# Import necessary libraries

from scipy import stats

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.preprocessing import LabelEncoder, StandardScaler

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans, DBSCAN

from sklearn.metrics import silhouette\_score, calinski\_harabasz\_score, davies\_bouldin\_score

from sklearn.mixture import GaussianMixture

import numpy as np

from tabulate import tabulate

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

import seaborn as sns

# Load the dataset

dataset = pd.read\_csv('Customers.csv')

# Show dataset information and first 5 rows

print("Dataset Information:")

print(dataset.info())

print("\nFirst 5 Rows of Dataset:")

print(tabulate(dataset.head(), headers='keys', tablefmt='pretty'))

# Fill missing values with the most frequent value in each column

dataset = dataset.apply(lambda x: x.fillna(x.mode()[0]) if x.isna().sum() > 0 else x)

print("Missing values handled.")

# Drop 'CustomerID'

if 'CustomerID' in dataset.columns:

dataset = dataset.drop('CustomerID', axis=1)

print("customerid id is removed")

label\_encoders = {}

# Convert categorical columns to numerical values

categorical\_columns = dataset.select\_dtypes(include='object').columns

if categorical\_columns.empty:

print("No categorical columns found for encoding.")

else:

for column in categorical\_columns:

le = LabelEncoder()

dataset[column] = le.fit\_transform(dataset[column])

label\_encoders[column] = le # Store encoder to access class mappings later

print("Encoding column:", column)

# Generate and print the mapping in the desired format

mapping = {idx: category for idx, category in enumerate(le.classes\_)}

mapping\_output = ', '.join([f"{idx}: {category}" for idx, category in mapping.items()])

print(f"\nMapping for column '{column}':")

print(mapping\_output)

print("Categorical data encoded.")

print("\nFirst 5 Rows of Dataset after Encoding:")

print(tabulate(dataset.head(), headers='keys', tablefmt='grid'))

# Normalize the data

scaler = StandardScaler()

normalized\_data = pd.DataFrame(scaler.fit\_transform(dataset), columns=dataset.columns)

# Plotting before and after normalization side by side

fig, axes = plt.subplots(1, 2, figsize=(15, 6))

# Plot data distribution before normalization

sns.boxplot(data=dataset, ax=axes[0])

axes[0].set\_title("Feature Distribution Before Normalization")

for label in axes[0].get\_xticklabels():

label.set\_rotation(35)

# Plot data distribution after normalization

sns.boxplot(data=normalized\_data, ax=axes[1])

axes[1].set\_title("Feature Distribution After Normalization")

for label in axes[1].get\_xticklabels():

label.set\_rotation(35)

plt.tight\_layout()

plt.show()

# Save the cleaned, normalized dataset

cleaned\_data\_afterNormalization = normalized\_data.copy() # Save a copy for later use

cleaned\_data\_afterNormalization.to\_csv('cleaned\_Customers\_dataset.csv', index=False)

print("Cleaned dataset saved as 'cleaned\_Customers\_dataset.csv'.")

def remove\_outliers\_iqr(data):

# Calculate Q1 (25th percentile) and Q3 (75th percentile)

Q1 = data.quantile(0.25)

Q3 = data.quantile(0.75)

IQR = Q3 - Q1

# Define bounds for outliers

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

outliers = ((data < lower\_bound) | (data > upper\_bound)).any(axis=1)

outlier\_data = data[outliers]

print(f"Number of Outliers: {outliers.sum()}")

# Filter out outliers

return data[~outliers]

# Apply the IQR outlier removal to normalized\_data

cleaned\_data\_ForKmeans\_and\_Gmm = remove\_outliers\_iqr(cleaned\_data\_afterNormalization)

z\_scores = np.abs(stats.zscore(cleaned\_data\_ForKmeans\_and\_Gmm))

threshold = 3

outliers = (z\_scores > threshold).any(axis=1)

print(outliers)

# Correlation matrix heatmap

plt.figure(figsize=(10, 8))

sns.heatmap(cleaned\_data\_afterNormalization.corr(), annot=True, cmap="coolwarm", fmt=".2f")

plt.title("Correlation Matrix")

plt.show()

# Plot histograms for each column

cleaned\_data\_afterNormalization.hist(bins=15, figsize=(15, 10))

plt.suptitle('Distribution of Features')

plt.show()

from sklearn.cluster import KMeans

import matplotlib.pyplot as plt

from sklearn.metrics import silhouette\_score

from sklearn.decomposition import PCA

import numpy as np

# Elbow Method to determine optimal K

sse = []

k\_values = range(1, 11) # Test k from 1 to 10

for k in k\_values:

kmeans = KMeans(n\_clusters=k, random\_state=0, n\_init=10) # Added n\_init

kmeans.fit(cleaned\_data\_afterNormalization)

sse.append(kmeans.inertia\_)

# Plot SSE vs. number of clusters (k)

plt.figure(figsize=(8, 5))

plt.plot(k\_values, sse, marker='o', linestyle='-')

plt.xlabel('Number of Clusters (k)')

plt.ylabel('Sum of Squared Distances (SSE)')

plt.title('Elbow Method for Optimal k')

plt.grid() # Added grid for better readability

plt.show()

print(sse)

# Silhouette Score analysis

silhouette\_scores = []

for k in range(2, 11): # Silhouette score requires at least 2 clusters

kmeans = KMeans(n\_clusters=k, random\_state=0, n\_init=10) # Added n\_init

labels = kmeans.fit\_predict(cleaned\_data\_afterNormalization)

silhouette\_scores.append(silhouette\_score(cleaned\_data\_afterNormalization, labels))

# Plot Silhouette Score vs. number of clusters (k)

plt.figure(figsize=(8, 5))

plt.plot(range(2, 11), silhouette\_scores, marker='o', linestyle='-')

plt.xlabel('Number of Clusters (k)')

plt.ylabel('Silhouette Score')

plt.title('Silhouette Method for Optimal k')

plt.grid() # Added grid for better readability

plt.show()

# PCA Analysis

pca = PCA().fit(cleaned\_data\_afterNormalization)

explained\_variance = np.cumsum(pca.explained\_variance\_ratio\_)

# Plot cumulative explained variance

plt.figure(figsize=(8, 5))

plt.plot(range(1, len(explained\_variance) + 1), explained\_variance, marker='o', linestyle='-')

plt.xlabel('Number of Principal Components')

plt.ylabel('Cumulative Explained Variance')

plt.title('PCA Analysis')

plt.axhline(y=0.90, color='r', linestyle='--') # 90% variance line for reference

plt.grid() # Added grid for better readability

plt.show()

import numpy as np

import pandas as pd

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score

from sklearn.model\_selection import KFold

import matplotlib.pyplot as plt

# Assuming cleaned\_data\_afterNormalization is a DataFrame

# Number of clusters to evaluate

k\_values = range(2, 11)

n\_splits = 5 # Number of folds for cross-validation

# Prepare KFold cross-validation

kf = KFold(n\_splits=n\_splits, shuffle=True, random\_state=42)

# Store silhouette scores

cross\_val\_silhouette\_scores = {k: [] for k in k\_values}

# Cross-validated silhouette score calculation

for k in k\_values:

for train\_index, test\_index in kf.split(cleaned\_data\_afterNormalization):

# Use .iloc to access the DataFrame rows by index

X\_train = cleaned\_data\_afterNormalization.iloc[train\_index]

X\_test = cleaned\_data\_afterNormalization.iloc[test\_index]

kmeans = KMeans(n\_clusters=k, random\_state=0, n\_init=10)

labels = kmeans.fit\_predict(X\_train)

score = silhouette\_score(X\_train, labels)

cross\_val\_silhouette\_scores[k].append(score)

# Calculate average silhouette scores

avg\_silhouette\_scores = [np.mean(cross\_val\_silhouette\_scores[k]) for k in k\_values]

# Plotting average silhouette scores

plt.figure(figsize=(8, 5))

plt.plot(k\_values, avg\_silhouette\_scores, marker='o', linestyle='-')

plt.xlabel('Number of Clusters (k)')

plt.ylabel('Average Silhouette Score')

plt.title('Cross-Validated Silhouette Scores for Optimal k')

plt.grid()

plt.show()

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score, calinski\_harabasz\_score, davies\_bouldin\_score

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D # Importing the 3D toolkit

# Define the range of k and n\_components to iterate over

k\_values = [2, 3, 4, 5, 6, 7, 8]

pca\_components = [5, 6, 7] # Ensure components start from 3 for 3D plotting

# Initialize variables to track the best scores and corresponding parameters

best\_silhouette = -1

best\_calinski\_harabasz = -1

best\_davies\_bouldin = float('inf')

best\_params = {

'silhouette': None,

'calinski\_harabasz': None,

'davies\_bouldin': None

}

# Create a figure for 3D plots

fig = plt.figure(figsize=(15, 12))

fig.suptitle('K-Means Clustering on PCA-Reduced Data (3D)', fontsize=16, y=0.92)

for i, k in enumerate(k\_values):

for j, n\_components in enumerate(pca\_components):

# Apply PCA

pca = PCA(n\_components=n\_components)

reduced\_data = pca.fit\_transform(cleaned\_data\_afterNormalization)

# Apply K-Means clustering

kmeans = KMeans(n\_clusters=k, random\_state=0)

labels = kmeans.fit\_predict(reduced\_data)

# Check if n\_components is at least 3 for 3D plotting

if n\_components >= 3:

# Create a 3D plot

ax = fig.add\_subplot(len(k\_values), len(pca\_components), i \* len(pca\_components) + j + 1, projection='3d')

# Plot the clustering results in 3D

scatter = ax.scatter(reduced\_data[:, 0], reduced\_data[:, 1], reduced\_data[:, 2], c=labels, cmap='viridis', marker='o')

ax.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], kmeans.cluster\_centers\_[:, 2], s=300, c='red', marker='X')

ax.set\_title(f'k = {k}, PCA = {n\_components} components')

ax.set\_xlabel('Principal Component 1')

ax.set\_ylabel('Principal Component 2')

ax.set\_zlabel('Principal Component 3')

# Compute performance metrics

silhouette = silhouette\_score(reduced\_data, labels)

calinski\_harabasz = calinski\_harabasz\_score(reduced\_data, labels)

davies\_bouldin = davies\_bouldin\_score(reduced\_data, labels)

# Print performance metrics

print(f"K = {k}, PCA = {n\_components}:")

print(f" Silhouette Score: {silhouette:.3f}")

print(f" Calinski-Harabasz Score: {calinski\_harabasz:.3f}")

print(f" Davies-Bouldin Score: {davies\_bouldin:.3f}\n")

# Update best scores and parameters

if silhouette > best\_silhouette:

best\_silhouette = silhouette

best\_params['silhouette'] = (k, n\_components)

if calinski\_harabasz > best\_calinski\_harabasz:

best\_calinski\_harabasz = calinski\_harabasz

best\_params['calinski\_harabasz'] = (k, n\_components)

if davies\_bouldin < best\_davies\_bouldin:

best\_davies\_bouldin = davies\_bouldin

best\_params['davies\_bouldin'] = (k, n\_components)

else:

# Skip or handle cases where n\_components < 3 (could also plot in 2D if desired)

print(f"Skipping plot for k = {k} with PCA = {n\_components} components (less than 3 components).")

continue

# Adjust layout

plt.tight\_layout(rect=[0, 0, 1, 0.95])

plt.colorbar(scatter, ax=fig.axes, label='Cluster Label', orientation='vertical')

plt.show()

# Summary of best parameters for each metric

print("Best Parameters based on Performance Metrics:")

print(f"Best Silhouette Score: K = {best\_params['silhouette'][0]}, PCA Components = {best\_params['silhouette'][1]}")

print(f"Best Calinski-Harabasz Score: K = {best\_params['calinski\_harabasz'][0]}, PCA Components = {best\_params['calinski\_harabasz'][1]}")

print(f"Best Davies-Bouldin Score: K = {best\_params['davies\_bouldin'][0]}, PCA Components = {best\_params['davies\_bouldin'][1]}")

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score, calinski\_harabasz\_score, davies\_bouldin\_score

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D # For 3D plotting

# Set specific values for k and the number of PCA components

k\_values = [4, 6, 7]

n\_components = 5

# Create a figure for 3D plots

fig = plt.figure(figsize=(15, 12))

fig.suptitle('K-Means Clustering on PCA-Reduced Data (3D)', fontsize=16, y=0.92)

# Iterate over the specified values of k

for k in k\_values:

# Apply PCA with the specified number of components

pca = PCA(n\_components=n\_components)

reduced\_data = pca.fit\_transform(cleaned\_data\_afterNormalization)

# Apply K-Means clustering

kmeans = KMeans(n\_clusters=k, random\_state=0)

labels = kmeans.fit\_predict(reduced\_data) # Capture the labels here

# Create a 3D plot for the clustering results

ax = fig.add\_subplot(1, len(k\_values), k\_values.index(k) + 1, projection='3d')

scatter = ax.scatter(reduced\_data[:, 0], reduced\_data[:, 1], reduced\_data[:, 2], c=labels, cmap='viridis', marker='o')

ax.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], kmeans.cluster\_centers\_[:, 2], s=300, c='red', marker='X')

ax.set\_title(f'K-Means Clustering (k = {k}, PCA = {n\_components} components)')

ax.set\_xlabel('Principal Component 1')

ax.set\_ylabel('Principal Component 2')

ax.set\_zlabel('Principal Component 3')

fig.colorbar(scatter, ax=ax, label='Cluster Label')

# Compute performance metrics

silhouette = silhouette\_score(reduced\_data, labels)

calinski\_harabasz = calinski\_harabasz\_score(reduced\_data, labels)

davies\_bouldin = davies\_bouldin\_score(reduced\_data, labels)

print(f"K = {k}, PCA = {n\_components}:")

print(f" Silhouette Score: {silhouette:.3f}")

print(f" Calinski-Harabasz Score: {calinski\_harabasz:.3f}")

print(f" Davies-Bouldin Score: {davies\_bouldin:.3f}\n")

# Adjust layout

plt.tight\_layout(rect=[0, 0, 1, 0.95])

plt.show()

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score, calinski\_harabasz\_score, davies\_bouldin\_score

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D # For 3D plotting

# Set specific values for k and n\_components

k = 6

n\_components = 5

# Apply PCA with 3 components

pca = PCA(n\_components=n\_components)

reduced\_data = pca.fit\_transform(cleaned\_data\_afterNormalization)

# Apply K-Means clustering

kmeans = KMeans(n\_clusters=k, random\_state=0)

labels = kmeans.fit\_predict(reduced\_data) # Capture the labels here

# Plot the clustering results in 3D

fig = plt.figure(figsize=(10, 8))

ax = fig.add\_subplot(111, projection='3d')

scatter = ax.scatter(reduced\_data[:, 0], reduced\_data[:, 1], reduced\_data[:, 2], c=labels, cmap='viridis', marker='o')

ax.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], kmeans.cluster\_centers\_[:, 2], s=300, c='red', marker='X')

ax.set\_title(f'K-Means Clustering (k = {k}, PCA = {n\_components} components)')

ax.set\_xlabel('Principal Component 1')

ax.set\_ylabel('Principal Component 2')

ax.set\_zlabel('Principal Component 3')

fig.colorbar(scatter, label='Cluster Label')

plt.show()

# Compute performance metrics

silhouette = silhouette\_score(reduced\_data, labels)

calinski\_harabasz = calinski\_harabasz\_score(reduced\_data, labels)

davies\_bouldin = davies\_bouldin\_score(reduced\_data, labels)

print(f"K = {k}, PCA = {n\_components}:")

print(f" Silhouette Score: {silhouette:.3f}")

print(f" Calinski-Harabasz Score: {calinski\_harabasz:.3f}")

print(f" Davies-Bouldin Score: {davies\_bouldin:.3f}")

import plotly.express as px

import plotly.graph\_objects as go

from sklearn.decomposition import PCA

from sklearn.cluster import KMeans

from sklearn.metrics import silhouette\_score, calinski\_harabasz\_score, davies\_bouldin\_score

# Set specific values for k and n\_components

k = 6

n\_components = 5

# Apply PCA with 5 components

pca = PCA(n\_components=n\_components)

reduced\_data = pca.fit\_transform(cleaned\_data\_afterNormalization)

# Apply K-Means clustering

kmeans = KMeans(n\_clusters=k, random\_state=0)

labels = kmeans.fit\_predict(reduced\_data) # Capture the labels here

centroids = kmeans.cluster\_centers\_

# Create a 3D scatter plot using Plotly

fig = go.Figure()

# Add points for each cluster

fig.add\_trace(go.Scatter3d(

x=reduced\_data[:, 0],

y=reduced\_data[:, 1],

z=reduced\_data[:, 2],

mode='markers',

marker=dict(

size=5,

color=labels, # Color by cluster

colorscale='Viridis',

opacity=0.7

),

name='Data Points'

))

# Add centroids

fig.add\_trace(go.Scatter3d(

x=centroids[:, 0],

y=centroids[:, 1],

z=centroids[:, 2],

mode='markers',

marker=dict(

size=10,

color='red',

symbol='x',

opacity=0.9

),

name='Centroids'

))

# Set titles and axis labels

fig.update\_layout(

title=f'K-Means Clustering (k = {k}, PCA = {n\_components} components)',

scene=dict(

xaxis\_title='Principal Component 1',

yaxis\_title='Principal Component 2',

zaxis\_title='Principal Component 3'

)

)

fig.show()

# Compute performance metrics

silhouette = silhouette\_score(reduced\_data, labels)

calinski\_harabasz = calinski\_harabasz\_score(reduced\_data, labels)

davies\_bouldin = davies\_bouldin\_score(reduced\_data, labels)

print(f"K = {k}, PCA = {n\_components}:")

print(f" Silhouette Score: {silhouette:.3f}")

print(f" Calinski-Harabasz Score: {calinski\_harabasz:.3f}")

print(f" Davies-Bouldin Score: {davies\_bouldin:.3f}")

from sklearn.neighbors import NearestNeighbors

import numpy as np

# Determine eps using Nearest Neighbors =distance to the nearest neighbor starts to increase sharply

neigh = NearestNeighbors(n\_neighbors=2)

nbrs = neigh.fit(cleaned\_data\_afterNormalization)

distances, indices = nbrs.kneighbors(cleaned\_data\_afterNormalization)

distances = np.sort(distances[:, 1], axis=0)

plt.plot(distances)

plt.xlabel('Data Points')

plt.ylabel('Distance to Nearest Neighbor')

plt.title('K-distance Graph for DBSCAN')

plt.show()

from sklearn.cluster import DBSCAN

import pandas as pd

from tabulate import tabulate

# Define your parameter values

min\_samples\_values = [7,8, 9, 10, 11] # Start around 8-10 based on dimensionality

eps\_values = [1.25,1.5, 1.75, 2.0] # Adjust these based on K-distance graph

cleaned\_data\_dbscan=cleaned\_data\_afterNormalization.copy()

# Assume cleaned\_data is already defined

results = [] # To store results for each combination

# Variable to track the best eps and min\_samples with the least -1 count

best\_eps = None

best\_min\_samples = None

min\_noise\_count = float('inf') # Start with infinity for comparison

# Loop through all combinations of eps and min\_samples

for min\_samples in min\_samples\_values:

for eps in eps\_values:

# Fit the DBSCAN model

db\_scanclustermodel = DBSCAN(eps=eps, min\_samples=min\_samples).fit(cleaned\_data\_afterNormalization)

# Add the cluster labels to cleaned\_data

cleaned\_data\_dbscan.loc[:, 'cluster'] = db\_scanclustermodel.labels\_

# Count and print the number of clusters, including noise (-1)

cluster\_counts = cleaned\_data\_dbscan['cluster'].value\_counts()

noise\_count = cluster\_counts.get(-1, 0) # Get count of noise points, default to 0

print(f"DBSCAN with eps={eps} and min\_samples={min\_samples}")

print(tabulate(cleaned\_data\_dbscan.head(), headers='keys', tablefmt='grid'))

print(f"Noise points (-1) count: {noise\_count}")

# Check if this is the least noise count observed so far

if noise\_count < min\_noise\_count:

min\_noise\_count = noise\_count

best\_eps = eps

best\_min\_samples = min\_samples

# Append the result for further analysis if needed

results.append((eps, min\_samples, noise\_count))

# Print the best parameters with the least -1 cluster count

print(f"\nBest eps: {best\_eps}, Best min\_samples: {best\_min\_samples} with noise count: {min\_noise\_count}")

from sklearn.cluster import DBSCAN

from sklearn.metrics import silhouette\_score, davies\_bouldin\_score

from sklearn.decomposition import PCA

import matplotlib.pyplot as plt

# Use the best parameters obtained from previous runs

best\_eps = 2.0

best\_min\_samples = 7

# Fit the DBSCAN model with the best parameters

db\_scanclustermodel = DBSCAN(eps=best\_eps, min\_samples=best\_min\_samples).fit(cleaned\_data\_dbscan)

# Add the cluster labels to cleaned\_data

cleaned\_data\_dbscan.loc[:, 'cluster'] = db\_scanclustermodel.labels\_

# Calculate performance metrics

if len(set(db\_scanclustermodel.labels\_)) > 1: # Ensure there is more than one cluster

silhouette\_avg = silhouette\_score(cleaned\_data\_dbscan, db\_scanclustermodel.labels\_)

db\_index = davies\_bouldin\_score(cleaned\_data\_dbscan, db\_scanclustermodel.labels\_)

else:

silhouette\_avg = -1 # If there's only one cluster or all are noise

db\_index = float('inf') # Arbitrarily high since there's no valid clustering

# Print the metrics

print(f"\nSilhouette Score: {silhouette\_avg}")

print(f"Davies-Bouldin Index: {db\_index}")

# PCA for 2D visualization

pca = PCA(n\_components=2)

reduced\_data = pca.fit\_transform(cleaned\_data\_dbscan)

# Create a scatter plot

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

scatter = plt.scatter(reduced\_data[:, 0], reduced\_data[:, 1], c=cleaned\_data\_dbscan['cluster'], cmap='viridis', marker='o', s=30)

plt.title(f"DBSCAN Clustering with eps={best\_eps} and min\_samples={best\_min\_samples}")

plt.xlabel("PCA Component 1")

plt.ylabel("PCA Component 2")

# Create a legend for clusters

plt.legend(\*scatter.legend\_elements(), title="Clusters")

# Show the plot

plt.grid()

plt.show()

from sklearn.mixture import GaussianMixture

from sklearn.metrics import silhouette\_score

# Range of components to try

n\_components\_range = range(1, 11)

aic\_values = []

bic\_values = []

silhouette\_scores = []

cleaned\_data\_gmm=cleaned\_data\_afterNormalization.copy()

for n\_components in n\_components\_range:

gmm = GaussianMixture(n\_components=n\_components, covariance\_type='full', random\_state=42)

gmm.fit(cleaned\_data\_gmm)

# Append AIC and BIC values

aic\_values.append(gmm.aic(cleaned\_data\_gmm))

bic\_values.append(gmm.bic(cleaned\_data\_gmm))

# If there are more than one cluster, compute silhouette score

if n\_components > 1:

cluster\_labels = gmm.predict(cleaned\_data\_gmm)

silhouette\_avg = silhouette\_score(cleaned\_data\_gmm, cluster\_labels)

silhouette\_scores.append(silhouette\_avg)

else:

silhouette\_scores.append(-1) # Not defined for a single cluster

# Plotting AIC and BIC-- lowest points on the AIC and BIC curves

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(n\_components\_range, aic\_values, marker='o', label='AIC')

plt.plot(n\_components\_range, bic\_values, marker='o', label='BIC')

plt.title('AIC and BIC for different n\_components')

plt.xlabel('Number of Components')

plt.ylabel('Score')

plt.legend()

# Plotting silhouette scores--highest silhouette score indicates the best-defined clusters

plt.subplot(1, 2, 2)

plt.plot(n\_components\_range[1:], silhouette\_scores[1:], marker='o')

plt.title('Silhouette Scores for different n\_components')

plt.xlabel('Number of Components')

plt.ylabel('Silhouette Score')

plt.tight\_layout()

plt.show()

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

from sklearn.mixture import GaussianMixture

from matplotlib.colors import ListedColormap

from mpl\_toolkits.mplot3d import Axes3D # Import 3D plotting

# Assuming cleaned\_data\_gmm is already defined and contains the normalized data

# Reduce the dimensions of the data to 3D using PCA

pca = PCA(n\_components=3)

reduced\_data = pca.fit\_transform(cleaned\_data\_gmm)

# Fit GMM with 7 components

gmm\_7 = GaussianMixture(n\_components=7, covariance\_type='full', random\_state=42)

gmm\_7.fit(reduced\_data)

labels\_7 = gmm\_7.predict(reduced\_data)

# Fit GMM with 5 components

gmm\_5 = GaussianMixture(n\_components=5, covariance\_type='full', random\_state=42)

gmm\_5.fit(reduced\_data)

labels\_5 = gmm\_5.predict(reduced\_data)

# Count points in each cluster for 7 components

cluster\_counts\_7 = pd.Series(labels\_7).value\_counts().sort\_index()

print("\nCluster counts for GMM with 7 components:")

print(cluster\_counts\_7)

# Count points in each cluster for 5 components

cluster\_counts\_5 = pd.Series(labels\_5).value\_counts().sort\_index()

print("\nCluster counts for GMM with 5 components:")

print(cluster\_counts\_5)

# Define custom colors for clusters

colors\_7 = ['red', 'green', 'blue', 'purple', 'orange', 'cyan', 'magenta']

colors\_5 = ['red', 'green', 'blue', 'purple', 'orange']

# Visualize results for 7 components in 3D

fig = plt.figure(figsize=(14, 6))

ax = fig.add\_subplot(121, projection='3d')

scatter = ax.scatter(reduced\_data[:, 0], reduced\_data[:, 1], reduced\_data[:, 2],

c=labels\_7, cmap=ListedColormap(colors\_7), marker='o', s=30)

ax.set\_title('GMM Clustering with 7 Components')

ax.set\_xlabel('PCA Component 1')

ax.set\_ylabel('PCA Component 2')

ax.set\_zlabel('PCA Component 3')

fig.colorbar(scatter, ticks=range(len(colors\_7)), label='Cluster Label')

ax.grid()

# Visualize results for 5 components in 3D

ax = fig.add\_subplot(122, projection='3d')

scatter = ax.scatter(reduced\_data[:, 0], reduced\_data[:, 1], reduced\_data[:, 2],

c=labels\_5, cmap=ListedColormap(colors\_5), marker='o', s=30)

ax.set\_title('GMM Clustering with 5 Components')

ax.set\_xlabel('PCA Component 1')

ax.set\_ylabel('PCA Component 2')

ax.set\_zlabel('PCA Component 3')

fig.colorbar(scatter, ticks=range(len(colors\_5)), label='Cluster Label')

ax.grid()

plt.tight\_layout()

plt.show()

from sklearn.metrics import silhouette\_score, calinski\_harabasz\_score, davies\_bouldin\_score

# Function to calculate performance metrics

def calculate\_performance\_metrics(data, labels):

silhouette\_avg = silhouette\_score(data, labels)

calinski\_harabasz = calinski\_harabasz\_score(data, labels)

davies\_bouldin = davies\_bouldin\_score(data, labels)

return silhouette\_avg, calinski\_harabasz, davies\_bouldin

# Calculate performance metrics for GMM with 7 components

metrics\_7 = calculate\_performance\_metrics(reduced\_data, labels\_7)

print(f"\nPerformance Metrics for GMM with 7 Components:")

print(f"Silhouette Score: {metrics\_7[0]}")

print(f"Calinski-Harabasz Index: {metrics\_7[1]}")

print(f"Davies-Bouldin Index: {metrics\_7[2]}")

# Calculate performance metrics for GMM with 5 components

metrics\_5 = calculate\_performance\_metrics(reduced\_data, labels\_5)

print(f"\nPerformance Metrics for GMM with 5 Components:")

print(f"Silhouette Score: {metrics\_5[0]}")

print(f"Calinski-Harabasz Index: {metrics\_5[1]}")

print(f"Davies-Bouldin Index: {metrics\_5[2]}")