Market Segmentation Analysis: Step 5 - Extracting Segments

Data-Driven Approaches to Consumer Clustering

November 8, 2025

Abstract

Data-driven market segmentation analysis is fundamentally exploratory. Consumer datasets typically lack clear structure, with preferences distributed across multidimensional space rather than forming distinct natural clusters. This comprehensive chapter explores how different extraction algorithms impose unique structural assumptions on segmented data, revealing that segmentation results depend equally on the underlying data characteristics and the chosen algorithmic approach. We examine distance-based methods (hierarchical clustering, k-means, self-organizing maps), model-based approaches (finite mixture models), and specialized algorithms with integrated variable selection. Through rigorous mathematical foundations, practical Python implementations, and detailed case studies using real consumer data, this work demonstrates that no single "best" algorithm exists—rather, each method offers distinct advantages for specific data characteristics and business objectives.

Contents

1 Introduction: The Power of Algorithmic Choice

The Central Challenge

Market segmentation analysis faces a fundamental paradox: consumer data rarely contains naturally distinct groups, yet businesses require clear, actionable market segments. This chapter addresses the critical question:
How do different algorithms shape the segments they extract, and how can we select methods aligned with our business objectives?
The answer lies in understanding that:

- Segmentation methods **impose structure** on unstructured data
- Algorithm selection is as important as data quality
- Multiple valid segmentation solutions often exist
- Systematic exploration across methods is essential

1.1 Why Algorithm Choice Matters: The Spiral Example

Consider Figure 1, which demonstrates how dramatically different algorithms can produce different results from identical data.

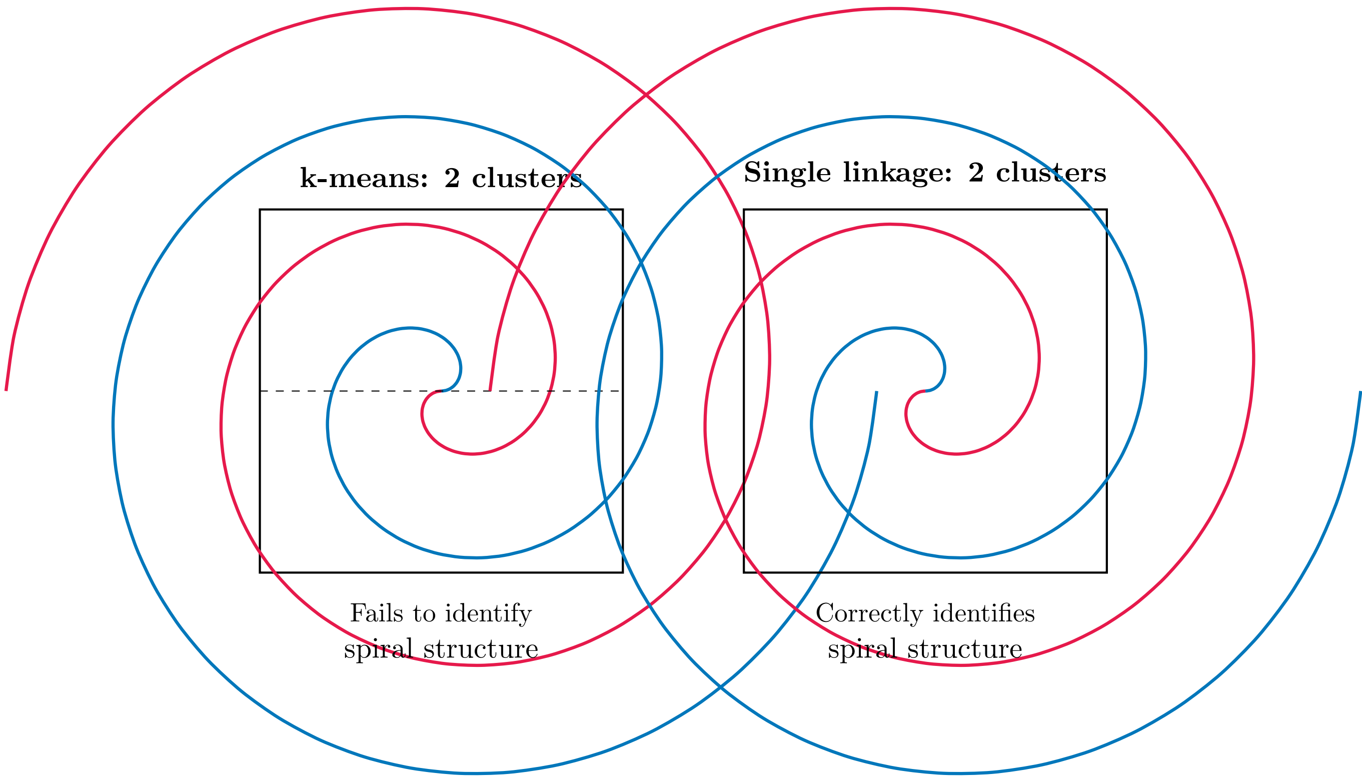
k-means: 2 clusters
Single linkage: 2 clusters
Fails to identify spiral structure
Correctly identifies spiral structure

Figure 1: Comparison of k-means and single linkage clustering on spiral data. k-means imposes compact, spherical cluster assumptions and fails to detect the spiral structure. Single linkage allows snake-shaped clusters and correctly identifies both spirals.

Critical Insight

The spiral example illustrates a fundamental truth:

- **k-means** assumes compact, equally-sized, spherical clusters
- **Single linkage** allows elongated, chain-like structures
- Neither algorithm is universally "better"—each has optimal use cases
- The data structure determines which assumptions are appropriate

For consumer data, which typically lacks well-defined structure, algorithmic tendencies become even more influential in shaping the final segmentation solution.

1.2 Guiding Principles for Algorithm Selection

Algorithm Selection Framework

Three key dimensions inform extraction algorithm selection:

1. Data Characteristics

- Sample size: n consumers, p variables
- Variable scale: nominal, ordinal, metric, or mixed
- Data structure: presence of outliers, niche segments, hierarchical relationships
- Special features: temporal data, spatial data, repeated measurements

2. Desired Segment Properties

- Within-segment similarity: what should consumers share?
- Between-segment differences: how should segments differ?
- Segment size constraints: minimum viable segment size
- Number of segments: predetermined vs. exploratory

3. Computational Constraints

- Available memory for distance matrix computation
- Processing time requirements
- Software capabilities and available implementations

Table 1: Matching Data and Segment Characteristics to Algorithm Families

Scenario	Recommended Approach	Rationale
Small datasets ($n < 500$)	Hierarchical clustering	Dendrogram aids interpretation; memory not limiting
Large datasets ($n > 5000$)	Partitioning methods (k-means, SOM)	Computational efficiency; scalability
Mixed variable types	Model-based (finite mixtures)	Accommodates different distributions
Binary data with noise	Biclustering, VSBD	Simultaneous variable selection
Niche segment detection	Bagged clustering	Bootstrap resampling reveals rare segments
Unknown segment count	Hierarchical + dendrogram	Visual inspection suggests optimal k
Clear segment count	Partitioning methods	Direct optimization for specified k

2 Grouping Consumers: Core Concepts

2.1 The Nature of Consumer Data

Unstructured Consumer Preferences

Consumer data presents unique challenges:
Characteristics of Real Consumer Datasets:

1. **Continuous distributions**: Preferences spread across entire feature space
2. **No clear boundaries**: Gradual transitions rather than distinct clusters
3. **High dimensionality**: Many product attributes or behavioral variables
4. **Heterogeneity**: Consumers truly differ in individual ways

Implications for Segmentation:

- Algorithms must *create* structure from unstructured data
- Multiple valid segmentation solutions can coexist
- Domain knowledge must guide algorithm selection
- Validation through stability analysis is essential

2.2 Distance Measures: Foundations of Similarity

All distance-based segmentation methods require a mathematical definition of similarity or dissimilarity between consumers.

Distance Measure Requirements

A function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ is a valid distance measure if it satisfies:

$$d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x}) \quad \text{(Symmetry)} \tag{1}$$
$$d(\mathbf{x}, \mathbf{y}) = 0 \Leftrightarrow \mathbf{x} = \mathbf{y} \quad \text{(Identity)} \tag{2}$$
$$d(\mathbf{x}, \mathbf{z}) \leq d(\mathbf{x}, \mathbf{y}) + d(\mathbf{y}, \mathbf{z}) \quad \text{(Triangle inequality)} \tag{3}$$

where $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^p$ are p-dimensional consumer profiles.

2.2.1 Common Distance Measures

Let $\mathbf{x} = (x_1, \ldots, x_p)$ and $\mathbf{y} = (y_1, \ldots, y_p)$ represent two consumer profiles.

Table 2: Primary Distance Measures for Market Segmentation

Distance Type	Formula	Use Case
Euclidean	$d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{j=1}^{p} (x_j - y_j)^2}$	Most common; metric data
Manhattan	$d(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{p} \lvert x_j - y_j \rvert$	Robust to outliers
Asymmetric Binary	$d(\mathbf{x}, \mathbf{y}) = \frac{\sum_{j:x_j=1 \vee y_j=1} \nVdash_{x_j \neq y_j}}{\sum_{j:x_j=1 \vee y_j=1} 1}$	Activity patterns

Choosing Distance Measures

Euclidean Distance:

- Penalizes large differences more than small ones (squared terms)
- Assumes all dimensions equally important
- Requires standardization if variables have different scales
- Default choice for continuous consumer data

Manhattan Distance:

- Treats all differences equally (absolute values)
- More robust to outliers than Euclidean
- Natural for grid-like preference structures
- Often similar results to Euclidean on consumer data

Asymmetric Binary Distance:

- Critical for activity-based segmentation

- Shared presence (both = 1) indicates similarity
- Shared absence (both = 0) ignored as uninformative
- Example: Tourists who both engage in horseback riding are similar; tourists who both don't ride are not necessarily similar

3 Distance-Based Methods

3.1 Overview of Distance-Based Approaches

Core Principle

Distance-based methods identify market segments by:

1. Defining a mathematical measure of similarity (distance) between consumers
2. Finding groups where **within-segment similarity** is maximized
3. Ensuring **between-segment dissimilarity** is maximized

These methods translate the conceptual goal of "finding similar consumers" into precise mathematical optimization problems.

3.2 Hierarchical Clustering Methods

Hierarchical clustering mimics human intuition: we naturally think about grouping objects by progressively merging similar items or progressively dividing collections into subgroups.

3.2.1 Agglomerative vs Divisive Strategies

Table 3: Comparison of Hierarchical Clustering Strategies

Aspect	**Agglomerative (Bottom-Up)**	**Divisive (Top-Down)**
Starting point	Each consumer = own segment (n singletons)	All consumers in one segment
Process	Merge closest pairs iteratively	Split segments recursively
Ending point	One segment containing all consumers	Each consumer = own segment
Computational cost	$O(n^2 \log n)$	Typically higher
Most common	Yes (standard implementations)	No (rarely used)

Why Agglomerative Dominates

Agglomerative hierarchical clustering is overwhelmingly preferred in practice because:

- Well-established, efficient algorithms exist (Lance & Williams framework)
- Deterministic: same data always produces same result
- Visualization through dendrograms aids interpretation
- Standard implementations available in all software packages

3.2.2 Linkage Methods: Defining Distance Between Groups

The critical decision in hierarchical clustering: *how do we measure distance between two groups of consumers?*

Linkage Method Definitions

Let X and Y be two sets of consumer profiles. Define distance $d(X, Y)$ between these sets:

Single Linkage (Nearest Neighbor):

$$d_{\text{single}}(X, Y) = \min_{\mathbf{x}\in X, \mathbf{y}\in Y} d(\mathbf{x}, \mathbf{y}) \tag{4}$$

Complete Linkage (Furthest Neighbor):

$$d_{\text{complete}}(X, Y) = \max_{\mathbf{x}\in X, \mathbf{y}\in Y} d(\mathbf{x}, \mathbf{y}) \tag{5}$$

Average Linkage:

$$d_{\text{average}}(X, Y) = \frac{1}{|X| \cdot |Y|} \sum_{\mathbf{x}\in X} \sum_{\mathbf{y}\in Y} d(\mathbf{x}, \mathbf{y}) \tag{6}$$

Ward's Method (Minimum Variance):

$$d_{\text{Ward}}(X, Y) = \frac{|X| \cdot |Y|}{|X| + |Y|} \|\bar{\mathbf{x}} - \bar{\mathbf{y}}\|^2 \tag{7}$$

where $\bar{\mathbf{x}}$ and $\bar{\mathbf{y}}$ are cluster centroids (mean vectors).

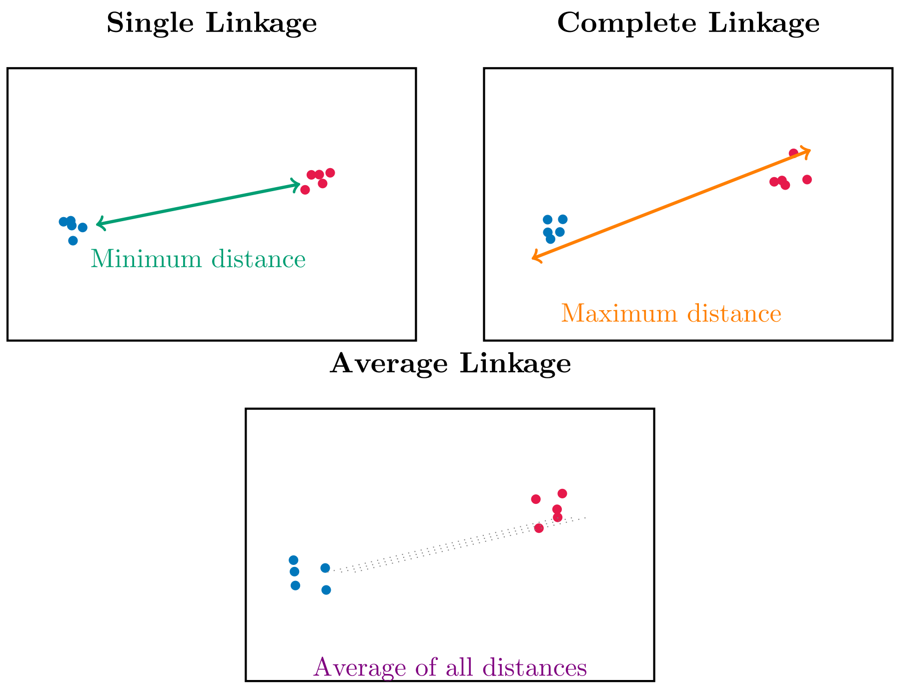

Figure 2: Visual comparison of linkage methods. Single linkage uses nearest neighbors, complete linkage uses furthest neighbors, and average linkage considers all pairwise distances.

Linkage Method Characteristics

Single Linkage:

- Detects elongated, chain-like, non-convex clusters (e.g., spirals)
- Sensitive to noise and outliers (”chaining” problem)
- Good for: Data with natural non-spherical structures

Complete Linkage:

- Produces compact, spherical clusters
- Less sensitive to outliers than single linkage
- Tends to create evenly-sized segments
- Good for: Well-separated consumer groups

Average Linkage:

- Compromise between single and complete
- Balances compactness with flexibility
- Often produces interpretable segments
- Good for: General-purpose consumer segmentation

Ward's Method:

- Minimizes within-cluster variance
- Strongly favors equal-sized, spherical clusters
- Most popular in market segmentation practice
- Uses squared Euclidean distance specifically
- Good for: Creating balanced segment portfolios

3.2.3 The Dendrogram: Visualizing Hierarchical Structure

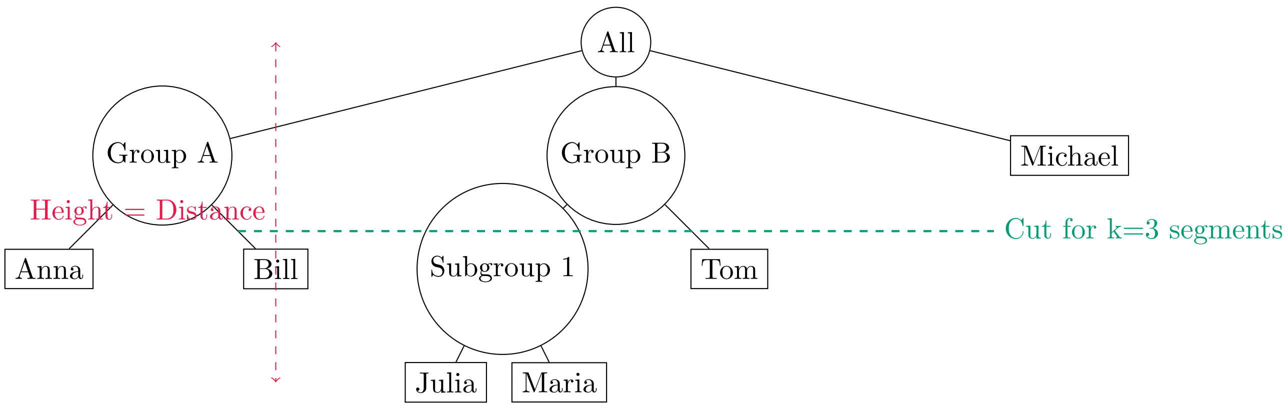

Figure 3: Schematic dendrogram showing hierarchical cluster structure. Height represents distance between merged clusters. Horizontal line shows where to cut tree for desired number of segments.

Dendrogram Interpretation Challenges

For real consumer data, dendrograms rarely provide clear guidance on optimal segment count because:

- Consumer preferences are typically continuously distributed
- No natural "elbow" or clear height jump exists
- Leaf ordering is arbitrary (can permute left/right branches)
- Different software may display different-looking but equivalent dendrograms

Recommendation: Use dendrograms for rough exploration, but rely on systematic stability analysis (Section 7.5) for final segment count determination.

3.2.4 Python Implementation: Hierarchical Clustering

```python
import numpy as np
import pandas as pd
```

```python
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from scipy.spatial.distance import pdist
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

# Example: Tourist vacation activities
data = pd.DataFrame({
    'beach': [100, 100, 60, 70, 80, 0, 50],
    'action': [0, 0, 40, 0, 0, 90, 20],
    'culture': [0, 0, 0, 30, 20, 10, 30]
}, index=['Anna', 'Bill', 'Frank', 'Julia', 'Maria', 'Michael', 'Tom'])

# Standardize if needed (for variables on different scales)
# scaler = StandardScaler()
# data_scaled = scaler.fit_transform(data)

# Calculate pairwise distances (Manhattan distance)
distances = pdist(data, metric='cityblock')  # 'cityblock' = Manhattan

# Perform hierarchical clustering (complete linkage)
linkage_matrix = linkage(distances, method='complete')

# Create dendrogram
plt.figure(figsize=(10, 6))
dendrogram(linkage_matrix, labels=data.index, leaf_font_size=12)
plt.xlabel('Tourist')
plt.ylabel('Distance')
plt.title('Complete Linkage Hierarchical Clustering')
plt.tight_layout()
plt.show()

# Extract k=3 segments
segments = fcluster(linkage_matrix, t=3, criterion='maxclust')
data['Segment'] = segments

# Display segment means
segment_means = data.groupby('Segment').mean()
print("\nSegment Characteristics:")
print(segment_means)
```

3.3 Partitioning Methods

Partitioning methods create a single division of consumers into k segments, optimizing directly for that specific number of segments rather than building a complete hierarchy.

3.3.1 The k-Means Algorithm

k-Means Clustering Algorithm

Input: Dataset $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, number of segments k
Output: Partition $S_1, \ldots, S_k$ and centroids $\mathbf{c}_1, \ldots, \mathbf{c}_k$

```
1: Initialize: Randomly select $k$ observations as initial centroids $\mathbf{c}_1^{(0)}, \ldots, \mathbf{c}_k^{(0)}$
2: Set iteration counter $t \leftarrow 0$
```

```
3: repeat
4:     $t \leftarrow t + 1$
5:     Assignment Step: Assign each $\mathbf{x}_i$ to nearest centroid
           $S_j^{(t)} = \{\mathbf{x}_i : \|\mathbf{x}_i - \mathbf{c}_j^{(t-1)}\|^2 \leq \|\mathbf{x}_i - \mathbf{c}_h^{(t-1)}\|^2, \ \forall h \in \{1, \dots, k\}\}$
6:     Update Step: Recompute centroids as segment means
           $\mathbf{c}_j^{(t)} = \frac{1}{|S_j^{(t)}|} \sum_{\mathbf{x}_i \in S_j^{(t)}} \mathbf{x}_i$
7: until Centroids no longer change: $\mathbf{c}_j^{(t)} = \mathbf{c}_j^{(t-1)}$ for all $j$
8: return Segments $S_1^{(t)}, \dots, S_k^{(t)}$ and centroids $\mathbf{c}_1^{(t)}, \dots, \mathbf{c}_k^{(t)}$
```

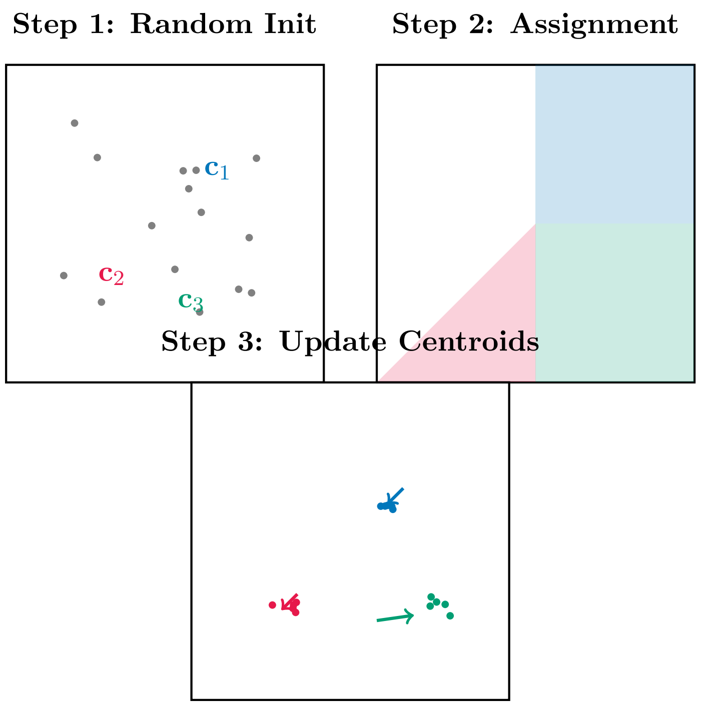

Figure 4: k-Means algorithm visualization: (1) Random initialization, (2) Assignment to nearest centroid, (3) Centroid update to segment means. Process repeats until convergence.

Key Properties of k-Means

Strengths:

- Computationally efficient: $O(nkp \cdot \text{iterations})$
- Scales to large datasets ($n > 10{,}000$)
- Simple to understand and implement

- Produces compact, spherical segments

Limitations:

- Requires pre-specification of k
- Sensitive to initialization (local optima)
- Assumes spherical clusters of similar size
- Only works with Euclidean distance
- Sensitive to outliers

Best Practices:

1. Run multiple times (10-50) with different random initializations
2. Select solution with minimum within-cluster sum of squares
3. Standardize variables if on different scales
4. Consider k-means++ initialization for better starting points

3.3.2 Python Implementation: k-Means Clustering

```python
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from sklearn.metrics import silhouette_score

# Generate or load data
np.random.seed(42)
n_consumers = 500

# Artificial mobile phone data
features = np.concatenate([
    np.random.normal(2, 0.5, 200),
    np.random.normal(5, 0.5, 200),
    np.random.normal(8, 0.5, 100)
])
prices = np.concatenate([
    np.random.normal(2, 0.5, 200),
    np.random.normal(5, 0.5, 200),
    np.random.normal(8, 0.5, 100)
])

data = pd.DataFrame({'features': features, 'price': prices})

# Standardize (important for k-means!)
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)
```

```

# Elbow method: try k=1 to k=10
inertias = []
silhouette_scores = []
K_range = range(1, 11)

for k in K_range:
    kmeans = KMeans(n_clusters=k, n_init=10, random_state=42)
    kmeans.fit(data_scaled)
    inertias.append(kmeans.inertia_)
    if k > 1:
        silhouette_scores.append(silhouette_score(data_scaled, kmeans.labels_))

# Plot elbow curve
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(K_range, inertias, 'bo-')
ax1.set_xlabel('Number of Segments (k)')
ax1.set_ylabel('Within-Cluster Sum of Squares')
ax1.set_title('Elbow Method')
ax1.grid(True)

ax2.plot(range(2, 11), silhouette_scores, 'ro-')
ax2.set_xlabel('Number of Segments (k)')
ax2.set_ylabel('Silhouette Score')
ax2.set_title('Silhouette Analysis')
ax2.grid(True)

plt.tight_layout()
plt.show()

# Fit final model with k=3
kmeans_final = KMeans(n_clusters=3, n_init=50, random_state=42)
data['Segment'] = kmeans_final.fit_predict(data_scaled)

# Visualize segments
plt.figure(figsize=(8, 6))
for segment in range(3):
    segment_data = data[data['Segment'] == segment]
    plt.scatter(segment_data['features'], segment_data['price'],
                label=f'Segment {segment+1}', alpha=0.6, s=50)

# Plot centroids (transform back to original scale)
centroids_original = scaler.inverse_transform(kmeans_final.cluster_centers_)
plt.scatter(centroids_original[:, 0], centroids_original[:, 1],
            c='black', marker='*', s=300, edgecolors='white', linewidth=2,
            label='Centroids')

plt.xlabel('Features')
plt.ylabel('Price')
plt.title('k-Means Segmentation (k=3)')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Segment profiles
```

```python
print("\nSegment Profiles:")
print(data.groupby('Segment')[['features', 'price']].mean())
print("\nSegment Sizes:")
print(data['Segment'].value_counts().sort_index())
```

3.4 Self-Organizing Maps (SOM)

Self-Organizing Maps represent a fundamentally different approach to segmentation, borrowing from neural network methodology to create topology-preserving mappings of high-dimensional consumer data.

The SOM Principle

Self-Organizing Maps (Kohonen networks) simultaneously achieve two objectives:

1. **Dimensionality reduction**: Map high-dimensional consumer data onto a 2D grid
2. **Clustering**: Group similar consumers while preserving topological relationships

The key innovation: **neighboring segments on the map are more similar than distant segments**, providing intuitive visualization of market landscape.

3.4.1 SOM Architecture and Training

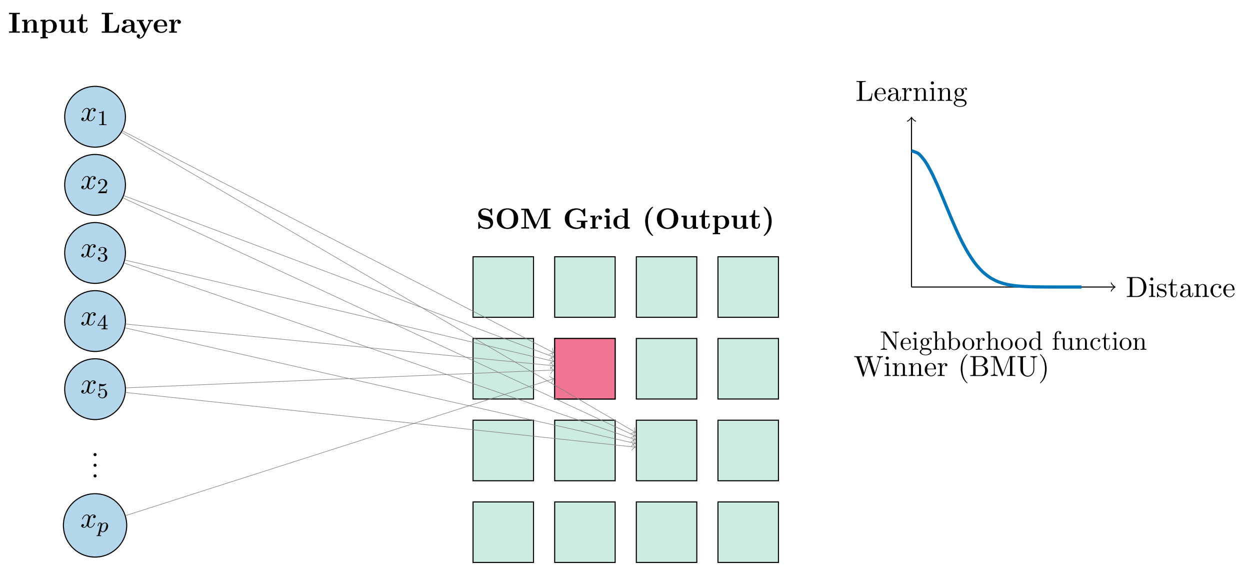

Figure 5: Self-Organizing Map architecture showing input layer (consumer features), 2D output grid, winner node (BMU - Best Matching Unit), and neighborhood function for competitive learning.

SOM Training Algorithm

Input: Dataset $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, grid dimensions $m \times n$
Output: Trained weight vectors $\mathbf{w}_{ij}$ for each grid node (i, j)

```
 1: Initialize: Randomly initialize weight vectors $\mathbf{w}_{ij}^{(0)} \in \mathbb{R}^p$ for all grid nodes
 2: Set learning rate $\alpha(0)$ and neighborhood radius $\sigma(0)$
 3: for epoch $t = 1$ to $T_{\max}$ do
 4:     for each consumer $\mathbf{x}_k \in X$ do
 5:         Competition: Find Best Matching Unit (BMU)
                $\text{BMU} = \arg\min_{i,j} \|\mathbf{x}_k - \mathbf{w}_{ij}^{(t)}\|$
 6:         Cooperation: Compute neighborhood function
                $h_{ij,\text{BMU}}(t) = \exp\left(-\frac{\|\mathbf{r}_{ij} - \mathbf{r}_{\text{BMU}}\|^2}{2\sigma(t)^2}\right)$
            where $\mathbf{r}_{ij}$ is grid position of node $(i, j)$
 7:         Adaptation: Update weights
                $\mathbf{w}_{ij}^{(t+1)} = \mathbf{w}_{ij}^{(t)} + \alpha(t) \cdot h_{ij,\text{BMU}}(t) \cdot (\mathbf{x}_k - \mathbf{w}_{ij}^{(t)})$
 8:     end for
 9:     Decrease learning rate: $\alpha(t + 1) = \alpha(t) \cdot \text{decay factor}$
10:     Decrease neighborhood: $\sigma(t + 1) = \sigma(t) \cdot \text{decay factor}$
11: end for
12: return Weight vectors $\mathbf{w}_{ij}$
```

SOM Advantages for Market Segmentation

Unique Benefits:

- **Visualization**: 2D map shows relationships between segments
- **Topology preservation**: Similar segments are neighbors on grid
- **Outlier detection**: Isolated nodes reveal niche segments
- **No predetermined** k: Segments emerge naturally from grid structure
- **Hierarchical view**: Can perform secondary clustering on SOM nodes

Practical Applications:

1. Initial exploratory analysis of consumer landscape
2. Identifying transition zones between major segments
3. Detecting emerging micro-segments
4. Creating intuitive management presentations

3.4.2 Python Implementation: Self-Organizing Maps

```python
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
```

```python
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

# Note: Need to install minisom library
# pip install minisom
from minisom import MiniSom

# Generate sample consumer data
np.random.seed(42)
n_consumers = 500

# Three natural segments
seg1 = np.random.normal([2, 8], [0.5, 0.5], (200, 2))
seg2 = np.random.normal([8, 2], [0.5, 0.5], (200, 2))
seg3 = np.random.normal([5, 5], [0.7, 0.7], (100, 2))

data = np.vstack([seg1, seg2, seg3])
data_df = pd.DataFrame(data, columns=['feature_preference',
↪  'price_sensitivity'])

# Standardize data
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)

# Initialize and train SOM
som_shape = (10, 10)  # 10x10 grid
som = MiniSom(som_shape[0], som_shape[1], data_scaled.shape[1],
              sigma=1.5,  # Initial neighborhood radius
              learning_rate=0.5,
              neighborhood_function='gaussian')

# Initialize weights
som.random_weights_init(data_scaled)

# Train SOM
print("Training SOM...")
som.train_random(data_scaled, num_iteration=1000)
print("Training complete!")

# Visualize SOM
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# Distance map (U-matrix)
from matplotlib.collections import PatchCollection

distance_map = som.distance_map()
ax1.imshow(distance_map.T, cmap='bone_r', origin='lower')
ax1.set_title('U-Matrix (Unified Distance Matrix)', fontsize=14)
ax1.set_xlabel('SOM Grid X')
ax1.set_ylabel('SOM Grid Y')

# Add colorbar
cbar1 = plt.colorbar(ax1.images[0], ax=ax1)
cbar1.set_label('Average Distance to Neighbors')

# Hit map (frequency)
```

```python
frequencies = np.zeros(som_shape)
for x in data_scaled:
    winner = som.winner(x)
    frequencies[winner] += 1

im2 = ax2.imshow(frequencies.T, cmap='YlOrRd', origin='lower')
ax2.set_title('Hit Map (Consumer Distribution)', fontsize=14)
ax2.set_xlabel('SOM Grid X')
ax2.set_ylabel('SOM Grid Y')

# Annotate hit counts
for i in range(som_shape[0]):
    for j in range(som_shape[1]):
        if frequencies[i, j] > 0:
            ax2.text(i, j, int(frequencies[i, j]),
                    ha='center', va='center', fontsize=8)

cbar2 = plt.colorbar(im2, ax=ax2)
cbar2.set_label('Number of Consumers')

plt.tight_layout()
plt.show()

# Assign consumers to SOM nodes
som_clusters = np.array([som.winner(x) for x in data_scaled])
data_df['som_x'] = som_clusters[:, 0]
data_df['som_y'] = som_clusters[:, 1]

# Perform secondary k-means clustering on SOM nodes
from sklearn.cluster import KMeans

# Get unique SOM node positions
unique_nodes = np.unique(som_clusters, axis=0)
node_weights = np.array([som.get_weights()[i, j]
                         for i, j in unique_nodes])

# Cluster SOM nodes into k=3 macro-segments
kmeans_som = KMeans(n_clusters=3, random_state=42)
node_labels = kmeans_som.fit_predict(node_weights)

# Map back to consumers
node_to_label = {tuple(node): label for node, label
                 in zip(unique_nodes, node_labels)}
data_df['segment'] = [node_to_label[tuple(pos)]
                      for pos in som_clusters]

# Visualize final segmentation
fig, ax = plt.subplots(figsize=(10, 8))

colors = ['blue', 'red', 'green']
for segment in range(3):
    segment_data = data_df[data_df['segment'] == segment]
    ax.scatter(segment_data['feature_preference'],
               segment_data['price_sensitivity'],
               c=colors[segment], label=f'Segment {segment+1}',
               alpha=0.6, s=50)
```

```python

# Plot segment centroids
centroids_original = scaler.inverse_transform(kmeans_som.cluster_centers_)
ax.scatter(centroids_original[:, 0], centroids_original[:, 1],
           c='black', marker='*', s=500, edgecolors='white',
           linewidth=2, label='Centroids', zorder=5)

ax.set_xlabel('Feature Preference')
ax.set_ylabel('Price Sensitivity')
ax.set_title('SOM-based Market Segmentation')
ax.legend()
ax.grid(True, alpha=0.3)
plt.show()

# Segment profiles
print("\nSegment Profiles:")
print(data_df.groupby('segment')[['feature_preference',
                                   'price_sensitivity']].mean())
print("\nSegment Sizes:")
print(data_df['segment'].value_counts().sort_index())
```

4 Model-Based Methods

4.1 Finite Mixture Models: Probabilistic Segmentation

Model-based segmentation represents a paradigm shift from distance-based approaches. Instead of measuring similarity through distances, these methods assume consumers are drawn from a mixture of probability distributions.

4.2 The EM Algorithm

Expectation-Maximization (EM) Algorithm

Purpose: Maximum likelihood estimation for finite mixture models

```
1: Initialize: Set initial parameter values $\mathbf{\Theta}^{(0)}$
2: Set iteration counter $t \leftarrow 0$
3: repeat
4:     $t \leftarrow t + 1$
5:     E-Step (Expectation): Compute posterior probabilities
           $$\tau_{ij}^{(t)} = P(z_i = j|\mathbf{y}_i, \mathbf{\Theta}^{(t-1)})$$
       for all consumers $i = 1, \ldots, n$ and segments $j = 1, \ldots, k$
6:     M-Step (Maximization): Update parameters
7:         Update segment sizes:
           $$\pi_j^{(t)} = \frac{1}{n}\sum_{i=1}^{n} \tau_{ij}^{(t)}$$
```

```
8:          Update segment-specific parameters $\boldsymbol{\theta}_j^{(t)}$ by maximizing:
```

$$\boldsymbol{\theta}_j^{(t)} = \arg\max_{\boldsymbol{\theta}_j} \sum_{i=1}^{n} \tau_{ij}^{(t)} \log f_j(\mathbf{y}_i|\boldsymbol{\theta}_j)$$

```
9:  until Convergence: $|\ell(\boldsymbol{\Theta}^{(t)}) - \ell(\boldsymbol{\Theta}^{(t-1)})| < \epsilon$
10: return Parameters $\boldsymbol{\Theta}^{(t)}$
```

where $\ell(\boldsymbol{\Theta})$ is the log-likelihood function.

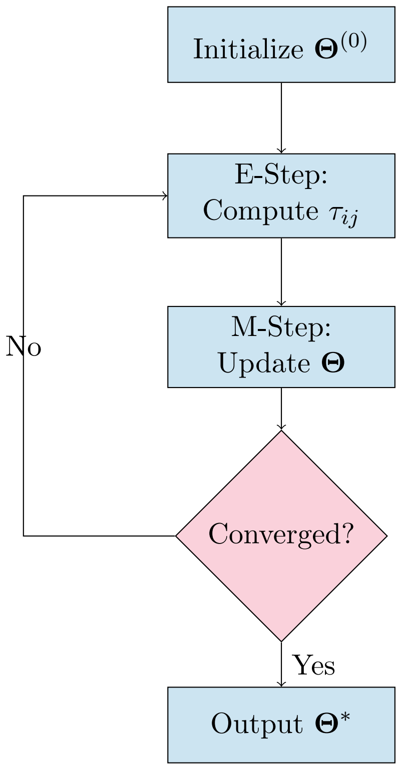
Initialize $\boldsymbol{\Theta}^{(0)}$
E-Step: Compute τ_{ij}
M-Step: Update $\boldsymbol{\Theta}$
Converged?
No
Yes
Output $\boldsymbol{\Theta}^{*}$

Figure 6: Flowchart of the Expectation-Maximization (EM) algorithm for estimating finite mixture model parameters. The algorithm iterates between computing membership probabilities (E-step) and updating parameters (M-step) until convergence.

4.2.1 Model Selection Criteria

Information Criteria for Model Selection

When number of segments k is unknown, compare models using:
Akaike Information Criterion (AIC):

$$\text{AIC} = -2\ell(\boldsymbol{\Theta}) + 2d \tag{10}$$

Bayesian Information Criterion (BIC):

$$\text{BIC} = -2\ell(\boldsymbol{\Theta}) + d\log(n) \tag{11}$$

Integrated Completed Likelihood (ICL):

$$\text{ICL} = \text{BIC} - 2\sum_{i=1}^{n}\sum_{j=1}^{k}\tau_{ij}\log(\tau_{ij}) \tag{12}$$

where:

- $\ell(\boldsymbol{\Theta})$: Maximized log-likelihood
- d: Number of free parameters
- n: Sample size
- τ_{ij}: Posterior probability (from Eq. **??**)

Decision rule: Select model minimizing criterion value
Comparison:

- BIC penalizes complexity more heavily than AIC ($\log(n) > 2$ for $n > 7$)
- ICL additionally penalizes unclear segment assignments (entropy term)
- No single criterion universally superior—compare multiple

4.2.2 Python Implementation: Gaussian Mixture Models

```python
import numpy as np
import pandas as pd
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from scipy import stats

# Generate sample data
np.random.seed(42)

# Three segments with different means and covariances
seg1 = np.random.multivariate_normal([2, 2], [[1, 0.5], [0.5, 1]], 200)
seg2 = np.random.multivariate_normal([8, 8], [[1, -0.3], [-0.3, 1]], 200)
seg3 = np.random.multivariate_normal([5, 8], [[2, 0], [0, 0.5]], 100)

data = np.vstack([seg1, seg2, seg3])
data_df = pd.DataFrame(data, columns=['features', 'price'])

# Standardize
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)

# Model selection: try k=1 to k=8
k_range = range(1, 9)
bic_scores = []
aic_scores = []

```

```
for k in k_range:
    gmm = GaussianMixture(n_components=k, covariance_type='full',
                          n_init=10, random_state=42)
    gmm.fit(data_scaled)
    bic_scores.append(gmm.bic(data_scaled))
    aic_scores.append(gmm.aic(data_scaled))

# Plot information criteria
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(k_range, bic_scores, 'bo-', linewidth=2, markersize=8)
ax1.set_xlabel('Number of Segments (k)')
ax1.set_ylabel('BIC')
ax1.set_title('Bayesian Information Criterion')
ax1.grid(True, alpha=0.3)
ax1.axvline(x=np.argmin(bic_scores)+1, color='red',
            linestyle='--', label='Optimal k')
ax1.legend()

ax2.plot(k_range, aic_scores, 'ro-', linewidth=2, markersize=8)
ax2.set_xlabel('Number of Segments (k)')
ax2.set_ylabel('AIC')
ax2.set_title('Akaike Information Criterion')
ax2.grid(True, alpha=0.3)
ax2.axvline(x=np.argmin(aic_scores)+1, color='blue',
            linestyle='--', label='Optimal k')
ax2.legend()

plt.tight_layout()
plt.show()

# Fit optimal model (k=3)
gmm_optimal = GaussianMixture(n_components=3, covariance_type='full',
                              n_init=50, random_state=42)
data_df['segment'] = gmm_optimal.fit_predict(data_scaled)
data_df['max_probability'] = np.max(gmm_optimal.predict_proba(data_scaled),
↪   axis=1)

# Visualize with uncertainty
fig, ax = plt.subplots(figsize=(10, 8))

colors = ['blue', 'red', 'green']
for segment in range(3):
    segment_data = data_df[data_df['segment'] == segment]

    # Size reflects uncertainty (larger = more uncertain)
    sizes = 100 * (1 - segment_data['max_probability'])

    ax.scatter(segment_data['features'], segment_data['price'],
               c=colors[segment], s=sizes, alpha=0.6,
               label=f'Segment {segment+1}', edgecolors='black', linewidth=0.5)

# Plot Gaussian ellipses
from matplotlib.patches import Ellipse

for i in range(3):
    mean = scaler.inverse_transform(gmm_optimal.means_[i:i+1])[0]
    covariance = gmm_optimal.covariances_[i]

    # Transform covariance back to original scale
    # This is approximate for visualization
    v, w = np.linalg.eigh(covariance)
    angle = np.arctan2(w[1, 0], w[0, 0]) * 180 / np.pi

    # 2 standard deviations
    for n_std in [1, 2]:
        ell = Ellipse(mean,
                      width=2*n_std*np.sqrt(v[0]),
                      height=2*n_std*np.sqrt(v[1]),
                      angle=angle,
                      edgecolor=colors[i],
                      facecolor='none',
                      linewidth=2,
                      linestyle='--')
        ax.add_patch(ell)

ax.set_xlabel('Features')
ax.set_ylabel('Price')
ax.set_title('Gaussian Mixture Model Segmentation\n(Point size indicates
↪   uncertainty)')
ax.legend()
ax.grid(True, alpha=0.3)
plt.show()

# Segment profiles
print("\nSegment Profiles (Original Scale):")
print(data_df.groupby('segment')[['features', 'price']].agg(['mean', 'std']))
print("\nSegment Sizes:")
print(data_df['segment'].value_counts().sort_index())
print("\nAverage Posterior Probability by Segment:")
print(data_df.groupby('segment')['max_probability'].mean())
```

5 Specialized Segmentation Methods

5.1 Methods with Integrated Variable Selection

Standard clustering algorithms use all provided segmentation variables. For high-dimensional consumer data, however, many variables may be irrelevant or noisy, obscuring true market segments.

The Variable Selection Challenge

Consumer datasets often contain:

- **Relevant variables** that define true market segments
- **Irrelevant variables** that add noise
- **Redundant variables** that duplicate information

Including irrelevant variables can:

1. Mask true segment structure
2. Reduce statistical power
3. Increase computational cost
4. Decrease interpretability

5.1.1 Variable Selection for Binary Data (VSBD)

VSBD Optimization Problem

For binary segmentation variables, VSBD simultaneously:

1. Selects subset of $V \subseteq \{1, \ldots, p\}$ relevant variables
2. Partitions consumers into k segments based only on V

Objective function: Maximize Ratkowsky-Lance index

$$RL(V,k) = \frac{1}{\sqrt{k}} \cdot \frac{1}{|V|} \sum_{j \in V} \frac{B_j}{T_j} \tag{13}$$

where:

- $B_j = \sum_{h=1}^{k} n_h (\bar{y}_{hj} - \bar{y}_j)^2$: Between-segment variation for variable j
- $T_j = \sum_{i=1}^{n} (y_{ij} - \bar{y}_j)^2$: Total variation for variable j
- n_h: Size of segment h

VSBD Advantages

Key Benefits:

- Identifies **niche segments** defined by specific activity combinations
- Removes noisy variables automatically
- Increases interpretability (fewer variables per segment)
- Particularly effective for binary activity data

Typical Applications:

- Vacation activity segmentation
- Product feature preferences
- Media consumption patterns
- Purchase behavior clustering

5.2 Bagged Clustering: Ensemble Approach

Bagged clustering combines bootstrap aggregation with hierarchical and partitioning methods to improve stability and niche segment detection.

Bagged Clustering Procedure

```
 1: Input: Dataset $X$, number of bootstrap samples $b$, base number of clusters $k_{\text{base}}$
 2: Step 1: Generate $b$ bootstrap samples (draw with replacement)
 3: for each bootstrap sample $i = 1$ to $b$ do
 4:     Run partitioning algorithm (e.g., k-means) with $k_{\text{base}}$ clusters
 5:     Store resulting $k_{\text{base}}$ centroids
 6: end for
 7: Step 2: Collect all $b \times k_{\text{base}}$ centroids
 8: Step 3: Discard original data; use centroids as new dataset
 9: Step 4: Apply hierarchical clustering to centroids
10: Step 5: Inspect dendrogram; select final $k$
11: Step 6: Assign original consumers to nearest final cluster center
12: return Final segmentation with $k$ segments
```

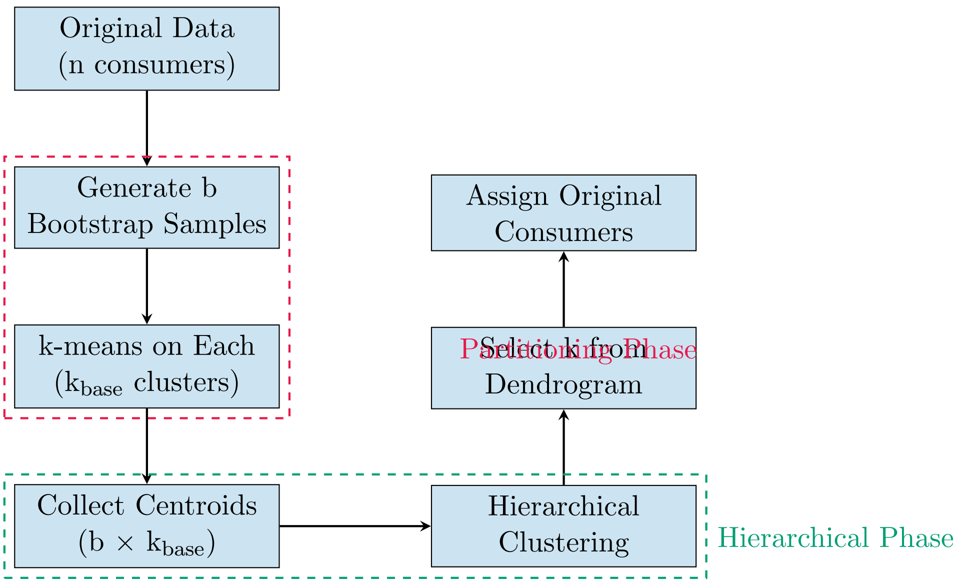
Original Data (n consumers)
Generate b Bootstrap Samples
k-means on Each (k_{base} clusters)
Collect Centroids (b × k_{base})
Hierarchical Clustering
Select k from Dendrogram
Assign Original Consumers
Partitioning Phase
Hierarchical Phase

Figure 7: Bagged clustering workflow combining bootstrap resampling, partitioning methods, and hierarchical clustering to improve niche segment detection.

5.2.1 Python Implementation: Bagged Clustering

```python
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

```

```python
# Generate sample data
np.random.seed(42)
n_consumers = 800

# Main segments + niche segment
main1 = np.random.normal([2, 2], [1, 1], (350, 2))
main2 = np.random.normal([8, 8], [1, 1], (350, 2))
niche = np.random.normal([5, 1], [0.3, 0.3], (100, 2))  # Small niche

data = np.vstack([main1, main2, niche])
true_labels = np.array([0]*350 + [1]*350 + [2]*100)

# Standardize
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)

# Bagged clustering parameters
n_bootstrap = 50
base_k = 10

# Store all centroids from bootstrap samples
all_centroids = []

print("Running bagged clustering...")
for b in range(n_bootstrap):
    # Bootstrap sample
    indices = np.random.choice(len(data_scaled), size=len(data_scaled),
                               replace=True)
    bootstrap_data = data_scaled[indices]

    # k-means with base_k clusters
    kmeans = KMeans(n_clusters=base_k, n_init=10, random_state=b)
    kmeans.fit(bootstrap_data)

    # Store centroids
    all_centroids.append(kmeans.cluster_centers_)

# Stack all centroids
all_centroids = np.vstack(all_centroids)
print(f"Collected {len(all_centroids)} centroids")

# Hierarchical clustering on centroids
linkage_matrix = linkage(all_centroids, method='average')

# Plot dendrogram
plt.figure(figsize=(12, 6))
dendrogram(linkage_matrix)
plt.xlabel('Centroid Index')
plt.ylabel('Distance')
plt.title('Bagged Clustering Dendrogram')
plt.axhline(y=2.5, color='r', linestyle='--', label='Cut line for k=3')
plt.legend()
plt.tight_layout()
plt.show()

# Extract k=3 segments from dendrogram
```

```python
final_k = 3
centroid_labels = fcluster(linkage_matrix, t=final_k, criterion='maxclust')

# For each centroid, compute its final segment
# Then assign original consumers to segments
from scipy.spatial.distance import cdist

# Get final segment centers (mean of centroids in each segment)
final_centers = np.array([all_centroids[centroid_labels == j].mean(axis=0)
                          for j in range(1, final_k+1)])

# Assign original consumers to nearest final center
distances = cdist(data_scaled, final_centers)
consumer_segments = np.argmin(distances, axis=1)

# Visualize results
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# Original data with true labels
scatter1 = ax1.scatter(data[:, 0], data[:, 1], c=true_labels,
                       cmap='viridis', alpha=0.6, s=50)
ax1.set_title('True Segments')
ax1.set_xlabel('Feature 1')
ax1.set_ylabel('Feature 2')
plt.colorbar(scatter1, ax=ax1)

# Bagged clustering results
scatter2 = ax2.scatter(data[:, 0], data[:, 1], c=consumer_segments,
                       cmap='viridis', alpha=0.6, s=50)
final_centers_orig = scaler.inverse_transform(final_centers)
ax2.scatter(final_centers_orig[:, 0], final_centers_orig[:, 1],
            c='red', marker='*', s=500, edgecolors='white', linewidth=2,
            label='Final Centers')
ax2.set_title('Bagged Clustering Results')
ax2.set_xlabel('Feature 1')
ax2.set_ylabel('Feature 2')
ax2.legend()
plt.colorbar(scatter2, ax=ax2)

plt.tight_layout()
plt.show()

# Segment sizes
print("\nSegment Sizes:")
unique, counts = np.unique(consumer_segments, return_counts=True)
for seg, count in zip(unique, counts):
    print(f"Segment {seg+1}: {count} consumers ({100*count/len(data):.1f}%)")
```

6 Determining the Optimal Number of Segments

6.1 The Elbow Method

Scree Plot Analysis

The **elbow method** plots within-cluster sum of squares (WCSS) against number of clusters k:

$$\text{WCSS}(k) = \sum_{j=1}^{k} \sum_{\mathbf{x}_i \in S_j} \|\mathbf{x}_i - \mathbf{c}_j\|^2 \tag{14}$$

Interpretation:

- WCSS always decreases with increasing k
- Look for ”elbow” where decrease slows dramatically
- Elbow indicates optimal balance between fit and complexity

Limitation: Clear elbows rare with real consumer data!

Elbow at k=3

Figure 8: Idealized scree plot showing clear elbow at k=3 segments. In practice, consumer data rarely produces such distinct elbows.

6.2 Stability Analysis: The Gold Standard

Why Stability Matters

Consumer datasets typically lack natural, well-separated clusters. Multiple segmentation solutions may be statistically equivalent.
Stability analysis answers:

- Are extracted segments **reproducible**?
- Which k produces most **stable** solutions?
- Are individual segments **robust** to sampling variation?

6.2.1 Global Stability Analysis

Bootstrap Stability Procedure

```
1: Generate $b$ pairs of bootstrap samples (2b total)
2: for each $k = 2, 3, \ldots, k_{\max}$ do
3:     for each pair of bootstrap samples do
4:         Cluster both samples into $k$ segments
5:         Compute adjusted Rand index between the two solutions
6:     end for
```

```
 7:     Store $b$ adjusted Rand indices for this $k$
 8: **end for**
 9: Create boxplots of adjusted Rand indices vs. $k$
10: **Interpretation:**
11:   High values ($\approx 1$): Reproducible structure
12:   Low values ($\approx 0$): Artificial construction
13: **return** Recommended $k$ with highest median stability
```

Adjusted Rand Index

Measures similarity between two partitions, correcting for chance:

$$\text{ARI} = \frac{\text{Index} - \text{Expected Index}}{\text{Max Index} - \text{Expected Index}} \tag{15}$$

Properties:

- ARI = 1: Perfect agreement
- ARI = 0: Agreement by chance
- ARI ¡ 0: Less agreement than expected by chance

6.2.2 Python Implementation: Stability Analysis

```python
import numpy as np
from sklearn.cluster import KMeans
from sklearn.metrics import adjusted_rand_score
import matplotlib.pyplot as plt

def bootstrap_stability_analysis(data, k_range, n_bootstrap_pairs=50):
    """
    Perform global stability analysis using bootstrap resampling.
    """
    n = len(data)
    ari_results = {k: [] for k in k_range}

    print("Running bootstrap stability analysis...")
    for pair_idx in range(n_bootstrap_pairs):
        if pair_idx % 10 == 0:
            print(f"  Bootstrap pair {pair_idx+1}/{n_bootstrap_pairs}")

        # Generate pair of bootstrap samples
        indices1 = np.random.choice(n, size=n, replace=True)
        indices2 = np.random.choice(n, size=n, replace=True)

        sample1 = data[indices1]
        sample2 = data[indices2]

        for k in k_range:
            # Cluster both samples
            kmeans1 = KMeans(n_clusters=k, n_init=10, random_state=42)
            kmeans2 = KMeans(n_clusters=k, n_init=10, random_state=42)

            labels1 = kmeans1.fit_predict(sample1)
            labels2 = kmeans2.fit_predict(sample2)

            # Compute adjusted Rand index
            ari = adjusted_rand_score(labels1, labels2)
            ari_results[k].append(ari)

    return ari_results

# Example usage
np.random.seed(42)

# Generate data with 3 natural segments
seg1 = np.random.normal([2, 2], [0.5, 0.5], (200, 2))
seg2 = np.random.normal([5, 5], [0.5, 0.5], (200, 2))
seg3 = np.random.normal([8, 2], [0.5, 0.5], (200, 2))
data = np.vstack([seg1, seg2, seg3])

# Run stability analysis
k_range = range(2, 9)
ari_results = bootstrap_stability_analysis(data, k_range, n_bootstrap_pairs=50)

# Create stability boxplot
fig, ax = plt.subplots(figsize=(10, 6))

positions = list(k_range)
ari_data = [ari_results[k] for k in k_range]

bp = ax.boxplot(ari_data, positions=positions, widths=0.6, patch_artist=True,
                boxprops=dict(facecolor='lightblue', edgecolor='blue'),
                medianprops=dict(color='red', linewidth=2),
                whiskerprops=dict(color='blue'),
                capprops=dict(color='blue'))

ax.set_xlabel('Number of Segments (k)', fontsize=12)
ax.set_ylabel('Adjusted Rand Index', fontsize=12)
ax.set_title('Global Stability Analysis', fontsize=14, fontweight='bold')
ax.set_ylim([0, 1.05])
ax.grid(True, alpha=0.3)
ax.axhline(y=0.8, color='green', linestyle='--', alpha=0.5,
           label='High Stability Threshold')

# Highlight optimal k
medians = [np.median(ari_results[k]) for k in k_range]
optimal_k = list(k_range)[np.argmax(medians)]
ax.axvline(x=optimal_k, color='red', linestyle='--', linewidth=2,
           label=f'Optimal k={optimal_k}')

ax.legend()
plt.tight_layout()
plt.show()

print(f"\nOptimal number of segments: {optimal_k}")
print("\nMedian ARI by k:")
```

```python
for k in k_range:
    print(f"  k={k}: {np.median(ari_results[k]):.3f}")
```

7 Practical Recommendations

Algorithm Selection Guidelines

1. **Start with exploration:**
   - Hierarchical clustering with different linkages
   - k-means with range of k values
   - Self-organizing maps for visualization
2. **Assess data structure:**
   - Run global stability analysis
   - Generate scree plots (with caution)
   - Inspect dendrograms
3. **Refine with advanced methods:**
   - Finite mixture models for probabilistic segments
   - Bagged clustering for niche segment detection
   - VSBD for high-dimensional binary data
4. **Validate thoroughly:**
   - Segment-level stability analysis
   - Business interpretation and actionability
   - Cross-validation with holdout samples

Critical Questions for Practitioners

Before finalizing a segmentation solution, ask:

1. **Stability:** Do segments replicate across bootstrap samples?
2. **Interpretability:** Can we clearly describe each segment?
3. **Actionability:** Can we target these segments differently?
4. **Size viability:** Are segments large enough to be profitable?
5. **Accessibility:** Can we reach these segments through marketing channels?

8 Conclusion

Key Takeaways

Essential Insights:

1. **No universally best algorithm:** Algorithm choice shapes results
2. **Consumer data lacks natural structure:** Expect reproducible or constructive segmentation, not natural clusters
3. **Stability is paramount:** Reproducibility across samples indicates meaningful segments
4. **Domain knowledge essential:** Statistical methods guide, but business insight determines final selection
5. **Multiple solutions valid:** Explore alternatives, document rationale for final choice

References

[1] Dolnicar, S., Grün, B., and Leisch, F. (2018). *Market Segmentation Analysis: Understanding It, Doing It, and Making It Useful*. Springer.

[2] Hennig, C. (2015). What are the true clusters? *Pattern Recognition Letters*, 64, 53-62.

[3] Wedel, M., and Kamakura, W.A. (2000). *Market Segmentation: Conceptual and Methodological Foundations*. Kluwer Academic Publishers.

[4] Fraley, C., and Raftery, A.E. (2002). Model-based clustering, discriminant analysis, and density estimation. *Journal of the American Statistical Association*, 97(458), 611-631.

[5] Kohonen, T. (2001). *Self-Organizing Maps*. 3rd ed., Springer.

9 Case Studies: From Theory to Practice

9.1 Case Study 1: Tourist Vacation Activities

Business Context

A tourism organization wants to segment tourists based on their vacation activity patterns to design targeted marketing campaigns and service offerings.
Dataset: 7 tourists, 3 vacation activities (Beach, Action, Culture)
Objective: Identify natural groupings of tourists with similar activity preferences
Approach: Distance-based hierarchical clustering with multiple linkage methods

9.1.1 Data Structure and Initial Exploration

Table 4: Tourist Vacation Activities Dataset

Tourist	Beach (%)	Action (%)	Culture (%)
Anna	100	0	0
Bill	100	0	0
Frank	60	40	0
Julia	70	0	30
Maria	80	0	20
Michael	0	90	10
Tom	50	20	30

Initial Observations

Examining the raw data reveals:

- **Anna & Bill:** Identical profiles (100% beach orientation)
- **Michael:** Distinct outlier (no beach interest, high action)
- **Julia & Maria:** Similar beach+culture mix
- **Frank & Tom:** Mixed activity patterns

Expected outcome: 2-3 meaningful segments depending on granularity desired

9.1.2 Distance Matrix Analysis

```python
import numpy as np
import pandas as pd
from scipy.spatial.distance import pdist, squareform
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

# Create dataset
data = pd.DataFrame({
    'Beach': [100, 100, 60, 70, 80, 0, 50],
    'Action': [0, 0, 40, 0, 0, 90, 20],
    'Culture': [0, 0, 0, 30, 20, 10, 30]
}, index=['Anna', 'Bill', 'Frank', 'Julia', 'Maria', 'Michael', 'Tom'])

# Calculate Manhattan distance matrix
dist_manhattan = pdist(data, metric='cityblock')
dist_matrix = squareform(dist_manhattan)

# Create distance matrix DataFrame
dist_df = pd.DataFrame(dist_matrix,
                       index=data.index,
                       columns=data.index)

print("Manhattan Distance Matrix:")
print(dist_df.astype(int))

# Output:
#            Anna  Bill  Frank  Julia  Maria  Michael  Tom
# Anna          0     0     80     60     40      200  100
# Bill          0     0     80     60     40      200  100
# Frank        80    80      0     80     80      120   60
# Julia        60    60     80      0     20      180   40
# Maria        40    40     80     20      0      180   60
# Michael     200   200    120    180    180        0  140
# Tom         100   100     60     40     60      140    0
```

Distance Matrix Insights

Key Patterns:

- Anna-Bill distance = 0 (identical twins)
- Julia-Maria distance = 20 (close neighbors)
- Michael has large distances to everyone (outlier status)
- Tom shows moderate distances to most (bridge consumer)

9.1.3 Hierarchical Clustering Analysis

```
# Perform hierarchical clustering with different linkages
linkages = ['single', 'complete', 'average', 'ward']
results = {}

fig, axes = plt.subplots(2, 2, figsize=(14, 10))
axes = axes.ravel()

for idx, method in enumerate(linkages):
    # Compute linkage
    Z = linkage(dist_manhattan, method=method)
    results[method] = Z

    # Plot dendrogram
    ax = axes[idx]
    dendrogram(Z, labels=data.index, ax=ax)
    ax.set_title(f'{method.capitalize()} Linkage', fontsize=14,
    ↪  fontweight='bold')
    ax.set_xlabel('Tourist')
    ax.set_ylabel('Distance')

    # Add cut line for k=3
    if method in ['complete', 'average']:
        ax.axhline(y=100, color='red', linestyle='--',
                   label='k=3 cut', linewidth=2)
        ax.legend()

plt.tight_layout()
plt.show()
```

```

# Extract segments for complete linkage (k=3)
from scipy.cluster.hierarchy import fcluster

segments_complete = fcluster(results['complete'], t=3, criterion='maxclust')
data['Segment_Complete'] = segments_complete

print("\nSegment Assignment (Complete Linkage, k=3):")
for segment in range(1, 4):
    members = data[data['Segment_Complete'] == segment].index.tolist()
    print(f"Segment {segment}: {', '.join(members)}")
```

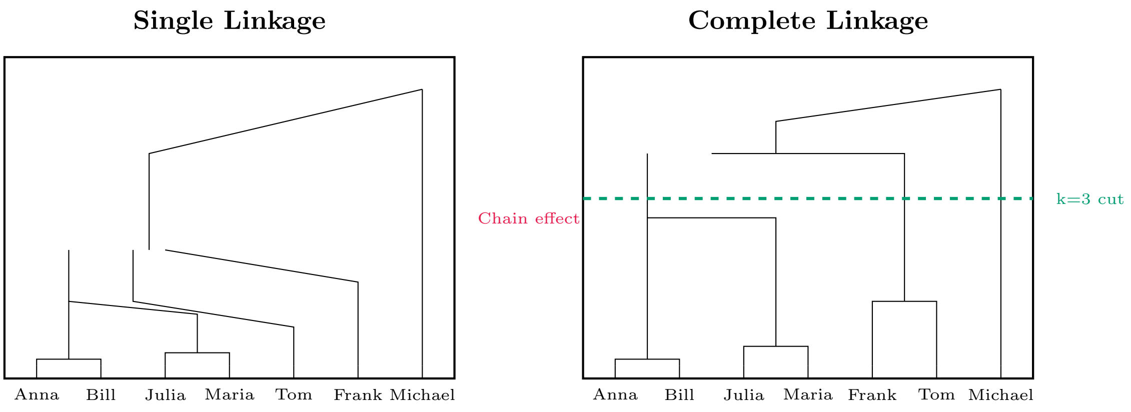

Figure 9: Comparison of single and complete linkage hierarchical clustering on tourist data. Single linkage shows chaining effect, while complete linkage produces more compact, interpretable segments.

9.1.4 Segment Profiling

```
# Calculate segment profiles
segment_profiles = data.groupby('Segment_Complete')[['Beach', 'Action',
↪  'Culture']].mean()

print("\nSegment Profiles (Complete Linkage, k=3):")
print(segment_profiles.round(1))

# Visualize segment profiles
fig, ax = plt.subplots(figsize=(10, 6))

x = np.arange(len(segment_profiles.columns))
width = 0.25

for i, segment in enumerate([1, 2, 3]):
    segment_data = segment_profiles.loc[segment]
    ax.bar(x + i*width, segment_data, width,
           label=f'Segment {segment}', alpha=0.8)

# Add population mean
pop_mean = data[['Beach', 'Action', 'Culture']].mean()
ax.plot(x + width, pop_mean, 'ro-', markersize=10, linewidth=2,
        label='Population Mean')

ax.set_xlabel('Activity', fontsize=12)
ax.set_ylabel('Percentage of Time', fontsize=12)
ax.set_title('Segment Activity Profiles', fontsize=14, fontweight='bold')
ax.set_xticks(x + width)
ax.set_xticks(segment_profiles.columns)
ax.legend()
ax.grid(axis='y', alpha=0.3)

plt.tight_layout()
plt.show()
```

Segment Interpretation Naming

Segment 1: "Beach Lovers" (Anna, Bill, Julia, Maria)

- High beach orientation (85% average)
- Minimal action activities
- Some cultural interest (Julia, Maria)
- **Marketing strategy:** Beach resort packages, relaxation focus

Segment 2: "Mixed Activity Seekers" (Frank, Tom)

- Balanced activity portfolio
- Moderate beach, action, and culture
- **Marketing strategy:** Diverse destination offerings, flexibility

Segment 3: "Action Enthusiasts" (Michael)

- No beach interest
- Dominant action focus (90%)
- Niche segment (single member in sample)
- **Marketing strategy:** Adventure tourism, extreme sports

9.2 Case Study 2: Austrian Winter Vacation Activities

Business Context

An Austrian tourism board seeks to segment winter tourists based on 27 different vacation activities to develop targeted marketing campaigns for different tourist types.
Dataset: 2,961 tourists, 27 binary activities
Challenge: High-dimensional binary data with potential niche segments
Approach: Bagged clustering to identify both major and niche segments

9.2.1 Dataset Overview

The dataset includes diverse winter activities: - **Winter sports:** Alpine skiing, cross-country skiing, snowboarding, ice-skating - **Wellness:** Spa visits, health facilities, pool/sauna - **Culture:** Museums, theater/opera, concerts - **Recreation:** Hiking, walks, excursions, relaxation - **Entertainment:** Evening activities, discos/bars, shopping

```python
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
import matplotlib.pyplot as plt

# Simulate winter activities data (binary)
np.random.seed(42)
n_tourists = 2961
n_activities = 27

# Create realistic activity patterns
# Segment 1: Alpine skiers (35%)
n_seg1 = int(0.35 * n_tourists)
seg1 = np.random.binomial(1, 0.8, (n_seg1, 5))  # High winter sports
seg1 = np.hstack([seg1, np.random.binomial(1, 0.3, (n_seg1, 22))])

# Segment 2: Health tourists (11%)
n_seg2 = int(0.11 * n_tourists)
seg2 = np.random.binomial(1, 0.2, (n_seg2, 27))
seg2[:, [9, 10, 26]] = np.random.binomial(1, 0.7, (n_seg2, 3))  # Spa, health,
↪ pool

# Segment 3: Cultural tourists (16%)
n_seg3 = int(0.16 * n_tourists)
seg3 = np.random.binomial(1, 0.3, (n_seg3, 27))
seg3[:, [19, 20, 21, 22]] = np.random.binomial(1, 0.6, (n_seg3, 4))  # Culture

# Segment 4: Mixed activities (38%)
n_seg4 = n_tourists - n_seg1 - n_seg2 - n_seg3
seg4 = np.random.binomial(1, 0.4, (n_seg4, 27))

# Combine
data = np.vstack([seg1, seg2, seg3, seg4])
true_labels = np.array([0]*n_seg1 + [1]*n_seg2 + [2]*n_seg3 + [3]*n_seg4)

activity_names = [
    'alpine_skiing', 'cross_country', 'snowboarding', 'ice_skating',
    ↪ 'ski_touring',
    'sleigh_riding', 'tennis', 'horseback_riding', 'going_to_spa',
    ↪ 'health_facilities',
    'hiking', 'walks', 'organized_excursions', 'excursions', 'relaxing',
    'evening_out', 'discos_bars', 'shopping', 'sight_seeing', 'museums',
    'theater_opera', 'heurigen', 'concerts', 'tyrolean_evenings',
    ↪ 'local_events',
    'pool_sauna', 'other'
]

df = pd.DataFrame(data, columns=activity_names[:27])
```

9.2.2 Bagged Clustering Implementation

```
def bagged_clustering(data, base_k=10, n_bootstrap=50):
    """
    Perform bagged clustering on binary activity data.
    """
    n_samples = len(data)
    all_centroids = []

    print(f"Performing bagged clustering with {n_bootstrap} bootstrap
    ↪  samples...")

    for b in range(n_bootstrap):
        # Bootstrap sample
        indices = np.random.choice(n_samples, size=n_samples, replace=True)
        bootstrap_data = data[indices]

        # k-means with base_k clusters
        kmeans = KMeans(n_clusters=base_k, n_init=10, random_state=b)
        kmeans.fit(bootstrap_data)

        # Store centroids
        all_centroids.append(kmeans.cluster_centers_)

        if (b + 1) % 10 == 0:
            print(f"  Completed {b+1}/{n_bootstrap} bootstrap samples")

    # Stack all centroids
    all_centroids = np.vstack(all_centroids)
    print(f"\nCollected {len(all_centroids)} centroids")

    # Hierarchical clustering on centroids
    linkage_matrix = linkage(all_centroids, method='average')

    return linkage_matrix, all_centroids

# Run bagged clustering
linkage_matrix, all_centroids = bagged_clustering(df.values,
                                                    base_k=10,
                                                    n_bootstrap=50)

# Plot dendrogram
plt.figure(figsize=(12, 6))
dendrogram(linkage_matrix, no_labels=True)
plt.xlabel('Centroid Index')
plt.ylabel('Distance')
plt.title('Bagged Clustering Dendrogram\nAustrian Winter Activities',
          fontsize=14, fontweight='bold')
plt.axhline(y=2.5, color='red', linestyle='--', linewidth=2, label='k=5 cut')
plt.legend()
plt.tight_layout()
plt.show()

# Extract k=5 segments
final_k = 5
centroid_labels = fcluster(linkage_matrix, t=final_k, criterion='maxclust')

# Get final segment centers
final_centers = np.array([all_centroids[centroid_labels == j].mean(axis=0)
                          for j in range(1, final_k+1)])

# Assign original tourists to segments
from scipy.spatial.distance import cdist
distances = cdist(df.values, final_centers)
tourist_segments = np.argmin(distances, axis=1)

df['Segment'] = tourist_segments
```

9.2.3 Segment Profiles and Business Insights

```
# Calculate segment activity probabilities
segment_profiles = df.groupby('Segment').mean() * 100  # Convert to percentages

# Visualize top activities per segment
fig, axes = plt.subplots(1, 5, figsize=(18, 6))

for seg in range(5):
    ax = axes[seg]
    profile = segment_profiles.iloc[seg].sort_values(ascending=False)[:10]

    ax.barh(range(len(profile)), profile.values, color=f'C{seg}')
    ax.set_yticks(range(len(profile)))
    ax.set_yticklabels(profile.index, fontsize=8)
    ax.set_xlabel('Activity Rate (%)')
    ax.set_title(f'Segment {seg+1}\n({(df["Segment"]==seg).sum()} tourists)')
    ax.invert_yaxis()
    ax.grid(axis='x', alpha=0.3)

plt.tight_layout()
plt.show()

# Segment sizes
print("\nSegment Sizes:")
print(df['Segment'].value_counts().sort_index())
print("\nSegment Size Distribution:")
print(df['Segment'].value_counts(normalize=True).sort_index() * 100)
```

10 Advanced Stability Analysis

10.1 Segment-Level Stability Within Solutions (SLSw)

Traditional global stability analysis assesses entire segmentation solutions. However, organizations typically target only one or a few segments, not the complete solution. Segment-level stability within solutions (SLSw) addresses this limitation.

The SLSw Principle

Segment-level stability within solutions (SLSw) answers:

- Which **individual segments** are most reproducible?
- Can we identify a single highly stable segment for targeting?
- Does the solution contain attractive niche segments?

Key insight: A globally unstable solution may contain one highly stable, valuable segment worth targeting.

10.1.1 SLSw Algorithm

Segment-Level Stability Within Solutions

Based on Hennig (2007) and Dolnicar & Leisch (2017)

```
1: Extract partition with $k$ segments: $S_1, \dots, S_k$
2: Draw $b$ bootstrap samples (typical: $b = 100$)
3: for each bootstrap sample $i = 1$ to $b$ do
4:     Extract $k$ segments: $S_1^{(i)}, \dots, S_k^{(i)}$
5: end for
6: for each original segment $h = 1$ to $k$ do
7:     for each bootstrap sample $i = 1$ to $b$ do
8:         Find bootstrap segment $S_j^{(i)}$ with maximum Jaccard similarity to $S_h$
9:         Store Jaccard index: $J_h^{(i)}$
10:    end for
11:    Compute distribution of $\{J_h^{(1)}, \dots, J_h^{(b)}\}$
12: end for
13: return Boxplots of Jaccard indices for each segment
```

Jaccard Index for Segment Similarity

For two segments A and B:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{16}$$

where $|A \cap B|$ is number of consumers in both segments.
Interpretation:

- $J = 1$: Perfect agreement (identical segments)
- $J = 0.8$: High stability (80% overlap)
- $J < 0.5$: Low stability (artificial segment)

10.1.2 Python Implementation: SLSw Analysis

```python
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

def jaccard_index(set1, set2):
    """Calculate Jaccard index between two sets."""
    intersection = len(set(set1) & set(set2))
    union = len(set(set1) | set(set2))
    return intersection / union if union > 0 else 0

def segment_level_stability_within(data, k, n_bootstrap=100, random_state=42):
    """
    Compute segment-level stability within solutions (SLSw).

    Returns: Dictionary with Jaccard indices for each segment
    """
    np.random.seed(random_state)
    n_samples = len(data)

    # Fit original solution
    kmeans_orig = KMeans(n_clusters=k, n_init=50, random_state=random_state)
    labels_orig = kmeans_orig.fit_predict(data)

    # Store Jaccard indices for each segment
    jaccard_results = {seg: [] for seg in range(k)}

    print(f"Computing SLSw for k={k} with {n_bootstrap} bootstrap samples...")

    for b in range(n_bootstrap):
        # Bootstrap sample
        indices = np.random.choice(n_samples, size=n_samples, replace=True)
        bootstrap_data = data[indices]

        # Cluster bootstrap sample
        kmeans_boot = KMeans(n_clusters=k, n_init=10, random_state=b)
        labels_boot = kmeans_boot.fit_predict(bootstrap_data)

        # For each original segment, find best matching bootstrap segment
        for orig_seg in range(k):
            # Original segment members (in bootstrap sample indices)
            orig_members = np.where(labels_orig == orig_seg)[0]
            orig_in_boot = np.array([np.where(indices == m)[0][0]
                                     if m in indices else -1
                                     for m in orig_members])
            orig_in_boot = orig_in_boot[orig_in_boot >= 0]

            # Find best matching bootstrap segment
            max_jaccard = 0
            for boot_seg in range(k):
                boot_members = np.where(labels_boot == boot_seg)[0]
                jaccard = jaccard_index(orig_in_boot, boot_members)
                max_jaccard = max(max_jaccard, jaccard)

            jaccard_results[orig_seg].append(max_jaccard)

        if (b + 1) % 20 == 0:
            print(f"  Completed {b+1}/{n_bootstrap} bootstrap samples")

    return jaccard_results

# Example usage
np.random.seed(42)

# Generate data with 3 segments of different stability
stable_seg = np.random.normal([2, 2], [0.3, 0.3], (200, 2))
moderate_seg = np.random.normal([5, 5], [0.8, 0.8], (200, 2))
unstable_seg = np.random.normal([8, 2], [1.5, 1.5], (200, 2))

data = np.vstack([stable_seg, moderate_seg, unstable_seg])

# Compute SLSw
slsw_results = segment_level_stability_within(data, k=3, n_bootstrap=100)

# Visualize
fig, ax = plt.subplots(figsize=(10, 6))

positions = [1, 2, 3]
slsw_data = [slsw_results[i] for i in range(3)]

bp = ax.boxplot(slsw_data, positions=positions, widths=0.6, patch_artist=True,
                boxprops=dict(facecolor='lightblue', edgecolor='blue'),
                medianprops=dict(color='red', linewidth=2),
                whiskerprops=dict(color='blue'),
                capprops=dict(color='blue'))

ax.set_xlabel('Segment Number', fontsize=12)
ax.set_ylabel('Jaccard Coefficient', fontsize=12)
ax.set_title('Segment-Level Stability Within Solutions (SLSw)',
             fontsize=14, fontweight='bold')
ax.set_ylim([0, 1.05])
ax.set_xticks(positions)
ax.grid(True, alpha=0.3, axis='y')
ax.axhline(y=0.8, color='green', linestyle='--', alpha=0.5,
           label='High Stability Threshold')

# Add median values as text
for i, pos in enumerate(positions):
    median_val = np.median(slsw_results[i])
    ax.text(pos, median_val + 0.05, f'{median_val:.3f}',
            ha='center', fontweight='bold')

ax.legend()
plt.tight_layout()
plt.show()

# Print statistics
print("\nSLSw Statistics:")
for seg in range(3):
    print(f"\nSegment {seg+1}:")
    print(f"  Median Jaccard: {np.median(slsw_results[seg]):.3f}")
    print(f"  Mean Jaccard: {np.mean(slsw_results[seg]):.3f}")
    print(f"  Std Dev: {np.std(slsw_results[seg]):.3f}")
```

10.2 Segment-Level Stability Across Solutions (SLSA)

The SLSA Principle

Segment-level stability across solutions (SLSA) tracks how segments evolve as k increases:

Questions addressed:

- Do certain segments persist across different k values?
- Which segments are ”natural” vs. artificially created?
- How are segments split or merged as k changes?

Interpretation:

- **Thick horizontal lines**: Stable segments that persist
- **Branching patterns**: Segments splitting with increasing k
- **Multiple inputs**: Artificial segments created by merging

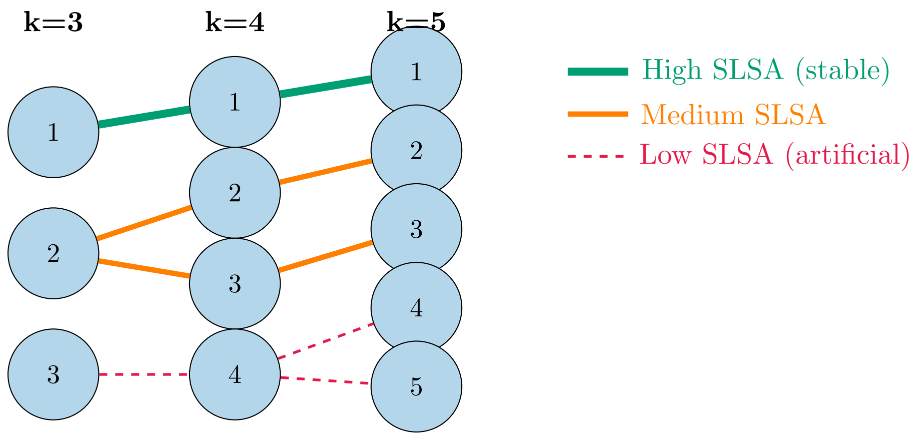

Figure 10: Segment-level stability across solutions (SLSA) visualization. Segment 1 remains highly stable across k values (thick green lines). Segment 2 progressively splits (orange). Segment 3 is unstable and artificial (red dashed).

10.2.1 SLSA with Entropy-Based Stability

Entropy-Based SLSA Measure

For segment S_i in solution with k_i segments, recruiting from segments in solution with k_{i-1} segments:
Let p_j = proportion of consumers in S_i from segment j in previous solution
Entropy:

$$H(S_i) = -\sum_{j=1}^{k_{i-1}} p_j \log(p_j) \tag{17}$$

Normalized SLSA stability:

$$\text{SLSA}(S_i) = 1 - \frac{H(S_i)}{\log(k_{i-1})} \tag{18}$$

Interpretation:

- SLSA $= 1$: All members from single parent segment (maximum stability)
- SLSA $= 0$: Members equally distributed from all parent segments (artificial)
- SLSA > 0.8: Highly stable segment

10.3 Response Style Detection

Response Bias in Survey Data

Survey-based segmentation faces a critical challenge: **response style bias**
Common response styles:

- **Acquiescence:** Tendency to agree with all statements
- **Disacquiescence:** Tendency to disagree with all statements
- **Midpoint:** Always selecting neutral/middle options
- **Extreme response:** Only using endpoint scales

Consequence: Segments may reflect response patterns, not true preferences!

10.3.1 Detecting Response Style Segments

Response Style Indicators

A segment likely reflects response style if:

1. **High agreement across all variables**
   - Acquiescence: >80% agreement on most items
   - Uniform pattern regardless of content
2. **High stability but questionable profile**

- Segment appears stable in SLSw/SLSA analysis
- Profile lacks coherent narrative
- Members agree with contradictory statements

3. **Correlation with demographic ”yes-sayers”**
   - Older respondents often show acquiescence
   - Lower education sometimes correlates with midpoint bias

Action: Exclude response style segments from targeting consideration

10.3.2 Python Implementation: Response Style Detection

```python
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

def detect_response_style(segment_data, threshold_agreement=0.8):
    """
    Detect potential response style bias in a segment.

    Parameters:
    - segment_data: DataFrame with binary/Likert responses
    - threshold_agreement: Minimum proportion for acquiescence flag

    Returns: Dictionary with response style indicators
    """
    n_variables = segment_data.shape[1]

    # Calculate mean agreement per variable
    agreement_rates = segment_data.mean()

    # Check for acquiescence (high agreement across board)
    high_agreement_count = (agreement_rates > threshold_agreement).sum()
    acquiescence_ratio = high_agreement_count / n_variables

    # Check for variance (low variance suggests response style)
    variances = segment_data.var()
    low_variance_count = (variances < 0.1).sum()
    low_variance_ratio = low_variance_count / n_variables

    # Overall response style score
    response_style_score = (acquiescence_ratio + low_variance_ratio) / 2

    results = {
        'acquiescence_ratio': acquiescence_ratio,
        'low_variance_ratio': low_variance_ratio,
        'response_style_score': response_style_score,
        'is_response_style': response_style_score > 0.6,
        'mean_agreement': agreement_rates.mean(),
        'std_agreement': agreement_rates.std()
    }

    return results

# Example: Simulate survey data
np.random.seed(42)

# Segment 1: True preference pattern (varied responses)
true_segment = np.random.binomial(1, [0.2, 0.8, 0.3, 0.7, 0.4], (100, 5))

# Segment 2: Acquiescence bias (agrees with everything)
acquiescence_segment = np.random.binomial(1, 0.85, (100, 5))

# Segment 3: Midpoint bias (moderate on everything)
midpoint_segment = np.random.binomial(1, 0.5, (100, 5))

segments_data = [
    pd.DataFrame(true_segment, columns=[f'Q{i+1}' for i in range(5)]),
    pd.DataFrame(acquiescence_segment, columns=[f'Q{i+1}' for i in range(5)]),
    pd.DataFrame(midpoint_segment, columns=[f'Q{i+1}' for i in range(5)])
]

# Analyze each segment
print("Response Style Analysis:")
print("=" * 60)

for i, seg_data in enumerate(segments_data, 1):
    print(f"\nSegment {i}:")
    results = detect_response_style(seg_data)

    print(f"  Acquiescence Ratio: {results['acquiescence_ratio']:.3f}")
    print(f"  Low Variance Ratio: {results['low_variance_ratio']:.3f}")
    print(f"  Response Style Score: {results['response_style_score']:.3f}")
    print(f"  Response Style Detected: {results['is_response_style']}")
    print(f"  Mean Agreement: {results['mean_agreement']:.3f}")
    print(f"  Std Agreement: {results['std_agreement']:.3f}")

# Visualize patterns
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

segment_names = ['True Preference', 'Acquiescence Bias', 'Midpoint Bias']

for i, (seg_data, ax) in enumerate(zip(segments_data, axes)):
    agreement_rates = seg_data.mean()

    ax.bar(range(5), agreement_rates, color=f'C{i}', alpha=0.7)
    ax.set_xlabel('Question')
    ax.set_ylabel('Agreement Rate')
    ax.set_title(segment_names[i])
    ax.set_ylim([0, 1])
    ax.axhline(y=0.8, color='red', linestyle='--', alpha=0.5, label='Threshold')
    ax.set_xticks(range(5))
    ax.set_xticklabels([f'Q{j+1}' for j in range(5)])
    ax.legend()
    ax.grid(axis='y', alpha=0.3)

```

```python
plt.tight_layout()
plt.show()
```

10.4 Handling Mixed Data Types

Real consumer datasets often contain mixed variable types: binary activities, ordinal ratings, metric measurements.

Mixed Data Challenges

Common variable types in consumer data:

- **Binary:** Purchased/not purchased, activity participation
- **Ordinal:** Likert scales, satisfaction ratings
- **Metric:** Spending amounts, age, income
- **Nominal:** Categorical preferences (brand, color)

Problem: Standard distance measures assume all variables are same type

10.4.1 Gower Distance for Mixed Data

Gower Distance Formula

For consumers i and j with p mixed-type variables:

$$d_{Gower}(i,j) = \frac{\sum_{k=1}^{p} \delta_{ijk} \cdot d_{ijk}}{\sum_{k=1}^{p} \delta_{ijk}} \tag{19}$$

where:

- $\delta_{ijk} = 0$ if variable k is missing for i or j, else 1
- d_{ijk} depends on variable type:

For binary/nominal: $d_{ijk} = \begin{cases} 0 & \text{if } x_{ik} = x_{jk} \\ 1 & \text{otherwise} \end{cases}$

For ordinal/metric: $d_{ijk} = \frac{|x_{ik} - x_{jk}|}{\text{range}_k}$

```python
from sklearn.preprocessing import LabelEncoder
from scipy.spatial.distance import pdist, squareform
import numpy as np

def gower_distance(X, categorical_features=None):
    """
    Compute Gower distance for mixed data types.

    Parameters:
    - X: DataFrame with mixed types
    - categorical_features: List of categorical column names

    Returns: Distance matrix
    """
    X = X.copy()
    n = len(X)

    if categorical_features is None:
        categorical_features = X.select_dtypes(include=['object',
         ↪  'category']).columns

    numerical_features = [c for c in X.columns if c not in categorical_features]

    # Initialize distance matrix
    distance_matrix = np.zeros((n, n))

    for i in range(n):
        for j in range(i+1, n):
            distance = 0
            valid_features = 0

            # Categorical features
            for feat in categorical_features:
                if pd.notna(X.iloc[i][feat]) and pd.notna(X.iloc[j][feat]):
                    distance += 0 if X.iloc[i][feat] == X.iloc[j][feat] else 1
                    valid_features += 1

            # Numerical features (normalized)
            for feat in numerical_features:
                if pd.notna(X.iloc[i][feat]) and pd.notna(X.iloc[j][feat]):
                    feat_range = X[feat].max() - X[feat].min()
                    if feat_range > 0:
                        distance += abs(X.iloc[i][feat] - X.iloc[j][feat]) /
                         ↪  feat_range
                    valid_features += 1

            distance_matrix[i, j] = distance / valid_features if valid_features
             ↪  > 0 else 0
            distance_matrix[j, i] = distance_matrix[i, j]

    return distance_matrix

# Example usage with mixed data
data_mixed = pd.DataFrame({
    'age': [25, 30, 28, 45, 50],
    'income': [30000, 45000, 35000, 70000, 65000],
    'purchased': [1, 0, 1, 0, 1],
    'preferred_brand': ['A', 'B', 'A', 'C', 'C']
})

dist_matrix = gower_distance(data_mixed,
 ↪  categorical_features=['preferred_brand'])
print("Gower Distance Matrix:")
print(dist_matrix)
```

11 Complete Market Segmentation Workflow

11.1 End-to-End Process Overview

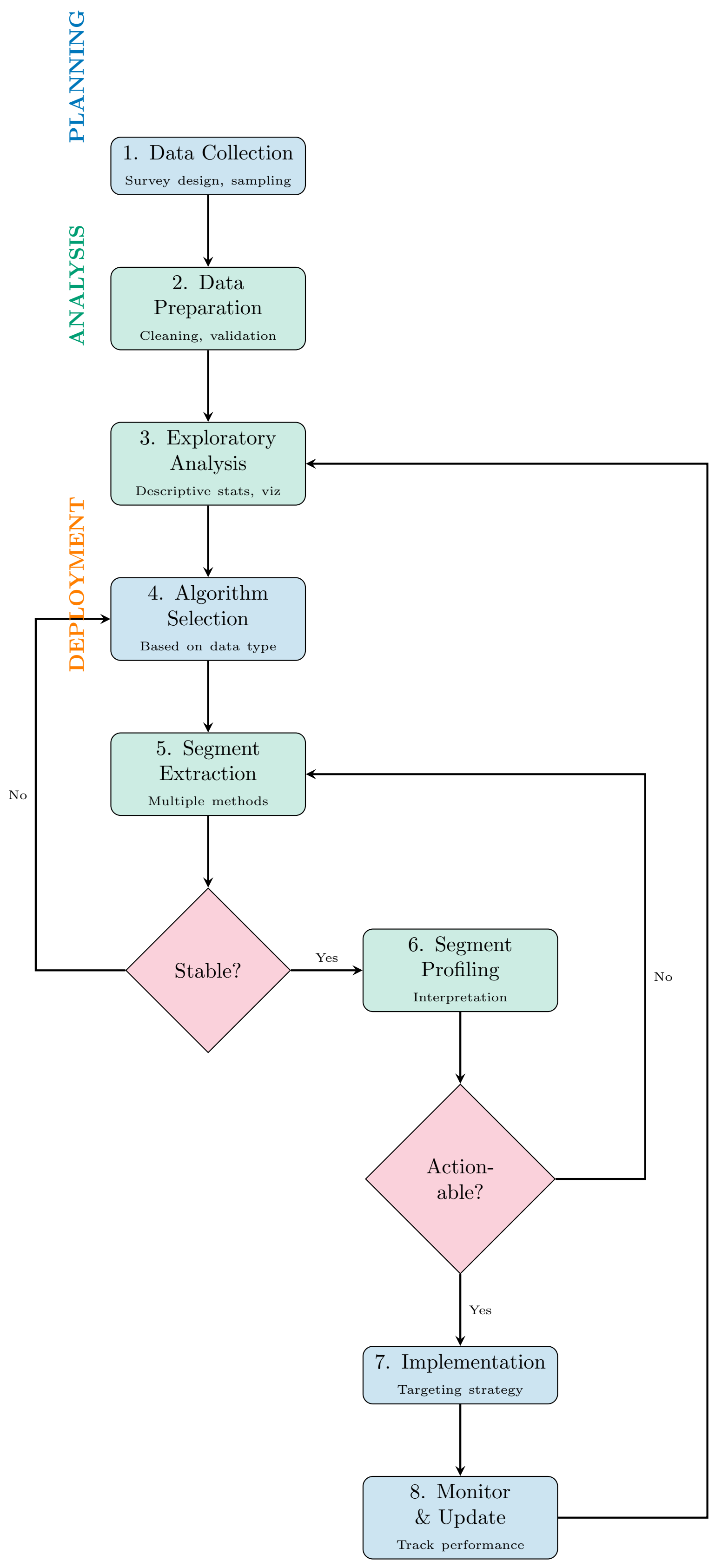

Figure 11: Complete market segmentation workflow from data collection to implementation. Iterative loops allow refinement when stability or actionability criteria are not met.

11.2 Decision Tree for Algorithm Selection

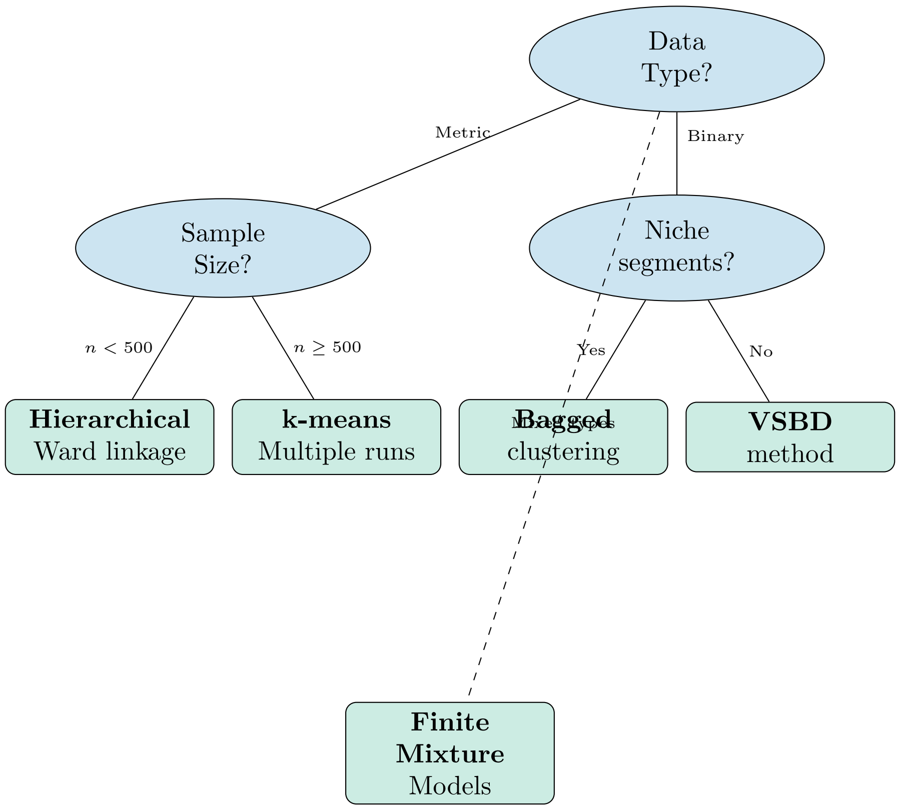

Figure 12: Decision tree for selecting appropriate segmentation algorithm based on data characteristics and business requirements.

11.3 Stage-by-Stage Checklists

Stage 1: Data Preparation Checklist

☐ Survey design reviewed by domain experts

☐ Sample size adequate ($n \geq 200$ minimum, $n \geq 500$ preferred)

☐ Missing data handled appropriately

☐ Outliers identified and investigated

☐ Variables standardized if on different scales

☐ Response style bias assessed

☐ Multicollinearity checked (VIF < 10)

☐ Data saved in analysis-ready format

Stage 2: Algorithm Selection Checklist

- ☐ Data type documented (binary/ordinal/metric/mixed)
- ☐ Appropriate distance measure selected
- ☐ Multiple algorithms chosen for comparison
- ☐ Algorithm assumptions match data characteristics
- ☐ Computational feasibility verified
- ☐ Software implementation tested
- ☐ Random seed set for reproducibility

Stage 3: Segment Extraction Checklist

- ☐ Multiple k values tested (typically $k = 2$ to $k = 8$)
- ☐ k-means run 50+ times with different initializations
- ☐ Dendrogram inspected for hierarchical methods
- ☐ Scree plot generated and examined
- ☐ Global stability analysis completed ($b \geq 50$ bootstrap samples)
- ☐ Segment-level stability (SLSw) computed
- ☐ Optimal k selected based on stability, not just fit
- ☐ Results documented with reproducibility details

Stage 4: Segment Validation Checklist

- ☐ Segments differ significantly on segmentation variables
- ☐ Segments interpretable with clear narratives
- ☐ Segments not artifacts of response style
- ☐ Segment sizes viable for business ($\geq 5\%$ preferred)
- ☐ Segments accessible through marketing channels
- ☐ Segments stable in SLSA analysis
- ☐ Segments validated on holdout sample if possible
- ☐ Stakeholder buy-in obtained

11.4 Common Pitfalls and Solutions

Table 5: Common Segmentation Pitfalls and Recommended Solutions

Pitfall	**Consequence**	**Solution**
Using only one algorithm	Biased solution reflecting algorithm assumptions	Test 3+ different methods; compare results
Selecting k by elbow method alone	Artificial segment count	Prioritize stability analysis over fit measures
Ignoring response style bias	Segments reflect survey artifacts, not preferences	Screen for acquiescence patterns; exclude biased segments
Accepting unstable segments	Targeting non-reproducible groups	Require median Jaccard > 0.7 in SLSw analysis
Creating too many segments	Unactionable, overlapping groups	Limit to 3-5 segments for practical targeting
Not standardizing variables	Distance dominated by high-variance features	Always standardize metric variables before clustering
Single random initialization (k-means)	Local optima, non-optimal solution	Run 50+ times; select best WCSS
Overfitting to sample	Poor generalization	Validate on holdout data; assess SLSA stability

11.5 Segment Profiling Best Practices

Creating Actionable Segment Profiles

A complete segment profile should include:

1. Demographic Description

- Age distribution, gender split, education level
- Income brackets, household composition
- Geographic concentration

2. Behavioral Characteristics

- Product/service usage patterns
- Purchase frequency and monetary value
- Channel preferences (online, retail, mobile)

3. Psychographic Insights

- Values, attitudes, lifestyle preferences
- Motivations and pain points

- Brand perceptions and preferences

4. **Segment Naming**
   - Memorable, descriptive names (not "Segment 1")
   - Captures essence of consumer group
   - Examples: "Budget Seekers", "Premium Loyalists", "Convenience Focused"
5. **Size and Value**
   - Absolute size (number of consumers)
   - Percentage of total market
   - Current and potential revenue
   - Lifetime value estimates

11.5.1 Python Implementation: Automated Profiling Report

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats

def generate_segment_profile_report(data, segment_col='segment',
                                    segmentation_vars=None,
                                    demographic_vars=None):
    """
    Generate comprehensive segment profiling report.
    """
    segments = data[segment_col].unique()
    n_segments = len(segments)

    report = {}

    print("=" * 80)
    print("SEGMENT PROFILING REPORT")
    print("=" * 80)

    for seg in sorted(segments):
        seg_data = data[data[segment_col] == seg]
        pop_size = len(seg_data)
        pop_pct = 100 * pop_size / len(data)

        print(f"\n{'='*80}")
        print(f"SEGMENT {seg+1}")
        print(f"{'='*80}")
        print(f"Size: {pop_size:,} consumers ({pop_pct:.1f}% of total)")

        seg_report = {
            'size': pop_size,
            'percentage': pop_pct
        }

        # Segmentation variables profile
        if segmentation_vars:
            print(f"\nSegmentation Variables:")
            print("-" * 40)

            seg_means = seg_data[segmentation_vars].mean()
            pop_means = data[segmentation_vars].mean()

            for var in segmentation_vars:
                seg_val = seg_means[var]
                pop_val = pop_means[var]
                diff = seg_val - pop_val
                pct_diff = 100 * diff / pop_val if pop_val != 0 else 0

                print(f"  {var:20s}: {seg_val:6.2f} "
                      f"(pop: {pop_val:6.2f}, "
                      f"{'↑' if diff > 0 else '↓'}{abs(pct_diff):5.1f}%)")

        # Demographics profile
        if demographic_vars:
            print(f"\nDemographic Profile:")
            print("-" * 40)

            for var in demographic_vars:
                if data[var].dtype in ['int64', 'float64']:
                    # Numeric variable
                    seg_val = seg_data[var].mean()
                    pop_val = data[var].mean()
                    print(f"  {var:20s}: {seg_val:.1f} (pop: {pop_val:.1f})")
                else:
                    # Categorical variable
                    seg_dist = seg_data[var].value_counts(normalize=True)
                    print(f"  {var}:")
                    for cat, pct in seg_dist.head(3).items():
                        print(f"    {cat}: {100*pct:.1f}%")

        # Statistical distinctiveness
        print(f"\nStatistical Tests:")
        print("-" * 40)

        if segmentation_vars:
            # ANOVA F-statistic
            other_data = data[data[segment_col] != seg]

            for var in segmentation_vars[:5]:  # Top 5 variables
                seg_vals = seg_data[var].values
                other_vals = other_data[var].values

                f_stat, p_val = stats.f_oneway(seg_vals, other_vals)

                print(f"  {var:20s}: F={f_stat:6.2f}, "
                      f"p={'<0.001' if p_val < 0.001 else f'{p_val:.3f}'}")

        report[f'segment_{seg+1}'] = seg_report

    return report

# Example usage
np.random.seed(42)

# Generate sample data
n_total = 1000

# Create 3 segments with distinct profiles
seg1_data = pd.DataFrame({
    'price_sensitivity': np.random.normal(8, 1, 350),
    'quality_importance': np.random.normal(3, 1, 350),
    'brand_loyalty': np.random.normal(2, 1, 350),
    'age': np.random.normal(35, 10, 350),
    'income': np.random.normal(45000, 10000, 350),
    'segment': 0
})

seg2_data = pd.DataFrame({
    'price_sensitivity': np.random.normal(3, 1, 400),
    'quality_importance': np.random.normal(8, 1, 400),
    'brand_loyalty': np.random.normal(7, 1, 400),
    'age': np.random.normal(45, 10, 400),
    'income': np.random.normal(75000, 15000, 400),
    'segment': 1
})

seg3_data = pd.DataFrame({
    'price_sensitivity': np.random.normal(5, 1, 250),
    'quality_importance': np.random.normal(5, 1, 250),
    'brand_loyalty': np.random.normal(4, 1, 250),
    'age': np.random.normal(28, 5, 250),
    'income': np.random.normal(55000, 12000, 250),
    'segment': 2
})

data = pd.concat([seg1_data, seg2_data, seg3_data], ignore_index=True)

# Generate report
segmentation_vars = ['price_sensitivity', 'quality_importance', 'brand_loyalty']
demographic_vars = ['age', 'income']

report = generate_segment_profile_report(data,
                                          segmentation_vars=segmentation_vars,
                                          demographic_vars=demographic_vars)
```

11.6 From Segmentation to Targeting Strategy

Strategic Segment Selection

Not all segments deserve equal investment. Prioritize based on:
Attractiveness Criteria:

1. **Size:** Large enough to be profitable
2. **Growth:** Expanding vs. declining segments
3. **Profitability:** High lifetime value potential
4. **Accessibility:** Reachable through available channels
5. **Stability:** Reproducible across bootstrap samples
6. **Differential response:** Unique needs requiring different offerings

Table 6: Segment Prioritization Matrix Example

Segment	**Size** (0-10)	**Value** (0-10)	**Stability** (0-10)	**Access** (0-10)	**Total**	**Priority**
Premium Loyalists	6	10	9	8	33	**1**
Convenience Focused	8	7	8	9	32	**2**
Budget Seekers	9	4	6	7	26	3
Niche Enthusiasts	2	9	5	4	20	4

11.7 Reporting to Stakeholders

Effective Segmentation Presentation

Executive Summary (1 page):

- Number of segments identified
- Brief characterization of each
- Recommended primary and secondary targets
- Expected business impact

Detailed Profiles (2-3 pages per segment):

- Persona-style description with representative quote
- Visual profile (bar charts, radar plots)
- Demographics, behaviors, attitudes
- Size and value metrics
- Targeting recommendations

Technical Appendix:

- Methodology description
- Algorithms tested and selection rationale
- Stability analysis results
- Statistical validation
- Reproducibility details

11.8 Software Implementation Guide

Table 7: Python vs R for Market Segmentation

Aspect	**Python**	**R**
Hierarchical	`scipy.cluster.hierarchy`	`hclust()`, `agnes()`
k-means	`sklearn.cluster.KMeans`	`kmeans()`, `cclust()`
Finite Mixtures	`sklearn.mixture.GaussianMixture`	`flexmix`, `mclust`
SOM	`minisom` package	`kohonen` package
Stability	Custom implementation	`fpc::clusterboot()`
Visualization	`matplotlib`, `seaborn`	`ggplot2`, base plots
Ecosystem	General purpose, ML focus	Statistical focus, segmentation-specific packages

12 Final Recommendations

Key Success Factors

1. **Prioritize Stability Over Fit**
   - Reproducible segments more valuable than perfect fit to one sample
   - Bootstrap stability analysis is non-negotiable
   - Accept that consumer data lacks natural clusters
2. **Use Multiple Methods**
   - Compare 3+ algorithms (e.g., Ward's, k-means, finite mixtures)
   - Look for segments appearing across multiple solutions
   - Document sensitivity to algorithm choice
3. **Focus on Business Actionability**
   - Statistical optimality secondary to business usability

- Prefer 3-5 segments over 8+ for practical targeting
- Ensure segments inform specific marketing actions

4. **Validate Thoroughly**
   - Holdout validation when sample size permits
   - Check for response style artifacts
   - Confirm segments differ on outcome variables (sales, satisfaction)
5. **Monitor and Update**
   - Consumer preferences evolve; segments aren't permanent
   - Re-run analysis annually or when market shifts
   - Track segment migration over time

Questions for Continuous Improvement

After implementing segmentation strategy:

1. Are we successfully reaching each target segment through chosen channels?
2. Do segments respond differently to our marketing mix as predicted?
3. What is the actual lifetime value of each segment vs. projections?
4. Are segment sizes stable or shifting over time?
5. Should we refine segmentation variables based on learnings?
6. Do emerging consumer trends suggest need for re-segmentation?

13 Conclusion: The Art and Science of Segmentation

Market segmentation combines rigorous statistical methodology with business intuition. This comprehensive guide has demonstrated that:

1. **Algorithm choice matters:** Different methods impose different structural assumptions, shaping results
2. **Stability is paramount:** Reproducibility across bootstrap samples indicates meaningful segments
3. **No universal "best":** Context-dependent selection based on data characteristics and business objectives
4. **Validation is essential:** Multiple complementary approaches reduce risk of spurious solutions

5. **Actionability trumps sophistication:** Simpler, interpretable segments often outperform complex models in practice

The methods and workflows presented equip practitioners to conduct rigorous, reproducible market segmentation analysis that translates statistical findings into competitive business advantage.

The goal is not to find perfect segments,
but to identify reproducible, actionable consumer groups
that enable differentiated marketing strategy.

References

[1] Dolnicar, S., Grün, B., and Leisch, F. (2018). *Market Segmentation Analysis: Understanding It, Doing It, and Making It Useful.* Springer.

[2] Hennig, C. (2007). Cluster-wise assessment of cluster stability. *Computational Statistics & Data Analysis*, 52(1), 258-271.

[3] Leisch, F. (1999). Bagged clustering. Working Paper 51, SFB Adaptive Information Systems and Modeling in Economics and Management Science, Vienna University of Economics.

[4] Wedel, M., and Kamakura, W.A. (2000). *Market Segmentation: Conceptual and Methodological Foundations.* 2nd ed., Kluwer Academic Publishers.

[5] Fraley, C., and Raftery, A.E. (2002). Model-based clustering, discriminant analysis, and density estimation. *Journal of the American Statistical Association*, 97(458), 611-631.

[6] Kohonen, T. (2001). *Self-Organizing Maps.* 3rd ed., Springer Series in Information Sciences.

[7] Ward, J.H. (1963). Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301), 236-244.

[8] Gower, J.C. (1971). A general coefficient of similarity and some of its properties. *Biometrics*, 27(4), 857-871.