Code conversion

Step 4

```
Import pandas as pd

df=pd.read_excel('McDonalds.xlsx')

print(df.columns)

df.head()
```

```
Md_x = df.iloc[:, 0:11].values

MD_x = (MD_x == "Yes").astype(int)

result = np.round(np.mean(MD_x, axis=0), 2)

print(result)
```

```
From sklearn.decomposition import PCA

Pca=PCA()

MD_pca = pca.fit_transform(MD_x)

#summary

print("Explained variance ratio:", pca.explained_variance_ratio_)
print("Singular values:", pca.singular_values_)

print(np.round(MD_pca,1))
```

```
import matplotlib.pyplot as plt

from mpl_toolkits.mplot3d import Axes3D

# 3D plot

plt.subplot(1, 2, 2, projection='3d')

plt.scatter(MD_pca[:, 0], MD_pca[:, 1], MD_pca[:, 2], color='grey')

plt.title('PCA Plot (3D)')

plt.xlabel('PC1')
```

```
plt.ylabel('PC2')

plt.zlabel('PC3')

plt.show()

#not accurately as shown in the figure but explains same thing
```

Step 5

```
from sklearn.cluster import KMeans

wcss=[]

for k in range(2, 9):

    # Perform K-means clustering

    kmeans = KMeans(n_clusters=k, n_init=10, random_state=0)

    cluster_assignments = kmeans.fit_predict(MD_x)

    wcss.append(kmeans.intertia_)

    # Save clustering results

    cluster_results[k] = cluster_assignments

#relabeling clusters based on segmentation criteria

relabel_mapping = {}

for k, assignments in cluster_results.items():

    unique_labels = np.unique(assignments)

    relabel_mapping[k] = {old_label: new_label for new_label, old_label in
enumerate(unique_labels)}
```

```
#plotting wcss to know number of optimal segments

plt.plot(range(2, 9), wcss, marker='o')

plt.title('Within-Cluster Sum of Squares vs Number of Clusters')

plt.xlabel('Number of Clusters')

plt.ylabel('Within-Cluster Sum of Squares (WCSS)')

plt.xticks(range(2, 9))
```

```
plt.grid(True)
plt.show()
```

```
from sklearn.utils import resample
# Assuming MD_x is wer data matrix
bootstrap_results = {}

# Range of cluster numbers
for k in range(2, 9):
    bootstrap_cluster_assignments = []
    for _ in range(100):  # Number of bootstrap samples
        # Resample the data with replacement
        bootstrap_sample = resample(MD_x)
  kmeans = KMeans(n_clusters=k, n_init=10, random_state=0)
        cluster_assignments = kmeans.fit_predict(bootstrap_sample)
        # Save clustering results
        bootstrap_cluster_assignments.append(cluster_assignments)
    # Save bootstrap results
    bootstrap_results[k] = bootstrap_cluster_assignments
```

```
from sklearn.metrics import adjusted_rand_score

adjusted_rand_indices = {}
for k, cluster_assignments_list in bootstrap_results.items():
    aris = [] #Adjusted Rand Indices
    for cluster_assignments in cluster_assignments_list:
```

```
        # Compute adjusted Rand index
        true_labels = np.random.choice(range(2), len(cluster_assignments))  # Random labels for demonstration
        aris.append(adjusted_rand_score(true_labels, cluster_assignments))
    adjusted_rand_indices[k] = aris

# Plot adjusted Rand index against the number of segments
plt.errorbar(range(2, 9), [np.mean(adjusted_rand_indices[k]) for k in range(2, 9)], yerr=[np.std(adjusted_rand_indices[k]) for k in range(2, 9)], fmt='-o')
plt.title('Adjusted Rand Index vs Number of Segments')
plt.xlabel('Number of Segments')
plt.ylabel('Adjusted Rand Index')
plt.xticks(range(2, 9))
plt.grid(True)
plt.show()
```

```
# MD_kmeans_4  having 4 custer_result
MD_kmeans_4 = cluster_results[4]

# Create a histogram of cluster assignments
plt.hist(MD_kmeans_4, bins=range(5), align='left', edgecolor='black')
plt.xlabel('Cluster Assignment')
plt.ylabel('Frequency')
plt.xlim(0, 4)  # Set x-axis limit from 0 to 4
plt.xticks(range(5))  # Set x-ticks from 0 to 4
plt.title('Histogram of Cluster Assignments for 4 Clusters')
plt.show()
```

```
from sklearn.metrics import silhouette_samples, silhouette_score
```

```
# Calculate silhouette scores
silhouette_scores = silhouette_samples(MD_x, MD_kmeans_4)

# overall silhouette score
overall_silhouette_score = silhouette_score(MD_x, MD_kmeans_4)

# Create a bar plot of silhouette scores
fig, ax = plt.subplots()
y_lower = 10
for i in range(4):  # Number of clusters
    # Aggregate the silhouette scores  and sort them
    cluster_silhouette_scores = silhouette_scores[MD_kmeans_4 == i]
    cluster_silhouette_scores.sort()

    # Calculate the number of samples in the current cluster
    size_cluster_i = cluster_silhouette_scores.shape[0]
    y_upper = y_lower + size_cluster_i


    color = plt.cm.viridis(float(i) / 4)  # Color map for different clusters
    ax.fill_betweenx(np.arange(y_lower, y_upper),
                     0, cluster_silhouette_scores,
                     facecolor=color, edgecolor=color, alpha=0.7)

    # Label the silhouette plots with their cluster numbers at the middle
    ax.text(-0.05, y_lower + 0.5 * size_cluster_i, str(i))
```

```
    # Compute the new y_lower for next plot
    y_lower = y_upper + 10  # 10 for the 0 samples

# Set labels, title, and legend
ax.set_xlabel("Silhouette coefficient values")
ax.set_ylabel("Cluster label")
ax.set_title("Silhouette plot for K-means clustering")
ax.axvline(x=overall_silhouette_score, color="red", linestyle="--")  # Add a vertical line for the average silhouette score
ax.set_yticks([])  # Clear the yaxis labels / ticks
ax.set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])

# Display the average silhouette score
ax.text(0.7, 2, "Average silhouette score: {:.2f}".format(overall_silhouette_score))

plt.show()
```

```
#In Python, there isn't a direct equivalent for the stability measure provided by slswFlexclust() in the flexclust package. However, we can compute stability measures using resampling methods and metrics such as the Jaccard index, Adjusted Rand Index, or others.
from sklearn.metrics import jaccard_score
MD_kmeans_4 = cluster_results[4]
def compute_jaccard_index(cluster_assignments1, cluster_assignments2):
    return jaccard_score(cluster_assignments1, cluster_assignments2)
segment_stability = []
for i in range(4):  # Number of clusters
    # Select data points belonging to the current segment
    segment_indices = np.where(MD_kmeans_4 == i)[0]
```

```
    segment_assignments = MD_kmeans_4[segment_indices]

    # Compute Jaccard index between original segment and resampled segment
    jaccard_indices = []
    for _ in range(100):  # Number of resampling iterations
        # Resample the segment assignments
        resampled_segment_assignments = resample(segment_assignments)

        # Compute Jaccard index
        jaccard_index = compute_jaccard_index(segment_assignments,
resampled_segment_assignments)
        jaccard_indices.append(jaccard_index)

    # Average Jaccard index across resampling iterations
    average_jaccard_index = np.mean(jaccard_indices)
    segment_stability.append(average_jaccard_index)

# Plot segment stability
plt.plot(range(1, 5), segment_stability, marker='o')
plt.xlabel('Segment Number')
plt.ylabel('Segment Stability (Jaccard Index)')
plt.title('Segment Stability for K-means Clustering (4 clusters)')
plt.xticks(range(1, 5))
plt.ylim(0, 1)
plt.grid(True)
plt.show()
```

#Flexmix function can be used in python as gaussianmixture function

```
# Defining a function to perform stepwise mixture model selection
def stepFlexmix(data, components_range, nrep, verbose=False):
    best_model = None
    best_bic = np.inf
        for n_components in components_range:
        for _ in range(nrep):
            # Fit a Gaussian Mixture Model with diagonal covariance structure
            gmm = GaussianMixture(n_components=n_components,
covariance_type='diag', random_state=np.random.randint(1000))
            gmm.fit(data)
        # Calculate the Bayesian Information Criterion (BIC)
            bic = gmm.bic(data)


            # Update the best model and BIC if necessary
            if bic < best_bic:
                best_bic = bic
                best_model = gmm


        if verbose:
            print(f"Number of components: {n_components}, BIC: {best_bic}")


    return best_model
# Perform stepwise mixture model selection
best_gmm_model = stepFlexmix(MD_x, components_range=range(2, 9), nrep=10,
verbose=False)
# Print the best model
print(best_gmm_model)
```

```
from sklearn.mixture import GaussianMixture
```

```
# Define a function to calculate ICL
def calculate_icl(gmm, data):
    return gmm.lower_bound_
# Perform stepwise mixture model selection
components_range = range(2, 9)
nrep = 10
aic_values = []
bic_values = []
icl_values = []

for n_components in components_range:
    best_bic = np.inf
    best_model = None

    for _ in range(nrep):
        # Fit a Gaussian Mixture Model with diagonal covariance structure
        gmm = GaussianMixture(n_components=n_components,
covariance_type='diag', random_state=np.random.randint(1000))
        gmm.fit(MD_x)

        # Calculate AIC and BIC
        aic = gmm.aic(MD_x)
        bic = gmm.bic(MD_x)

        # Update the best model and BIC if necessary
        if bic < best_bic:
            best_bic = bic
```

```
        best_model = gmm

    aic_values.append(aic)
    bic_values.append(best_bic)
    icl_values.append(calculate_icl(best_model, MD_x))

# Plot AIC, BIC, and ICL against the number of components
plt.plot(components_range, aic_values, label='AIC', marker='o')
plt.plot(components_range, bic_values, label='BIC', marker='o')
plt.plot(components_range, icl_values, label='ICL', marker='o')
plt.xlabel('Number of Components')
plt.ylabel('Value of Information Criteria')
plt.title('Information Criteria for Gaussian Mixture Models')
plt.legend()
plt.grid(True)
plt.show()
```

```
best_gmm_model_4 = GaussianMixture(n_components=4, covariance_type='diag', random_state=1234)
best_gmm_model_4.fit(MD_x)
gmm_cluster_assignments = best_gmm_model_4.predict(MD_x)
# Create a contingency table comparing the cluster assignments from K-means and GMM
contingency_table = np.zeros((4, 4))
for i in range(4):
    for j in range(4):
        contingency_table[i, j] = np.sum((MD_kmeans_4 == i) & (gmm_cluster_assignments == j))
# Display the contingency table
```

```
print(contingency_table)
```

#In Python, we can't directly replicate the functionality of flexmix from R, as flexmix provides finite mixture modeling with a wide variety of distributions and clustering methods. However , we can use the clustering assignments from K-means as inputs to train a new model using the flexmix-like behavior.

```
MD_df['cluster'] = MD_kmeans_4

MD_x = MD_df.drop(columns=['cluster']).values

# Fit a Gaussian Mixture Model with diagonal covariance structure using the K-means cluster assignments

gmm_with_kmeans_clusters = GaussianMixture(n_components=4, covariance_type='diag', random_state=1234)

gmm_with_kmeans_clusters.fit(MD_x)

# Get cluster assignments from the trained GMM

gmm_cluster_assignments = gmm_with_kmeans_clusters.predict(MD_x)

# Create a contingency table comparing the cluster assignments from K-means and GMM with K-means clusters

contingency_table = np.zeros((4, 4))

for i in range(4):

   for j in range(4):

      contingency_table[i, j] = np.sum((MD_kmeans_4 == i) & (gmm_cluster_assignments == j))

# Display the contingency table

print(contingency_table)
```

```
# Compute log-likelihood for the GMM trained with K-means cluster assignments

log_likelihood_kmeans = gmm_with_kmeans_clusters.score(MD_x)

# Compute log-likelihood for the GMM trained directly with the flexmix-like behavior

log_likelihood_flexmix_like = best_gmm_model_4.score(MD_x)
```

```
print("Log-likelihood for the GMM trained with K-means cluster assignments:", log_likelihood_kmeans)

print("Log-likelihood for the GMM trained directly with the flexmix-like behavior:", log_likelihood_flexmix_like)
```

```
# Create a table of counts for the 'Like' variable

like_counts = pd.value_counts(df['Like'])

# Reverse the order of the table

reversed_like_counts = like_counts[::-1]

# Print the reversed table

print(reversed_like_counts)


# Create a new column 'Like.n' by subtracting each value of 'Like' from 6

df['Like.n'] = 6 - df['Like'].astype(int)


# Display the table of counts for the 'Like.n' variable

like_n_counts = df['Like.n'].value_counts()

print(like_n_counts)
```

```
import patsy


# Get the names of the first 11 columns of the DataFrame

predictor_names = df.columns[:11]


# Construct the formula string

formula_str = "Like_n ~ " + " + ".join(predictor_names)


# Convert the formula string to a formula object

formula = patsy.Formula(formula_str)
```

```
# Display the formula object
print(formula)
```

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

# Define the number of components (k)
k = 2

# Define the number of repetitions
nrep = 10

# Perform stepwise finite mixture modeling
best_bic = np.inf
best_model = None

for _ in range(nrep):
    # Fit a Gaussian Mixture Model
    gmm = GaussianMixture(n_components=k, covariance_type='full',
random_state=np.random.randint(1000))
    gmm.fit(df)

    # Calculate the Bayesian Information Criterion (BIC)
    bic = gmm.bic(df)

    # Update the best model and BIC if necessary
    if bic < best_bic:
        best_bic = bic
```

```
    best_model = gmm


# Display the best model
print(best_model)
```

#In Python, there isn't a direct equivalent of the refit() function from the flexmix package in R. However, we can refit the Gaussian Mixture Model (GMM) using the same parameters obtained from the stepwise finite mixture modeling.

```
# Refit the best model obtained from stepwise finite mixture modeling
MD_ref2 = best_model
# Summary of the refitted model
print(MD_ref2)
```

```
# Get cluster assignments for each data point
cluster_assignments = MD_ref2.predict(df)
# Plot the data points colored by their cluster assignments
plt.figure(figsize=(8, 6))
sns.scatterplot(x='x_variable_name', y='y_variable_name',
hue=cluster_assignments, data=df, palette='viridis')
plt.title('Cluster Assignments from Refitted Gaussian Mixture Model')
plt.xlabel('X Variable')
plt.ylabel('Y Variable')
plt.legend(title='Cluster')
plt.grid(True)
plt.show()
```

Step 6

```
#In Python, we can perform hierarchical clustering using the scipy library.

from scipy.cluster.hierarchy import linkage, dendrogram

# Perform hierarchical clustering

Z = linkage(MD_x.T, method='complete', metric='euclidean')


# Plot the dendrogram

plt.figure(figsize=(10, 6))

dendrogram(Z, labels=range(1, MD_x.shape[1] + 1))

plt.title('Hierarchical Clustering Dendrogram')

plt.xlabel('Data Points')

plt.ylabel('Distance')

plt.show()
```

```
# Reverse the order of the variables after hierarchical clustering

MD_vclust_order_rev = MD_vclust_order[::-1]


# Get the unique cluster labels from K-means clustering

cluster_labels = np.unique(MD_k4)


# Create a bar chart

plt.figure(figsize=(10, 6))

for cluster_label in cluster_labels:

    cluster_counts = np.sum(MD_k4 == cluster_label, axis=0)

    plt.bar(range(len(cluster_counts)), cluster_counts, label=f'Cluster
{cluster_label}', alpha=0.7)


# Shade the bars based on the order of variables after hierarchical clustering
```

```
for i, order in enumerate(MD_vclust_order_rev):
    plt.axvspan(i - 0.5, i + 0.5, color='lightgrey', alpha=0.5)

plt.title('Cluster Assignments from K-means with Shading based on Hierarchical Clustering Order')
plt.xlabel('Variables')
plt.ylabel('Counts')
plt.xticks(range(len(MD_vclust_order_rev)), MD_vclust_order_rev)
plt.legend()
plt.show()
```

```
# Define colors for each cluster
colors = ['b', 'g', 'r', 'c']

# Create a scatter plot
plt.figure(figsize=(10, 6))
for cluster_label, color in zip(np.unique(MD_k4), colors):
    cluster_indices = MD_k4 == cluster_label
    plt.scatter(MD_pca[cluster_indices, 0], MD_pca[cluster_indices, 1], c=color, label=f'Cluster {cluster_label}', alpha=0.7)

# Plot projection axes from PCA
projection_axes = projAxes(MD_pca)  # We need to implement this function or use a library that provides it
plt.quiver(*MD_pca.mean(axis=0), *projection_axes[:, 0], color='k', scale=3, label='PC1')
plt.quiver(*MD_pca.mean(axis=0), *projection_axes[:, 1], color='k', scale=3, label='PC2')
```

```
plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.title('Cluster Assignments from K-means on PCA')

plt.legend()

plt.grid(True)

plt.show()
```

Step 7

#In Python, we can create a mosaic plot to visualize the relationship between the cluster assignments obtained from K-means clustering (k4) and the Like variable.

```
from statsmodels.graphics.mosaicplot import mosaic


# Create a DataFrame with cluster assignments and the 'Like' variable

data = pd.DataFrame({'Cluster': k4, 'Like': df['Like']})


# Create a mosaic plot

plt.figure(figsize=(10, 6))

mosaic(data, ['Cluster', 'Like'], title='Mosaic Plot of Cluster Assignments and Like Variable')

plt.xlabel('Segment Number')

plt.show()
```

```
contingency_table = pd.crosstab(k4, df['Gender'])

# Create a mosaic plot

plt.figure(figsize=(10, 6))

mosaic(contingency_table.stack(), title='Mosaic Plot of Cluster Assignments and Gender', labelizer=lambda k: f'Cluster {k[0]} - Gender {k[1]}', gap=0.01)

plt.show()
```

```
# Create a list to hold the age values for each cluster
age_by_cluster = [df[df['k4'] == cluster]['Age'] for cluster in np.unique(k4)]


# Create a boxplot
plt.figure(figsize=(10, 6))
plt.boxplot(age_by_cluster, labels=np.unique(k4), patch_artist=True, varwidth=True, notch=True)
plt.xlabel('Cluster')
plt.ylabel('Age')
plt.title('Boxplot of Age by Cluster')
plt.show()
```

```
#In Python, we can use the sklearn library to create a decision tree model similar to the ctree function in R's partykit package.
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.tree import plot_tree

# Encode the binary response variable
le = LabelEncoder()
y = le.fit_transform(k4 == 3)

# Encode categorical variables
X = df[['Like.n', 'Age', 'VisitFrequency', 'Gender']]
X_encoded = pd.get_dummies(X, drop_first=True)

# Create and fit the decision tree model
tree = DecisionTreeClassifier()
```

```
tree.fit(X_encoded, y)


# Plot the decision tree
plt.figure(figsize=(15, 10))
plot_tree(tree, feature_names=X_encoded.columns, class_names=['Not Cluster 3', 'Cluster 3'], filled=True)
plt.show()
```

Step 8

```
visit = df.groupby(k4)['VisitFrequency'].mean()
# Print the result
print(visit)
```

```
# Calculate the mean value of the 'Like.n' variable for each cluster
like = df.groupby(k4)['Like.n'].mean()
# Print the result
print(like)
```

```
# Calculate the proportion of females in each cluster
female = df['Gender'].eq('Female').groupby(k4).mean()


# Print the result
print(female)
```

#In Python, we can create a scatter plot to visualize the relationship between the mean visit frequency (visit) and the mean Like.n score (like) for each cluster obtained from K-means clustering.

```
# Create a scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(visit, like, s=10 * female, c=np.arange(1, len(visit) + 1))

# Set the x-axis and y-axis limits
plt.xlim(2, 4.5)
plt.ylim(-3, 3)

# Add text labels for each cluster
for i, txt in enumerate(range(1, len(visit) + 1)):
    plt.text(visit[i], like[i], txt)

# Add labels and title
plt.xlabel('Mean Visit Frequency')
plt.ylab
el('Mean Like.n Score')
plt.title('Relationship between Mean Visit Frequency and Mean Like.n Score by
Cluster')

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