In [63]:
```
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
```

In [64]:
```
dataset = pd.read_csv(r'C:\Users\WELCOME\Desktop\mcdonalds.csv')
```

In [65]:
```
dataset.head()
```

Out[65]:

	yummy	convenient	spicy	fattening	greasy	fast	cheap	tasty	expensive	healthy	disg
0	No	Yes	No	Yes	No	Yes	Yes	No	Yes	No	
1	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	
2	No	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	
3	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No	No	
4	No	Yes	No	Yes	Yes	Yes	Yes	No	No	Yes	

In [66]:
```
dataset.shape
```

Out[66]:
```
(1453, 15)
```

In [67]:
```
MD_x = dataset.iloc[:, 0:11].values
MD_x = (MD_x == "Yes").astype(int)
col_means = np.round(np.mean(MD_x, axis=0), 2)
```

In [68]:
```
col_means
```

Out[68]:
```
array([0.55, 0.91, 0.09, 0.87, 0.53, 0.9 , 0.6 , 0.64, 0.36, 0.2 , 0.24])
```

In [69]:
```
pca = PCA()
MD_pca = pca.fit_transform(MD_x)
```

In [70]:
```
std_deviation = pca.explained_variance_**0.5
print("Standard Deviation of Principal Components:")
print(std_deviation)

# Proportion of variance explained by each principal component
prop_variance = pca.explained_variance_ratio_
print("\nProportion of Variance Explained by Each Principal Component:")
print(prop_variance)

# Cumulative proportion of variance explained
cumulative_prop_variance = prop_variance.cumsum()
print("\nCumulative Proportion of Variance Explained:")
print(cumulative_prop_variance)
```

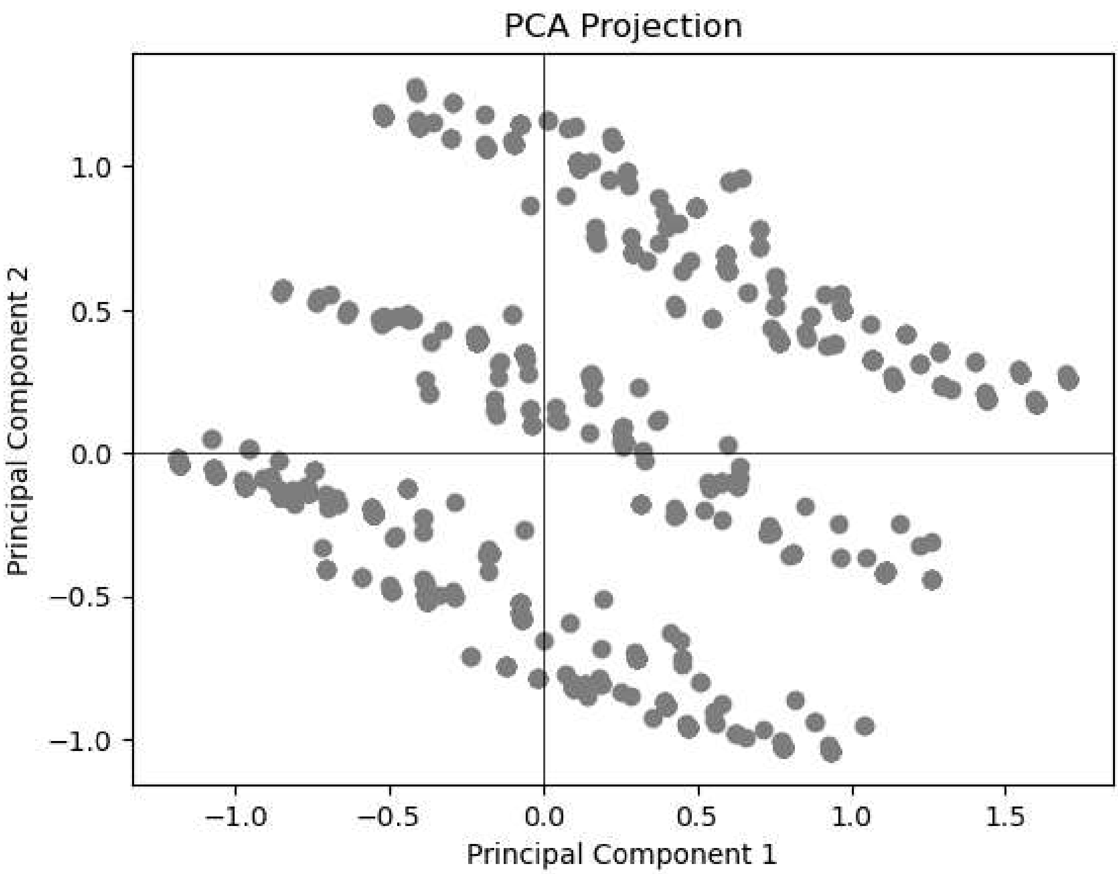
```
Standard Deviation of Principal Components:
[0.75704952 0.60745556 0.50461946 0.39879859 0.33740501 0.31027461
 0.28969732 0.27512196 0.2652511  0.24884182 0.23690284]

Proportion of Variance Explained by Each Principal Component:
[0.29944723 0.19279721 0.13304535 0.08309578 0.05948052 0.05029956
 0.0438491  0.03954779 0.0367609  0.03235329 0.02932326]

Cumulative Proportion of Variance Explained:
[0.29944723 0.49224445 0.6252898  0.70838558 0.7678661  0.81816566
 0.86201476 0.90156255 0.93832345 0.97067674 1.        ]
```

In [71]:

```
# Perform PCA
pca = PCA()
MD_pca = pca.fit_transform(MD_x)

# Get the words for your features
feature_names = ["yummy", "convenient", "spicy", "fattening", "greasy", "fa

# Create a DataFrame for better formatting
pca_components_df = pd.DataFrame(pca.components_, columns=feature_names)

# Print the DataFrame
print(pca_components_df.round(3))
```

```
     yummy  convenient  spicy  fattening  greasy   fast  cheap  tasty  \
0   -0.477      -0.155 -0.006      0.116   0.304 -0.108 -0.337 -0.472
1    0.364       0.016  0.019     -0.034  -0.064 -0.087 -0.611  0.307
2   -0.304      -0.063 -0.037     -0.322  -0.802 -0.065 -0.149 -0.287
3    0.055      -0.142  0.198     -0.354   0.254 -0.097  0.119 -0.003
4   -0.308       0.278  0.071     -0.073   0.361  0.108 -0.129 -0.211
5    0.171      -0.348 -0.355     -0.407   0.209 -0.595 -0.103 -0.077
6   -0.281      -0.060  0.708     -0.386   0.036 -0.087 -0.040  0.360
7    0.013      -0.113  0.376      0.590  -0.138 -0.628  0.140 -0.073
8    0.572      -0.018  0.400     -0.161  -0.003  0.166  0.076 -0.639
9   -0.110      -0.666 -0.076     -0.005   0.009  0.240  0.428  0.079
10   0.045      -0.542  0.142      0.251   0.002  0.339 -0.489  0.020

     expensive  healthy  disgusting
0        0.329   -0.214       0.375
1        0.601    0.077      -0.140
2        0.024    0.192      -0.089
3        0.068    0.763       0.370
4       -0.003    0.288      -0.729
5       -0.261   -0.178      -0.211
6       -0.068   -0.350      -0.027
7        0.030    0.176      -0.167
8        0.067   -0.186      -0.072
9        0.454   -0.038      -0.290
10      -0.490    0.158      -0.041
```

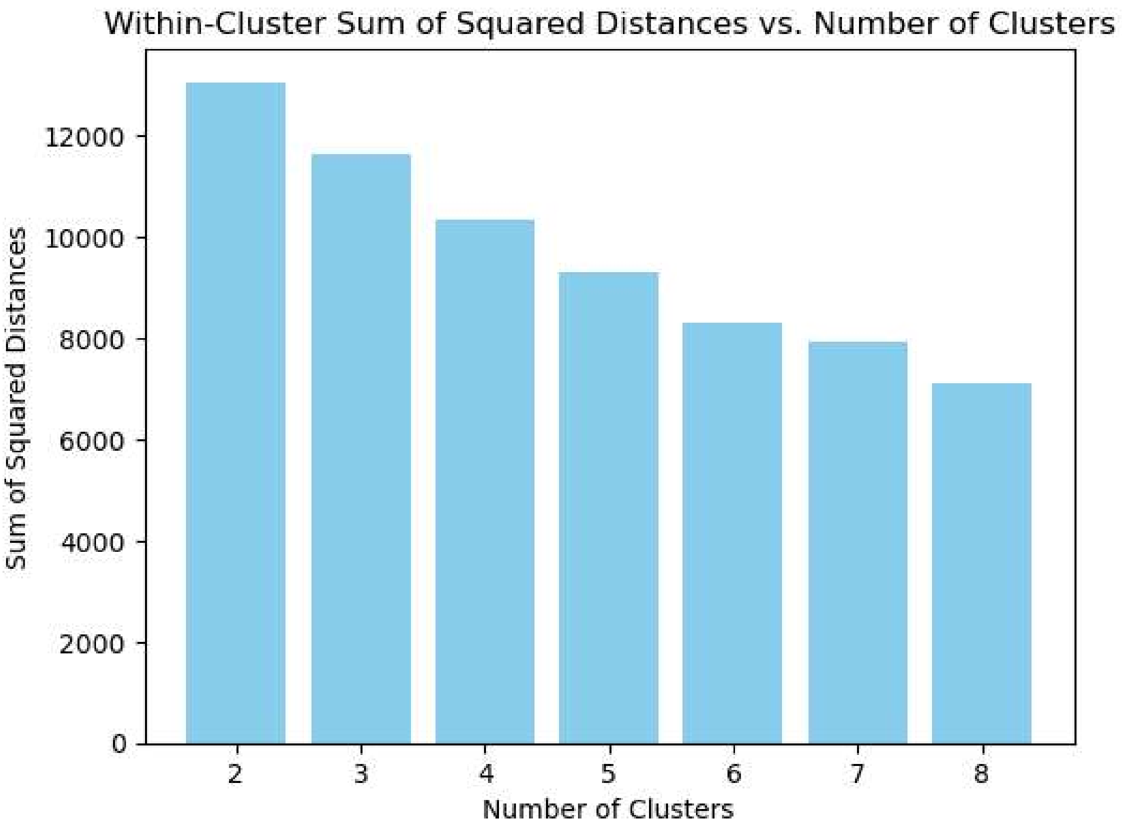
In [72]:
```
# Scatter plot of the first two principal components
plt.scatter(MD_pca[:, 0], MD_pca[:, 1], color="grey")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA Projection")

# Add axes for better interpretation
plt.axhline(0, color='black',linewidth=0.5)
plt.axvline(0, color='black',linewidth=0.5)

plt.show()
```



In [73]:
```
# Setting seed for reproducibility
np.random.seed(1234)

# Standardize the data (if needed)
scaler = StandardScaler()
MD_x_scaled = scaler.fit_transform(MD_x)

# Perform k-means clustering
n_clusters = 3  # Adjust the number of clusters as needed
model = KMeans(n_clusters=n_clusters, random_state=1234, n_init=10)
cluster_labels = model.fit_predict(MD_x_scaled)
```

In [74]:

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

# Standardize the data (if needed)
scaler = StandardScaler()
MD_x_scaled = scaler.fit_transform(MD_x)

# Perform k-means clustering with varying number of clusters
num_clusters_range = range(2, 9)
inertia_values = []

for n_clusters in num_clusters_range:
    model = KMeans(n_clusters=n_clusters, random_state=1234, n_init=10)
    model.fit(MD_x_scaled)
    inertia_values.append(model.inertia_)

# Plot the bar chart
plt.bar(num_clusters_range, inertia_values, color='skyblue')
plt.title('Within-Cluster Sum of Squared Distances vs. Number of Clusters')
plt.xlabel('Number of Clusters')
plt.ylabel('Sum of Squared Distances')
plt.show()
```

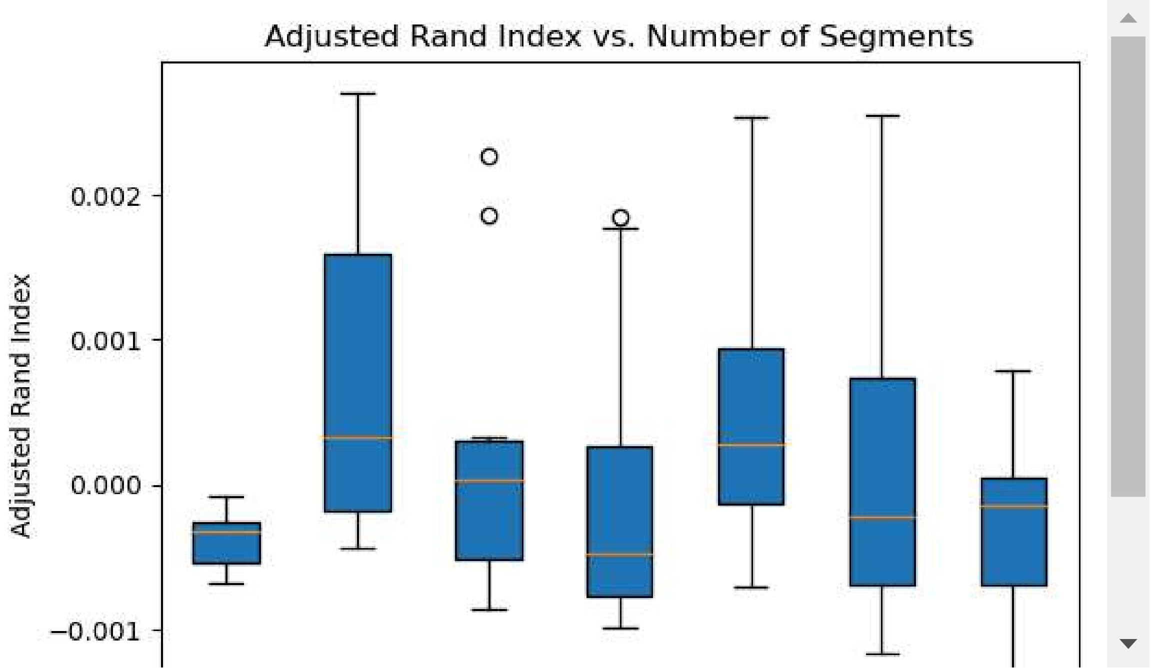


In [78]:

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

# Define the number of bootstrap samples and repetitions
n_bootstraps = 100
n_repetitions = 10

# Perform bootstrapped KMeans clustering
num_clusters_range = range(2, 9)
bootstrapped_results = []

for n_clusters in num_clusters_range:
    rand_indices = []
    for _ in range(n_repetitions):
        bootstrap_sample = resample(MD_x, random_state=np.random.randint(10
        model = KMeans(n_clusters=n_clusters, random_state=np.random.randin
        labels_true = np.random.randint(0, n_clusters, len(bootstrap_sample
        labels_pred = model.fit_predict(bootstrap_sample)
        rand_indices.append(adjusted_rand_score(labels_true, labels_pred))

    bootstrapped_results.append(rand_indices)

# Convert the results to a NumPy array for easier handling
bootstrapped_results = np.array(bootstrapped_results)
```

In [79]:

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

# Define the number of bootstrap samples and repetitions
n_bootstraps = 100
n_repetitions = 10

# Perform bootstrapped KMeans clustering and calculate adjusted Rand index
num_clusters_range = range(2, 9)
bootstrapped_results = []

for n_clusters in num_clusters_range:
    rand_indices = []
    for _ in range(n_repetitions):
        bootstrap_sample = resample(MD_x, random_state=np.random.randint(10
        model = KMeans(n_clusters=n_clusters, random_state=np.random.randin
        labels_true = np.random.randint(0, n_clusters, len(bootstrap_sample
        labels_pred = model.fit_predict(bootstrap_sample)
        rand_indices.append(adjusted_rand_score(labels_true, labels_pred))

    bootstrapped_results.append(rand_indices)

# Convert the results to a NumPy array for easier handling
bootstrapped_results = np.array(bootstrapped_results)

# Create boxplots
plt.boxplot(bootstrapped_results.T, labels=list(num_clusters_range), vert=T
plt.title('Adjusted Rand Index vs. Number of Segments')
plt.xlabel('Number of Segments')
plt.ylabel('Adjusted Rand Index')
plt.show()
```

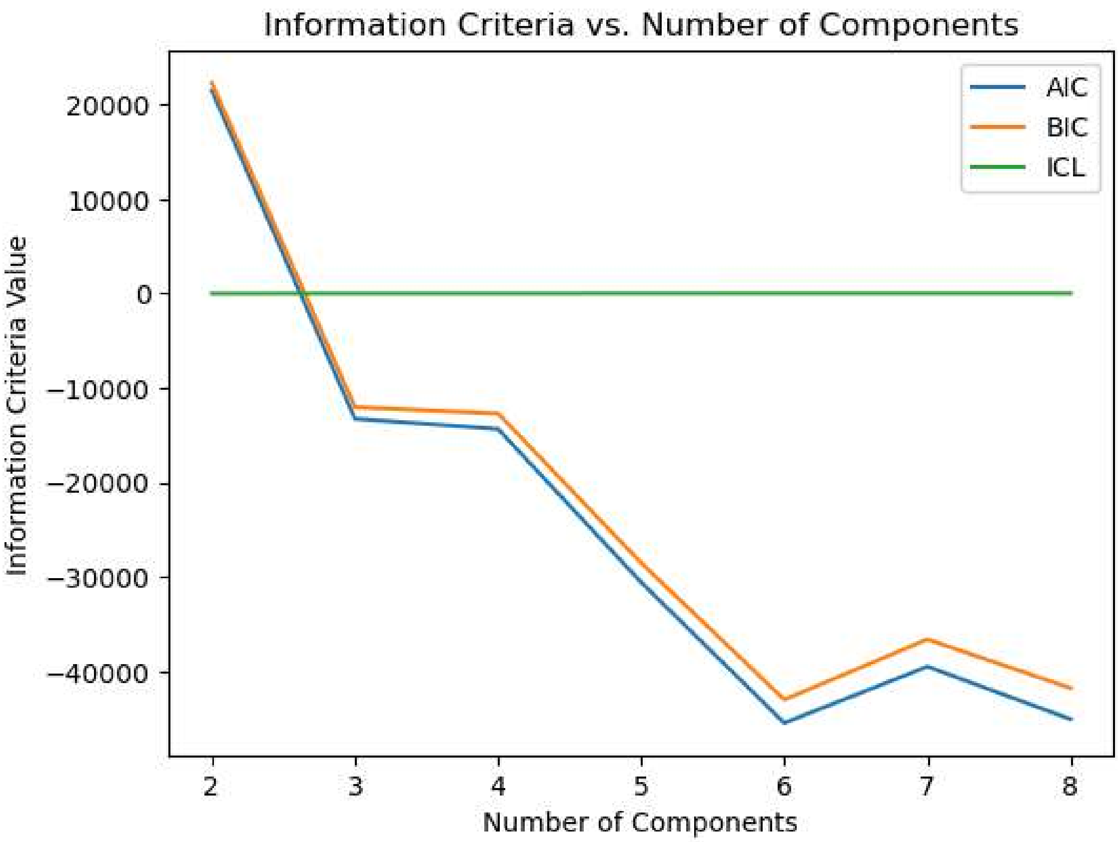


In [95]:

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

# Standardize the data (if needed)
scaler = StandardScaler()
MD_x_scaled = scaler.fit_transform(MD_x)

# Define the range of components to evaluate
n_components_range = range(2, 9)

# Initialize lists to store AIC, BIC, and ICL values
aic_values = []
bic_values = []
icl_values = []

# Loop over different numbers of components
for n_components in n_components_range:
    # Create a Gaussian Mixture Model
    gmm = GaussianMixture(n_components=n_components, random_state=1234)

    # Fit the model to your scaled data
    gmm.fit(MD_x_scaled)

    # Calculate AIC, BIC, and ICL
    aic_values.append(gmm.aic(MD_x_scaled))
    bic_values.append(gmm.bic(MD_x_scaled))
    icl_values.append(gmm.lower_bound_)

# Create a DataFrame to store the results
results_df = pd.DataFrame({
    'Components': n_components_range,
    'AIC': aic_values,
    'BIC': bic_values,
    'ICL': icl_values
})

# Print the results
print(results_df)
```

```
   Components           AIC           BIC        ICL
0           2  21456.798102  22275.412880  -7.276944
1           3 -13273.952049 -12043.389189   4.728132
2           4 -14347.590292 -12705.079351   5.149294
3           5 -30575.623904 -28521.164881  10.789272
4           6 -45458.433005 -42992.025901  15.964361
5           7 -39492.826265 -36614.471079  13.965184
6           8 -45054.182133 -41763.878864  15.932569
```

In [97]:
```
plt.plot(results_df['Components'], results_df['AIC'], label='AIC')
plt.plot(results_df['Components'], results_df['BIC'], label='BIC')
plt.plot(results_df['Components'], results_df['ICL'], label='ICL')

# Add labels and a legend
plt.xlabel('Number of Components')
plt.ylabel('Information Criteria Value')
plt.title('Information Criteria vs. Number of Components')
plt.legend()

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



In [83]:
```
# Select the cluster assignment for the four-segment solution
MD_k4 = cluster_labels
```

In [101]:

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

# Convert data to DataFrame if not already
if not isinstance(MD_x, pd.DataFrame):
    MD_x = pd.DataFrame(MD_x)

# Perform latent class analysis using Gaussian Mixture Model
k_range = range(2, 9)
nrep = 10
md_m28 = None

for k in k_range:
    for _ in range(nrep):
        md = GaussianMixture(n_components=k, random_state=1234)
        md.fit(MD_x)
        if md_m28 is None or md.score(MD_x) > md_m28.score(MD_x):
            md_m28 = md

# Print the resulting model
print(md_m28)
```

```
GaussianMixture(n_components=8, random_state=1234)
```