```
# Importing necessary libraries
import pandas as pd  # For data manipulation and analysis
from sklearn.decomposition import PCA  # For Principal Component Analysis (PCA) to reduce dimensions
import numpy as np  # For numerical operations
from sklearn import preprocessing  # For scaling and normalization of data
from bioinfokit.visuz import cluster  # For data visualization, specifically clustering
import matplotlib.pyplot as plt  # For plotting graphs
from sklearn.cluster import KMeans  # For K-Means clustering algorithm
from sklearn.utils import resample  # For resampling methods like bootstrapping
from sklearn.metrics import adjusted_rand_score  # For evaluating clustering results using Adjusted Rand Index


pip install bioinfokit #Install the bioinfokit package which provides tools for bioinformatics analysis and visualization
```

```
Requirement already satisfied: bioinfokit in /usr/local/lib/python3.11/dist-packages (2.1.4)
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (2.2.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (2.0.2)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (3.10.0)
Requirement already satisfied: scipy in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (1.15.2)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (1.6.1)
Requirement already satisfied: seaborn in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (0.13.2)
Requirement already satisfied: matplotlib-venn in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (1.1.2)
Requirement already satisfied: tabulate in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (0.9.0)
Requirement already satisfied: statsmodels in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (0.14.4)
Requirement already satisfied: textwrap3 in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (0.9.2)
Requirement already satisfied: adjustText in /usr/local/lib/python3.11/dist-packages (from bioinfokit) (1.3.0)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib->bioinfokit) (1.3.2)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.11/dist-packages (from matplotlib->bioinfokit) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib->bioinfokit) (4.57.0)
Requirement already satisfied: kiwisolver>=1.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib->bioinfokit) (1.4.8)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (from matplotlib->bioinfokit) (24.2)
Requirement already satisfied: pillow>=8 in /usr/local/lib/python3.11/dist-packages (from matplotlib->bioinfokit) (11.2.1)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.11/dist-packages (from matplotlib->bioinfokit) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.11/dist-packages (from matplotlib->bioinfokit) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas->bioinfokit) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas->bioinfokit) (2025.2)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn->bioinfokit) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn->bioinfokit) (3.6.0)
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.11/dist-packages (from statsmodels->bioinfokit) (1.0.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7->matplotlib->bioinfokit) (1.17.0)
```

```
# Reading the McDonald's dataset from the CSV file into a DataFrame
data = pd.read_csv("mcdonalds.csv")  # Load the dataset into 'data' DataFrame

# Reading the same dataset into a second DataFrame
data1 = pd.read_csv("mcdonalds.csv")  # Load the dataset again into 'data1' DataFrame

# Displaying the list of column names in the dataset
data.columns.values.tolist()  # Get the column names of the dataset as a list
```

```
['yummy',
 'convenient',
 'spicy',
 'fattening',
 'greasy',
 'fast',
 'cheap',
```

```
        'tasty',
        'expensive',
        'healthy',
        'disgusting',
        'Like',
        'Age',
        'VisitFrequency',
        'Gender']
```

```
# Getting the shape of the dataset, which returns the number of rows and columns
data.shape
```

```
(1453, 15)
```

```
# Displaying the first 3 rows of the dataset to inspect the data
data.head(3)
```

	yummy	convenient	spicy	fattening	greasy	fast	cheap	tasty	expensive	healthy	disgusting	Like	Age	VisitFrequency	Gender
0	No	Yes	No	Yes	No	Yes	Yes	No	Yes	No	No	-3	61	Every three months	Female
1	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	No	2	51	Every three months	Female
2	No	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	1	62	Every three months	Female

```
# Selecting the first 11 columns of the dataset and replacing 'Yes' with 1 and 'No' with 0
MD=data.iloc[:,0:11].replace("Yes",1).replace("No",0)
# Calculating the mean of each column in the transformed DataFrame, rounded to 2 decimal places
mean=round(MD.mean(),2)
mean   # Output the mean values
```

```
<ipython-input-8-9c9ce4da3744>:2: FutureWarning: Downcasting behavior in `replace` is deprecated and will be removed in a future version. To retain the old behavior, explicitl
  MD=data.iloc[:,0:11].replace("Yes",1).replace("No",0)
```

	0
yummy	0.55
convenient	0.91
spicy	0.09
fattening	0.87
greasy	0.53
fast	0.90
cheap	0.60
tasty	0.64
expensive	0.36
healthy	0.20
disgusting	0.24

dtype: float64

```
# Initializing PCA (Principal Component Analysis) object
pca = PCA()
# Applying PCA on the MD data and transforming it to a new space with fewer dimensions
MD_pca=pca.fit_transform(MD)
# Fitting PCA on the MD data to learn the principal components (without transforming the data yet)
MD_p=pca.fit(MD)

# Calculating the standard deviation for each principal component
SD=np.sqrt(pca.explained_variance_)
# Calculating the proportion of variance explained by each principal component
PV=pca.explained_variance_ratio_
# Creating a list of principal component names (e.g., 'PC1', 'PC2', etc.)
index=[]
for i in range(len(SD)):
    i=i+1
    index.append("PC{}".format(i))

# Creating a DataFrame to summarize standard deviations, proportion of variance, and cumulative variance
sum=pd.DataFrame({
    "Standard deviation":SD,"Proportion of Variance":PV,"Cumulative Proportion":PV.cumsum()
},index=index)
sum   #Display the summary DataFrame
```

	Standard deviation	Proportion of Variance	Cumulative Proportion
PC1	0.757050	0.299447	0.299447
PC2	0.607456	0.192797	0.492244
PC3	0.504619	0.133045	0.625290
PC4	0.398799	0.083096	0.708386
PC5	0.337405	0.059481	0.767866
PC6	0.310275	0.050300	0.818166
PC7	0.289697	0.043849	0.862015
PC8	0.275122	0.039548	0.901563
PC9	0.265251	0.036761	0.938323
PC10	0.248842	0.032353	0.970677
PC11	0.236903	0.029323	1.000000

```
# Printing the standard deviation of each principal component, rounded to 1 decimal place
print("Standard Deviation:\n",SD.round(1))

# Getting the loadings (principal component coefficients)
load = (pca.components_)
# Extracting the rotation matrix (principal components in the original space)
i=0
rot_matrix = MD_p.components_.T

# Creating a DataFrame to display the rotation matrix, with column names as principal components
rot_df = pd.DataFrame(rot_matrix, index=MD.columns.values, columns=index)
# Rounding the rotation matrix values to 3 decimal places and flipping the signs
rot_df=round(-rot_df,3)
rot_df   # Display the rotated matrix DataFrame
```

```
Standard Deviation:
 [0.8 0.6 0.5 0.4 0.3 0.3 0.3 0.3 0.3 0.2 0.2]
```

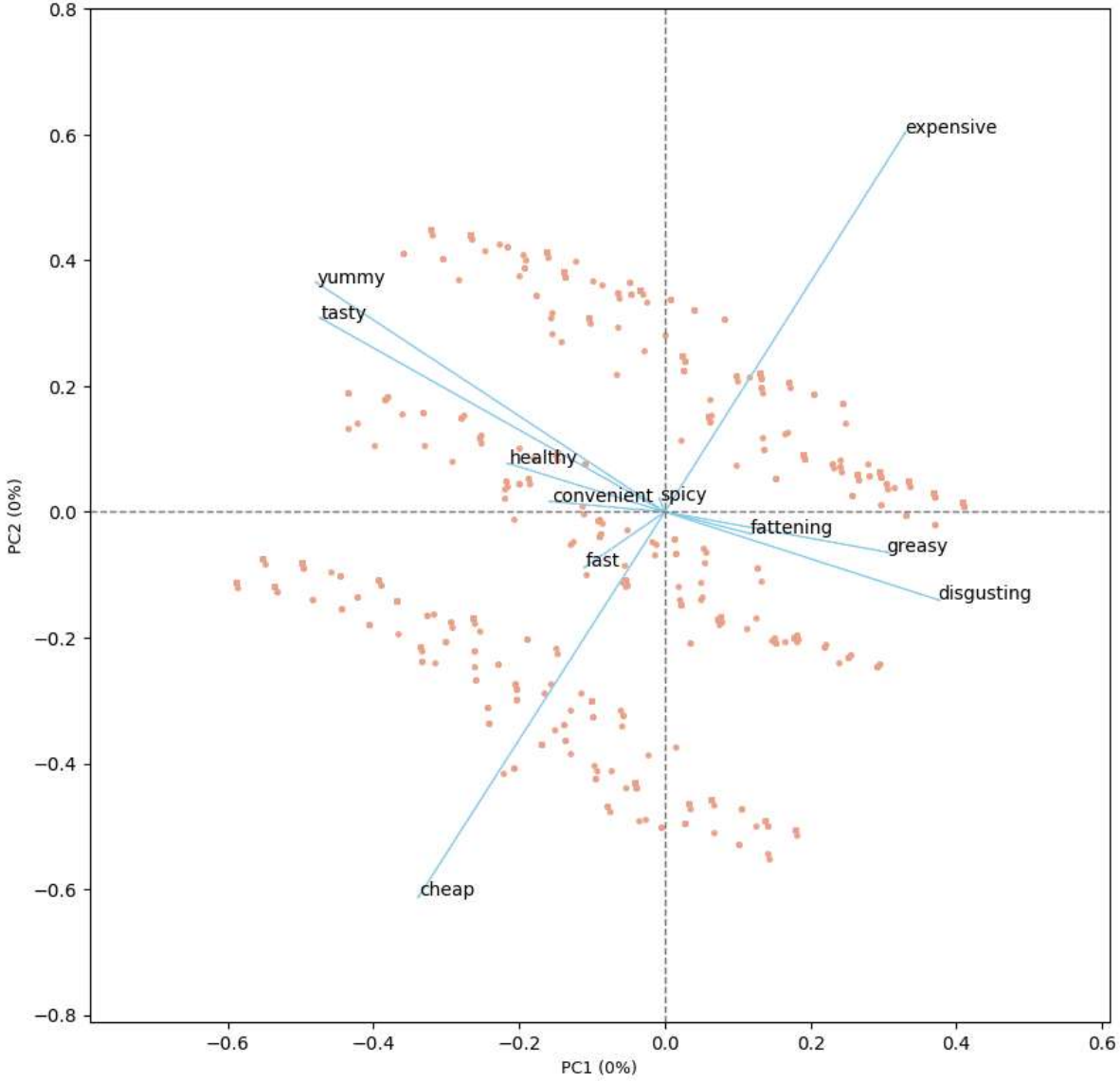
	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11
yummy	-0.477	0.364	-0.304	-0.055	-0.308	0.171	0.281	0.013	0.572	-0.110	0.045
convenient	-0.155	0.016	-0.063	0.142	0.278	-0.348	0.060	-0.113	-0.018	-0.666	-0.542
spicy	-0.006	0.019	-0.037	-0.198	0.071	-0.355	-0.708	0.376	0.400	-0.076	0.142
fattening	0.116	-0.034	-0.322	0.354	-0.073	-0.407	0.386	0.590	-0.161	-0.005	0.251
greasy	0.304	-0.064	-0.802	-0.254	0.361	0.209	-0.036	-0.138	-0.003	0.009	0.002
fast	-0.108	-0.087	-0.065	0.097	0.108	-0.595	0.087	-0.628	0.166	0.240	0.339
cheap	-0.337	-0.611	-0.149	-0.119	-0.129	-0.103	0.040	0.140	0.076	0.428	-0.489
tasty	-0.472	0.307	-0.287	0.003	-0.211	-0.077	-0.360	-0.073	-0.639	0.079	0.020
expensive	0.329	0.601	0.024	-0.068	-0.003	-0.261	0.068	0.030	0.067	0.454	-0.490
healthy	-0.214	0.077	0.192	-0.763	0.288	-0.178	0.350	0.176	-0.186	-0.038	0.158
disgusting	0.375	-0.140	-0.089	-0.370	-0.729	-0.211	0.027	-0.167	-0.072	-0.290	-0.041

```
# Displaying the rotation matrix (loadings of original features for each principal component)
rot_df
```

	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	PC9	PC10	PC11
yummy	-0.477	0.364	-0.304	-0.055	-0.308	0.171	0.281	0.013	0.572	-0.110	0.045
convenient	-0.155	0.016	-0.063	0.142	0.278	-0.348	0.060	-0.113	-0.018	-0.666	-0.542
spicy	-0.006	0.019	-0.037	-0.198	0.071	-0.355	-0.708	0.376	0.400	-0.076	0.142
fattening	0.116	-0.034	-0.322	0.354	-0.073	-0.407	0.386	0.590	-0.161	-0.005	0.251
greasy	0.304	-0.064	-0.802	-0.254	0.361	0.209	-0.036	-0.138	-0.003	0.009	0.002
fast	-0.108	-0.087	-0.065	0.097	0.108	-0.595	0.087	-0.628	0.166	0.240	0.339
cheap	-0.337	-0.611	-0.149	-0.119	-0.129	-0.103	0.040	0.140	0.076	0.428	-0.489
tasty	-0.472	0.307	-0.287	0.003	-0.211	-0.077	-0.360	-0.073	-0.639	0.079	0.020
expensive	0.329	0.601	0.024	-0.068	-0.003	-0.261	0.068	0.030	0.067	0.454	-0.490
healthy	-0.214	0.077	0.192	-0.763	0.288	-0.178	0.350	0.176	-0.186	-0.038	0.158
disgusting	0.375	-0.140	-0.089	-0.370	-0.729	-0.211	0.027	-0.167	-0.072	-0.290	-0.041

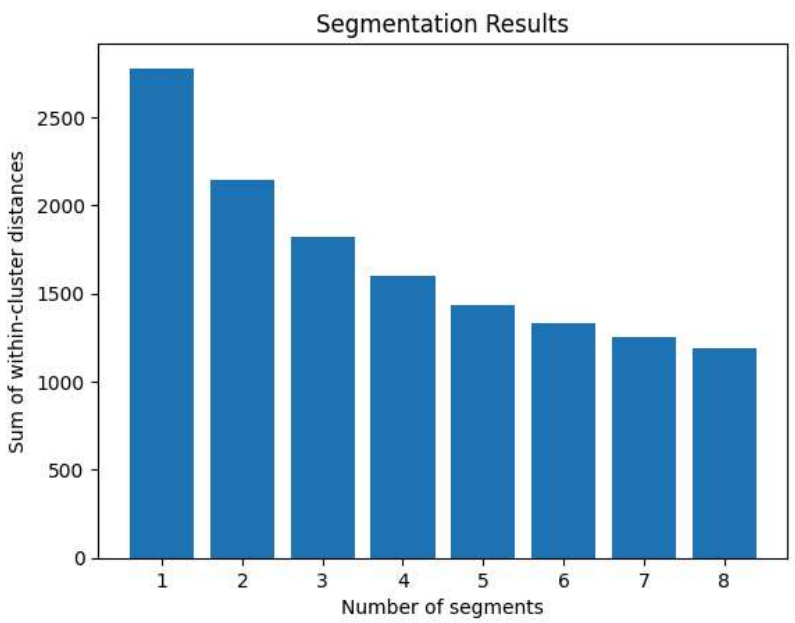
```
cluster.biplot(cscore=MD_pca, loadings=-load, labels=data.columns.values,var1=0,var2=0, show=True, dim=(10, 10))
```

```
WARNING:matplotlib.font_manager:findfont: Font family 'Arial' not found.
WARNING:matplotlib.font_manager:findfont: Font family 'Arial' not found.
WARNING:matplotlib.font_manager:findfont: Font family 'Arial' not found.
WARNING:matplotlib.font_manager:findfont: Font family 'Arial' not found.
WARNING:matplotlib.font_manager:findfont: Font family 'Arial' not found.
WARNING:matplotlib.font_manager:findfont: Font family 'Arial' not found.
WARNING:matplotlib.font_manager:findfont: Font family 'Arial' not found.
WARNING:matplotlib.font_manager:findfont: Font family 'Arial' not found.
```




```
np.random.seed(1234) # Set a random seed for reproducibility
nrep = 10 # Number of initializations to run for each KMeans clustering
```

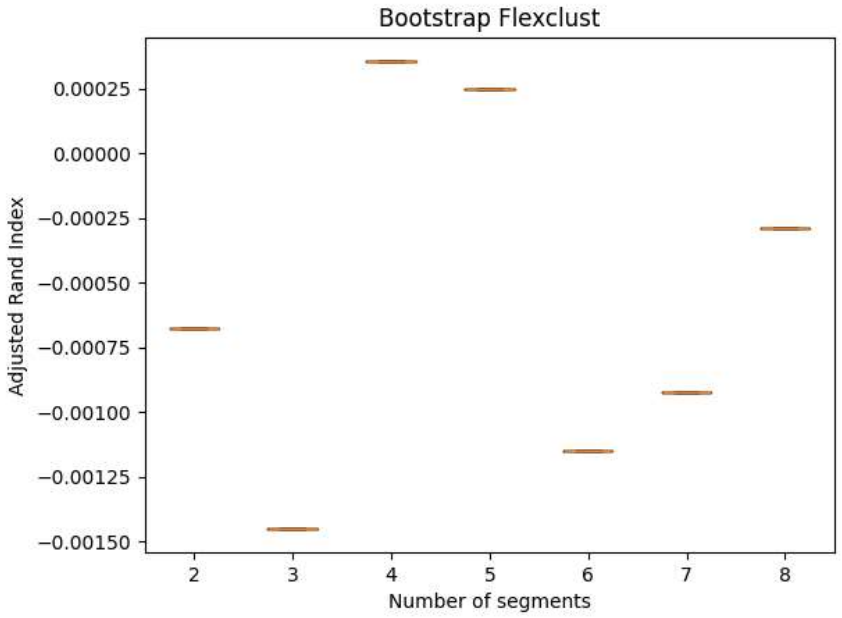
```
num_segments = range(1, 9) # Define the range of cluster numbers (from 1 to 8)
within_cluster_distances = [] # List to store the within-cluster sum of distances (inertia) for each k
MD_km28 = {} # Dictionary to store KMeans models for each cluster count
# Loop through the different numbers of clusters (from 1 to 8)
for k in num_segments:     # Create a KMeans model with 'k' clusters and specified initializations
    kmeans = KMeans(n_clusters=k, n_init=nrep, random_state=1234)
    kmeans.fit(MD)     # Fit the KMeans model to the data (MD)
    within_cluster_distances.append((kmeans.inertia_))     # Append the inertia (sum of squared distances) for the current k value
    MD_km28[str(k)] = kmeans     # Store the trained KMeans model in the dictionary using 'k' as the key

plt.bar(num_segments, within_cluster_distances) # Plot the within-cluster sum of squares (inertia) for each cluster count (k)
plt.xlabel("Number of segments") # Label the x-axis (number of clusters)
plt.ylabel("Sum of within-cluster distances") # Label the y-axis (within-cluster sum of distances)
plt.title("Segmentation Results") # Title of the plot
plt.show() # Display the plot
```




```
# Set random seed for reproducibility
np.random.seed(1234)

nboot = 100  # Number of bootstrap resamples
nrep = 10    # Number of initializations for each KMeans clustering

# Create bootstrap samples from the original dataset
bootstrap_samples = []
for _ in range(nboot):
    # Resample the dataset with replacement
    bootstrap_sample = resample(MD.values, random_state=1234)
    bootstrap_samples.append(bootstrap_sample)
```

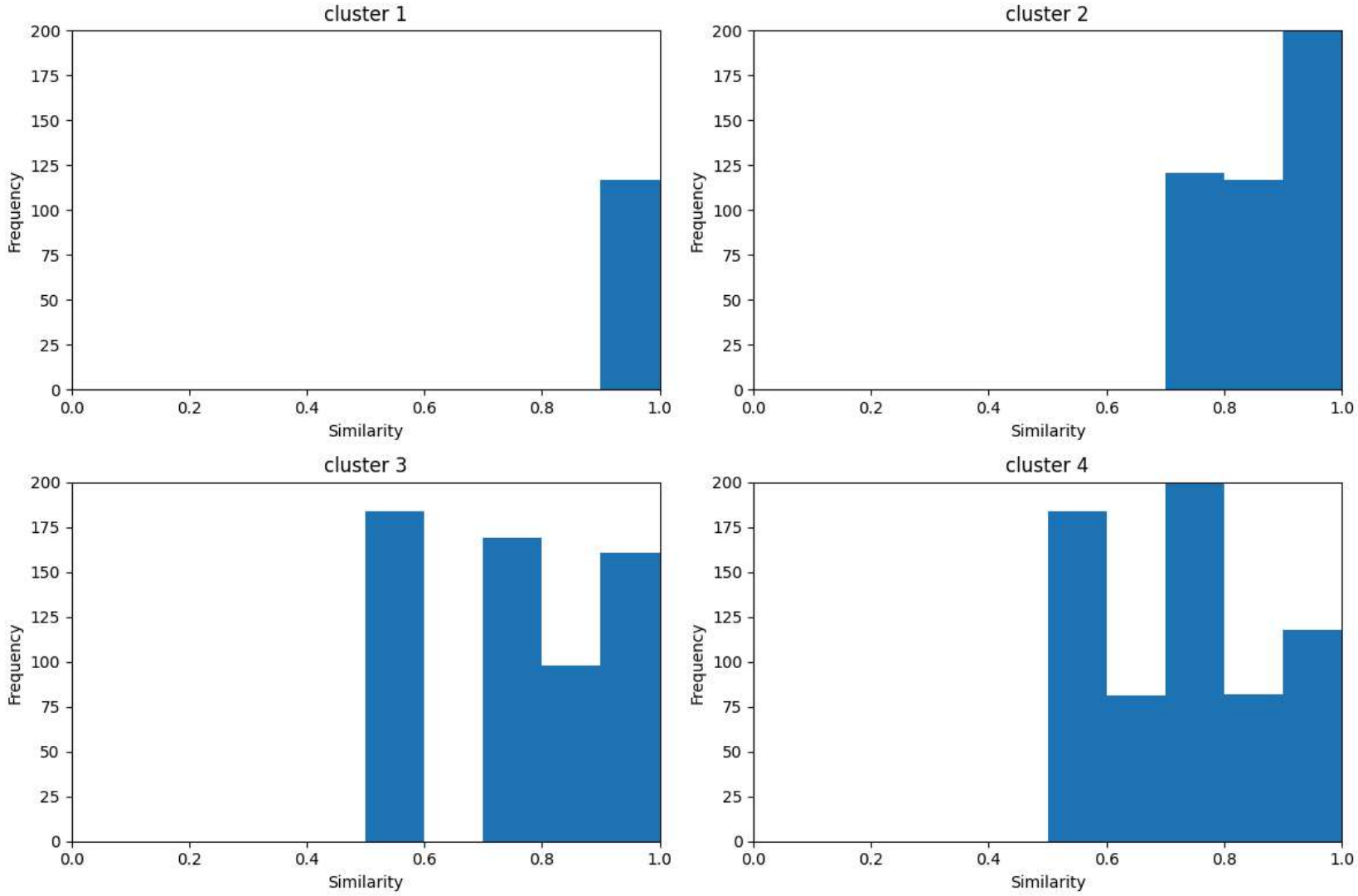
```
# List to store adjusted Rand index scores
adjusted_rand_index = []

# Define the range of cluster counts to evaluate (from 2 to 8)
num_segments = range(2, 9)

# For each number of clusters (k), compute stability scores
for k in num_segments:
    stability_scores = []
    for bootstrap_sample in bootstrap_samples:
        # Fit KMeans on the bootstrap sample
        kmeans = KMeans(n_clusters=k, n_init=nrep, random_state=1234)
        kmeans.fit(bootstrap_sample)

        # Predict labels for both the bootstrap sample and the original data
        cluster_labels = kmeans.predict(bootstrap_sample)
        true_labels = kmeans.predict(MD.values)

        # Compute Adjusted Rand Index to measure clustering similarity
        stability_score = adjusted_rand_score(true_labels, cluster_labels)
        stability_scores.append(stability_score)

    # Store stability scores for the current k
    adjusted_rand_index.append(stability_scores)

# Convert list to NumPy array and transpose to prepare for boxplot
adjusted_rand_index = np.array(adjusted_rand_index).T

# Create a boxplot to visualize clustering stability for different k values
plt.boxplot(adjusted_rand_index, labels=num_segments, whis=10)
plt.xlabel("Number of segments")
plt.ylabel("Adjusted Rand Index")
plt.title("Bootstrap Flexclust")
plt.show()
```

```
<ipython-input-14-4392ca741576>:27: MatplotlibDeprecationWarning: The 'labels' parameter of boxplot() has been renamed 'tick_labels' since Matplotlib 3.9; support for the old
  plt.boxplot(adjusted_rand_index, labels=num_segments, whis=10)
```

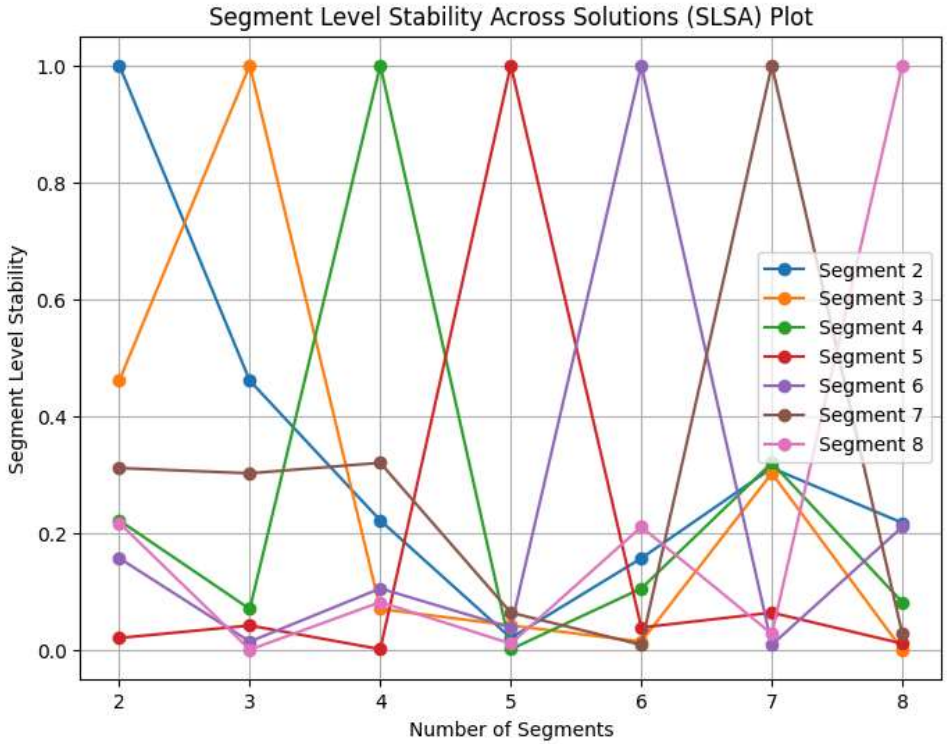



```
# Define the histogram range, number of bins, and max y-axis frequency
range_values = (0, 1)     # Similarity scores range from 0 to 1
num_bins = 10             # Number of bins in the histogram
max_frequency = 200       # Maximum frequency shown on the y-axis

# Create a 2x2 grid of subplots
fig, axs = plt.subplots(2, 2, figsize=(12, 8))

# Loop through cluster counts 1 to 4
for i in range(1, 5):
    # Predict cluster labels using the saved KMeans model
    labels = MD_km28[str(i)].predict(MD)

    # Compute similarity scores: distance to the nearest cluster center
    similarities = MD_km28[str(i)].transform(MD).min(axis=1)

    # Determine subplot location
    row = (i - 1) // 2
    col = (i - 1) % 2

    # Plot histogram of similarities
    axs[row, col].hist(similarities, bins=num_bins, range=range_values)
    axs[row, col].set_xlabel('Similarity')
    axs[row, col].set_ylabel('Frequency')
    axs[row, col].set_title('Cluster {}'.format(i))

    # Set consistent axes limits for comparison across subplots
```

```
        axs[row, col].set_xlim(range_values)
        axs[row, col].set_ylim(0, max_frequency)

        # Set specific x-tick marks for readability
        axs[row, col].set_xticks([0, 0.2, 0.4, 0.6, 0.8, 1.0])

# Adjust layout to avoid overlap
plt.tight_layout()

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

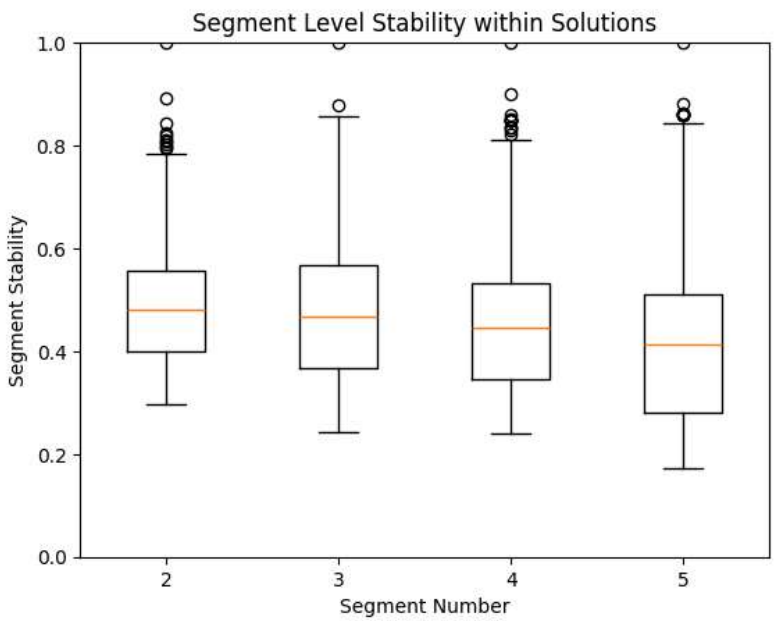


```
# Define the range of segment counts to evaluate (from 2 to 8)
num_segments = range(2, 9)

# List to store predicted labels for each segment solution
segment_stability = []

# Predict cluster labels for each segment count and store
for segment in num_segments:
    labels_segment = MD_km28[str(segment)].predict(MD)
    segment_stability.append(labels_segment)

# Create the plot
plt.figure(figsize=(8, 6))

# For each segment solution, calculate and plot similarity with other solutions
for i, segment in enumerate(num_segments):
    # Compare the i-th segmentation against all others
    stability_scores = [np.mean(segment_stability[i] == labels) for labels in segment_stability]

    # Plot the stability scores as a line
    plt.plot(num_segments, stability_scores, marker='o', label=f'Segment {segment}')

# Add plot labels and formatting
plt.xlabel('Number of Segments')
plt.ylabel('Segment Level Stability')
plt.title('Segment Level Stability Across Solutions (SLSA) Plot')
plt.xticks(num_segments)
plt.legend()
plt.grid(True)

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




```
# Define the segment solutions to analyze
segment_solutions = ["2", "3", "4", "5"]

# Dictionaries to hold cluster labels and similarity values for each solution
segment_labels = {}
segment_similarities = {}

# Step 1: For each solution, compute predicted labels and similarity (distance to assigned cluster center)
for segment in segment_solutions:
    segment_labels[segment] = MD_km28[segment].predict(MD)
    segment_similarities[segment] = MD_km28[segment].transform(MD).min(axis=1)

# Step 2: Normalize the similarity scores and store them
segment_stability_values = []
for segment in segment_solutions:
    similarities = segment_similarities[segment]

    # Normalize similarities to [0, 1] scale for comparability
    normalized_similarities = similarities / np.max(similarities)
    segment_stability_values.append(normalized_similarities)

# Step 3: Create a boxplot to visualize segment stability distribution for each segmentation solution
plt.boxplot(segment_stability_values, whis=1.5)
plt.xlabel("Segment Number")
plt.ylabel("Segment Stability")
```

```
plt.xticks(range(1, len(segment_solutions) + 1), segment_solutions)
plt.ylim(0, 1)
plt.title("Segment Level Stability within Solutions")
plt.show()
```

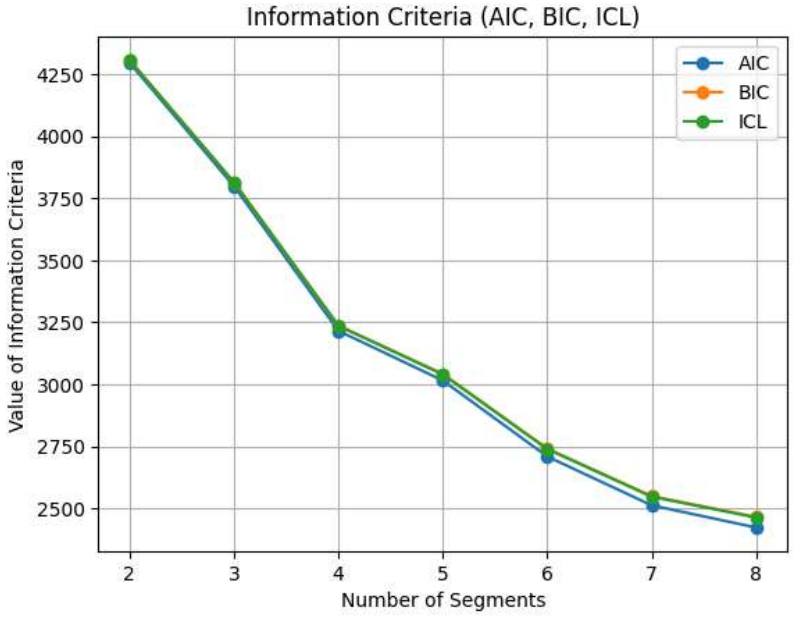



```
# Set seed for reproducibility
np.random.seed(1234)

# Define the range of cluster counts to evaluate
k_values = range(2, 9)

# Store model evaluation metrics
MD_m28 = []

# Loop over each k to fit KMeans and calculate model selection criteria
for k in k_values:
    model = KMeans(n_clusters=k, random_state=1234)
    model.fit(MD.values)  # Fit model on data

    # Gather model details and compute metrics
    iter_val = model.n_iter_  # Number of iterations until convergence
    converged = True          # Always True for scikit-learn KMeans if max_iter not hit
    k_val = k                 # Number of clusters
    k0_val = k                # Placeholder; often used in hierarchical models
    log_likelihood = -model.inertia_  # Use negative inertia as a proxy for log-likelihood

    n_samples, _ = MD.shape
    aic = -2 * log_likelihood + 2 * k               # AIC: model fit vs complexity
    bic = -2 * log_likelihood + np.log(n_samples) * k  # BIC: penalizes complexity more
    labels = model.labels_
```

```
        # Calculate entropy (uncertainty) of the cluster assignments
        counts = np.bincount(labels)
        probs = counts / float(counts.sum())
        class_entropy = entropy(probs)

        # ICL (Integrated Completed Likelihood): BIC adjusted by entropy
        icl = bic - class_entropy

        # Store all results in list
        MD_m28.append((iter_val, converged, k_val, k0_val, log_likelihood, aic, bic, icl))

# Convert results to DataFrame
MD_m28 = pd.DataFrame(MD_m28, columns=['iter', 'converged', 'k', 'k0', 'logLik', 'AIC', 'BIC', 'ICL'])

# Display model evaluation table
print(MD_m28)
```

```
   iter  converged  k  k0       logLik          AIC          BIC          ICL
0    10       True  2   2 -2146.062044  4296.124088  4306.686859  4306.015908
1     5       True  3   3 -1896.330266  3798.660532  3814.504689  3813.529671
2     9       True  4   4 -1603.913802  3215.827604  3236.953147  3235.627738
3     9       True  5   5 -1502.697153  3015.394306  3041.801234  3040.267284
4     7       True  6   6 -1348.665399  2709.330799  2741.019113  2739.277954
5    10       True  7   7 -1249.233890  2512.467780  2549.437480  2547.530062
6     9       True  8   8 -1203.646165  2423.292330  2465.543415  2463.533662
```

```
# Extract relevant columns from the results DataFrame
num_segments = MD_m28["k"]
AIC_values = MD_m28["AIC"]
BIC_values = MD_m28["BIC"]
ICL_values = MD_m28["ICL"]

# Plot AIC, BIC, and ICL values against number of segments
plt.plot(num_segments, AIC_values, marker='o', label='AIC')
plt.plot(num_segments, BIC_values, marker='o', label='BIC')
plt.plot(num_segments, ICL_values, marker='o', label='ICL')

# Add labels, title, legend, and grid for readability
plt.xlabel('Number of Segments')
plt.ylabel('Value of Information Criteria')
plt.title('Information Criteria (AIC, BIC, ICL)')
plt.legend()
plt.grid(True)

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




```
from sklearn.mixture import GaussianMixture

k = 4  # Number of clusters

# Step 1: Fit KMeans clustering on the full dataset
kmeans = KMeans(n_clusters=k, random_state=1234)
kmeans.fit(MD)
kmeans_clusters = kmeans.predict(MD)

# Step 2: Fit Gaussian Mixture Model (GMM) on the same dataset
gmm = GaussianMixture(n_components=k, random_state=1234)
gmm.fit(MD)
gmm_clusters = gmm.predict(MD)

# Step 3: Create a DataFrame comparing KMeans and GMM cluster assignments
results = pd.DataFrame({'kmeans': kmeans_clusters, 'mixture': gmm_clusters})

# Step 4: Subset the data where GMM assigns cluster label 3
MD_m4 = MD[results['mixture'] == 3]

# Step 5: Apply KMeans again within GMM cluster 3 to explore sub-segmentation
k4_m4 = KMeans(n_clusters=k, random_state=1234)
k4_m4.fit(MD_m4)
k4_m4_clusters = k4_m4.predict(MD_m4)

# Step 6: Store the new KMeans cluster assignments within GMM cluster 3
results_m4 = pd.DataFrame({'kmeans': k4_m4_clusters, 'mixture': 3})

# Step 7: Print a cross-tab to compare initial KMeans and GMM clusters
print(pd.crosstab(results['kmeans'], results['mixture']))
```

```
# Step 8: Print a cross-tab to compare initial KMeans clusters and sub-clusters within GMM cluster 3
print(pd.crosstab(results['kmeans'], results_m4['kmeans']))
```

```
mixture    0    1    2    3
kmeans
0        546    0    1   33
1          0  213   11    4
2         46    3  265    8
3         29   38    0  256
kmeans   0   1   2   3
kmeans
0       51  15  35  20
1       18   8  11  11
2       23   9  14  22
3       29   5  15  15
```

```
from sklearn.mixture import GaussianMixture
import numpy as np

# Fit Gaussian Mixture Model with 4 components to the dataset MD
gmm_m4a = GaussianMixture(n_components=4)
gmm_m4a.fit(MD)

# Compute average log-likelihood of the data under the fitted model
log_likelihood_m4a = gmm_m4a.score(MD)

# Fit another GMM with the same number of components (redundant, identical to above unless models are intentionally separate)
gmm_m4 = GaussianMixture(n_components=4)
gmm_m4.fit(MD)

# Compute average log-likelihood for the second model
log_likelihood_m4 = gmm_m4.score(MD)

# Print the log-likelihoods of both models
print("Log-likelihood for MD.m4a:", log_likelihood_m4a)
print("Log-likelihood for MD.m4:", log_likelihood_m4)
```

```
Log-likelihood for MD.m4a: 9.456781748071887
Log-likelihood for MD.m4: 3.6427464011765824
```

```
like_counts = pd.value_counts(data['Like']) # Count the occurrences of each unique value in the 'Like' column
# (e.g., 'I HATE IT!-5', '-4', ..., 'I LOVE IT!+5')
reversed_counts = like_counts.iloc[::-1] # Reverse the order of the counts (useful for custom display or visualization)
# Print the reversed count of each response
print(reversed_counts)
```

```
Like
-1              58
-2              59
-4              71
-3              73
I love it!+5    143
I hate it!-5    152
1               152
4               160
```

```
0              169
2              187
3              229
Name: count, dtype: int64
<ipython-input-22-d525372f2881>:1: FutureWarning: pandas.value_counts is deprecated and will be removed in a future version. Use pd.Series(obj).value_counts() instead.
  like_counts = pd.value_counts(data['Like'])
```

```
# Define a mapping of string values to corresponding numeric sentiment scores
like_mapping = {
    'I HATE IT!-5': -5,
    '-4': -4,
    '-3': -3,
    '-2': -2,
    '-1': -1,
    '0': 0,
    '1': 1,
    '2': 2,
    '3': 3,
    '4': 4,
    'I LOVE IT!+5': 5
}

# Apply the mapping to the 'Like' column and store the result in a new column 'Like.n'
data['Like.n'] = data['Like'].map(like_mapping)

# Count the occurrences of each numeric rating
like_n_counts = data['Like.n'].value_counts()

# Print the frequency of each score
print(like_n_counts)
```

```
Like.n
 3.0    229
 2.0    187
 0.0    169
 4.0    160
 1.0    152
-3.0     73
-4.0     71
-2.0     59
-1.0     58
Name: count, dtype: int64
```

```
from patsy import dmatrices

# Selecting the first 11 columns of the dataset as independent variables
independent_vars = data.columns[0:11]

# Creating a formula string: "var1 + var2 + ... + var11"
formula_str = ' + '.join(independent_vars)

# Constructing the full formula: "Like ~ var1 + var2 + ... + var11"
formula_str = 'Like ~ ' + formula_str

# Using patsy's dmatrices to generate design matrices
```

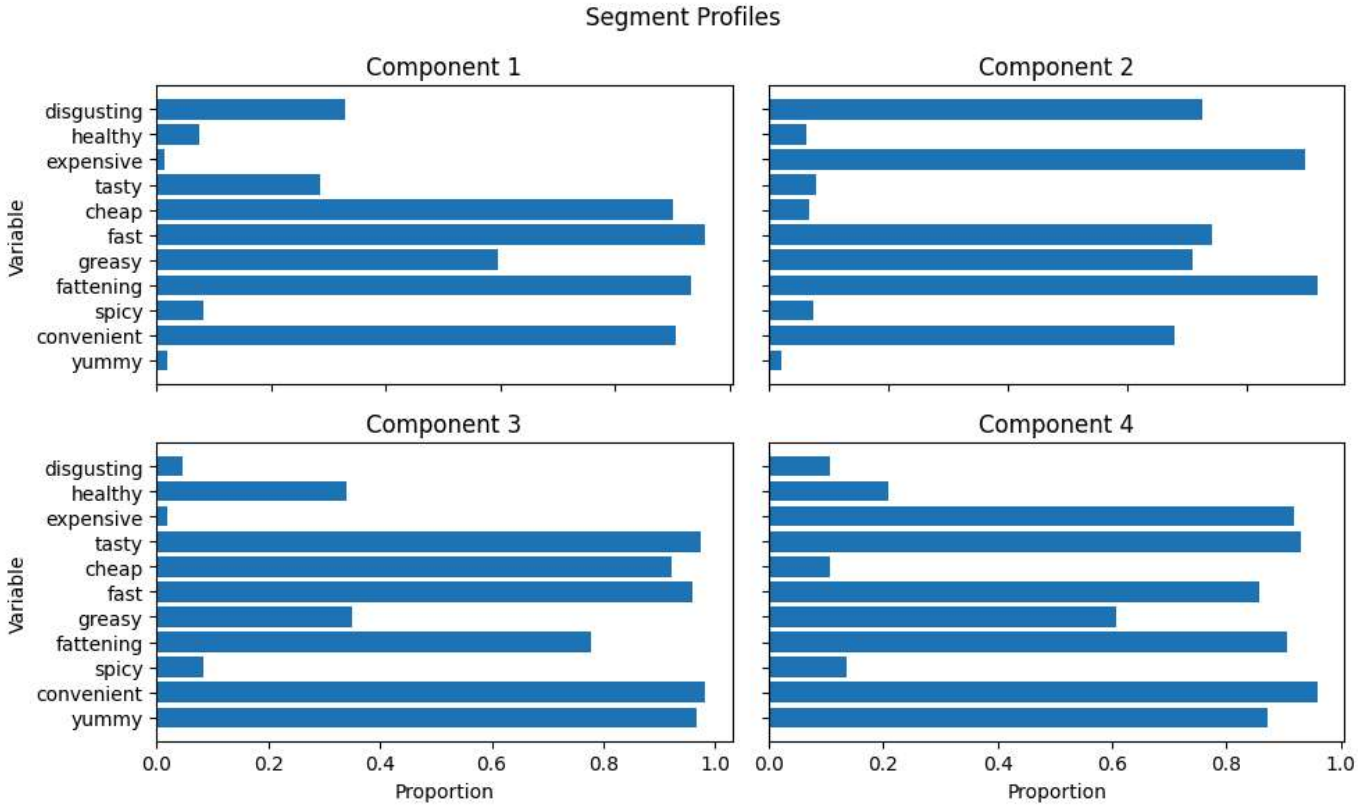
```
# dmatrices returns a tuple (y, X), where y is the dependent variable matrix, X is the design matrix
f = dmatrices(formula_str, data=data)[1]  # Extracting the independent variable matrix (X)

# Printing the design matrix
print(f)
```

```
[[1. 0. 1. ... 1. 0. 0.]
 [1. 1. 1. ... 1. 0. 0.]
 [1. 0. 1. ... 1. 1. 0.]
 ...
 [1. 1. 1. ... 1. 0. 0.]
 [1. 1. 1. ... 0. 1. 0.]
 [1. 0. 1. ... 1. 0. 1.]]
```

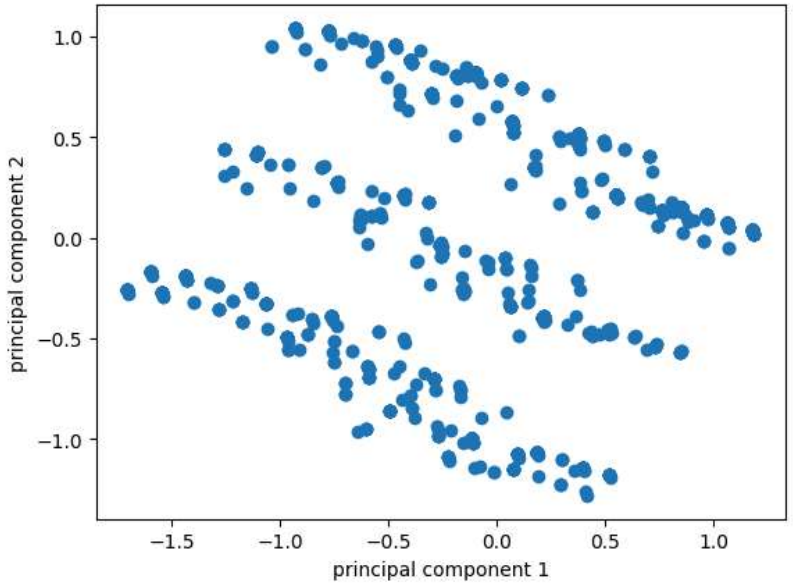
```
from sklearn.mixture import GaussianMixture  # For Gaussian Mixture Model clustering
from patsy import dmatrix  # For building design matrices using formulas
import numpy as np  # For numerical operations

np.random.seed(1234)  # Setting a random seed for reproducibility

# Creating the design matrix X using the formula and data (assumes 'f' is a formula object)
X = dmatrix(f.design_info, data=data)

# Creating the response matrix y from the 'Like' column in the data
y = dmatrix('Like', data=data)

# Model parameters
n_components = 2  # Number of clusters/components
n_init = 10       # Number of initializations to try
verbose = False   # Disable verbose output
n_rep = 10        # Unused in this snippet (possibly meant for looping or cross-validation)

# Fitting the Gaussian Mixture Model to the data
model = GaussianMixture(n_components=n_components, n_init=n_init, verbose=verbose)
MD_reg2 = model.fit(X, y)  # Note: GMM does not take 'y'; this might raise an error

# Output the fitted model summary (this will be a GMM object)
print(MD_reg2)

# Predict cluster assignments and count the number of points in each cluster
cluster_sizes = np.bincount(model.predict(X))

# Print out the size of each cluster
print("Cluster sizes:")
for i, size in enumerate(cluster_sizes):
    print(f"{i+1}: {size}")
```

```
GaussianMixture(n_components=2, n_init=10, verbose=False)
Cluster sizes:
1: 985
2: 468
```

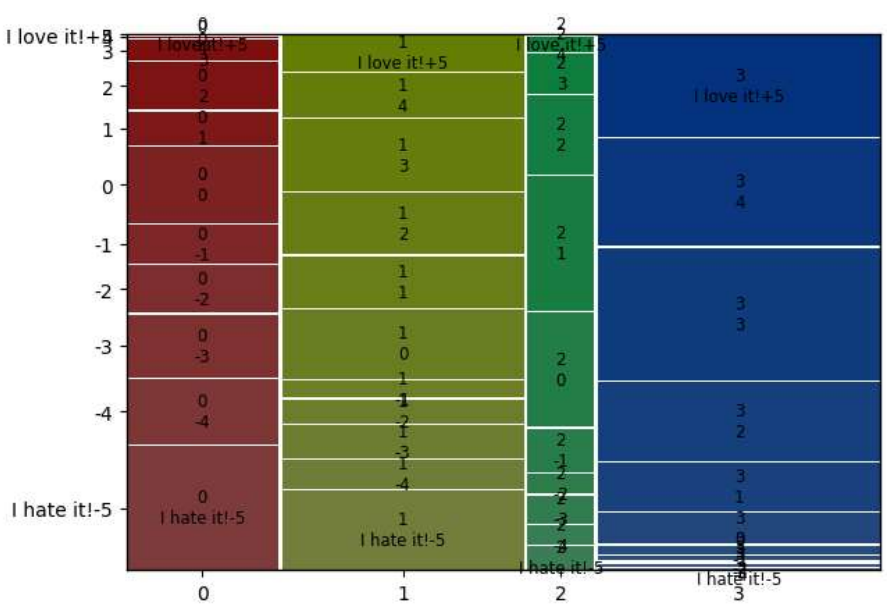
```
import pandas as pd  # Importing pandas for data manipulation
import matplotlib.pyplot as plt  # Importing matplotlib for data visualization

# Selecting the KMeans model labeled '4' from the MD_km28 dictionary
kmeans = MD_km28['4']

# Extracting the cluster labels assigned to each data point
labels = kmeans.labels_

# Grouping the original data (MD) by cluster labels and computing the mean for each cluster
MD_mean = MD.groupby(labels).mean()

# Creating a 2x2 grid of subplots to visualize the average profiles of the 4 clusters
fig, axs = plt.subplots(2, 2, figsize=(10, 6))

# Plotting the mean values of variables for each cluster as horizontal bar charts
axs[0, 0].barh(range(MD_mean.shape[1]), MD_mean.iloc[0])
axs[0, 0].set_title('Component 1')

axs[0, 1].barh(range(MD_mean.shape[1]), MD_mean.iloc[1])
axs[0, 1].set_title('Component 2')

axs[1, 0].barh(range(MD_mean.shape[1]), MD_mean.iloc[2])
axs[1, 0].set_title('Component 3')

axs[1, 1].barh(range(MD_mean.shape[1]), MD_mean.iloc[3])
axs[1, 1].set_title('Component 4')

# Customizing all subplots: setting x/y labels and y-tick labels to variable names
for ax in axs.flat:
    ax.set(ylabel='Variable', xlabel='Proportion')
    ax.set_yticks(range(MD_mean.shape[1]))
    ax.set_yticklabels(MD.columns)

# Removing inner axis labels for a cleaner look
for ax in axs.flat:
    ax.label_outer()

# Adding a common title for the entire figure
fig.suptitle('Segment Profiles')

# Adjusting layout to prevent overlapping
fig.tight_layout()

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




```
from sklearn.cluster import KMeans # Importing KMeans clustering algorithm from sklearn

from sklearn.decomposition import PCA # Importing PCA (Principal Component Analysis) for dimensionality reduction
import matplotlib.pyplot as plt # Importing matplotlib for plotting
kmeans = KMeans(n_clusters=4) # Creating a KMeans model with 4 clusters
kmeans.fit(MD) # Fitting the KMeans model on the dataset 'MD'
pca = PCA(n_components=2) # Initializing PCA to reduce data to 2 principal components
MD_pca = pca.fit_transform(MD) # Applying PCA on 'MD' to transform it into 2D data for visualization
fig, ax = plt.subplots() # Creating a new matplotlib figure and axis
ax.scatter(MD_pca[:, 0], MD_pca[:, 1]) # Creating a scatter plot using the two principal components
ax.set_xlabel('principal component 1') # Labeling the x-axis as the first principal component
ax.set_ylabel('principal component 2') # Labeling the y-axis as the second principal component
plt.show() # Displaying the scatter plot
```




```
from statsmodels.graphics.mosaicplot import mosaic # Importing mosaic plot for visualizing categorical data
from itertools import product # Importing product (not used here, but often for generating combinations)
#Label encoding for categorical - Converting 11 cols with yes/no
from sklearn.preprocessing import LabelEncoder # Importing LabelEncoder to convert categorical text data into numbers
def labelling(x): # Defining a function to apply label encoding on a given column

    data1[x] = LabelEncoder().fit_transform(data1[x])
    return data1

cat = ['yummy', 'convenient', 'spicy', 'fattening', 'greasy', 'fast', 'cheap',

       'tasty', 'expensive', 'healthy', 'disgusting'] # List of 11 categorical features (yes/no) related to food opinions


for i in cat:

    labelling(i) # Applying label encoding to each column in the list
data1 # Displaying the modified DataFrame with encoded categorical values
df_eleven = data1.loc[:,cat] # Extracting the 11 encoded columns into a new DataFrame
df_eleven # Displaying the new DataFrame containing only the 11 opinion features
kmeans = KMeans(n_clusters=4, init='k-means++', random_state=0).fit(df_eleven) # Applying KMeans clustering with 4 clusters on the 11 opinion features
data1['cluster_num'] = kmeans.labels_ # Applying KMeans clustering with 4 clusters on the 11 opinion features
crosstab =pd.crosstab(data1['cluster_num'],data1['Like']) # Creating a crosstab of clusters vs. Like responses for analysis
#Reordering cols
data1 # Displaying the updated DataFrame with cluster assignments
crosstab = crosstab[['I hate it!-5','-4','-3','-2','-1','0','1','2','3','4','I love it!+5']] # Reordering the 'Like' columns in the crosstab for better visual arrangement
crosstab # Displaying the reordered crosstab
plt.rcParams['figure.figsize'] = (7,5) # Setting the default size for plots
mosaic(crosstab.stack()) # Creating a mosaic plot to show the distribution of 'Like' responses across clusters
plt.show()# Displaying the mosaic plot
```




```
from statsmodels.graphics.mosaicplot import mosaic # Importing the mosaic plot function from statsmodels for categorical data visualization
MD_k4=MD_km28['4'] # Extracting the clustering model for 4 clusters from the dictionary MD_km28
k4 = MD_k4.labels_ # Getting the cluster labels assigned by the k-means model for each data point
ct = pd.crosstab(k4, data['Gender']) # Creating a cross-tabulation table of cluster labels vs. gender counts
ct # Displaying the cross-tabulation table
mosaic(ct.stack(),gap=0.01) # Plotting a mosaic plot to visually compare gender distribution across clusters with minimal gap
plt.show() # Displaying the mosaic plot
```

```
df = pd.DataFrame({'Segment': k4, 'Age': data['Age']}) # Creating a new DataFrame with 'Segment' labels (from k-means result k4) and corresponding 'Age' values
df.boxplot(by='Segment', column='Age') # Creating a box-and-whisker plot of 'Age' grouped by 'Segment' to show age distribution across segments
plt.title('Parallel box-and-whisker plot of age by segment') # Setting the title for the plot
plt.suptitle('') # Removing the default subplot title to keep the plot clean
plt.show() # Displaying the box plot
```