nCharacteristics of each cluster (Optimal k={optimal_k}):")
print(cluster_summary_df)

# You can also group by the cluster and get descriptive statistics
cluster_descriptive_stats = df.groupby('Optimal_Cluster')[['Recency', 'Frequency',
print("\nDescriptive statistics for each cluster:")
print(cluster_descriptive_stats)

# Compare with original segments (if applicable)
print("\nComparison of Optimal Clusters with Original Segments:")
```

```python
cluster_segment_crosstab = pd.crosstab(df['Optimal_Cluster'], df['Segment'])
print(cluster_segment_crosstab)
```

```
DataFrame with 3 Optimal Clusters:
   CustomerID   Segment  Recency  Frequency  MonetaryValue  Cluster  \
0           1  Segment1       10         25            500        1   
1           2  Segment2        5         40           1000        2   
2           3  Segment1       15         10            250        1   
3           4  Segment3        3         60           1500        2   
4           5  Segment2        8         35            800        2   

   Optimal_Cluster  
0                0  
1                2  
2                0  
3                2  
4                2  

Cluster counts for optimal k=3:
Optimal_Cluster
2    8
0    7
1    5
Name: count, dtype: int64

Characteristics of each cluster (Optimal k=3):
           Recency  Frequency  MonetaryValue
Cluster                                     
0        14.285714  21.428571     421.428571
1         1.600000  80.000000    2020.000000
2         5.125000  51.250000    1093.750000

Descriptive statistics for each cluster:
                   Recency  Frequency  MonetaryValue
Optimal_Cluster                                     
0                14.285714  21.428571     421.428571
1                 1.600000  80.000000    2020.000000
2                 5.125000  51.250000    1093.750000

Comparison of Optimal Clusters with Original Segments:
Segment          Segment1  Segment2  Segment3
Optimal_Cluster                              
0                       7         0         0
1                       0         0         5
2                       0         7         1
```

In [ ]: