In [ ]:
```
# Install required libraries
!pip install pandas
!pip install numpy
!pip install matplotlib
!pip install seaborn
!pip install scipy
!pip install scikit-learn
```

In [ ]:
```
# Import required libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import zscore
from sklearn.preprocessing import StandardScaler, PolynomialFeatures, MinMaxScaler, LabelEncoder
from sklearn.cluster import KMeans, AgglomerativeClustering, DBSCAN
from sklearn.metrics import silhouette_score, davies_bouldin_score, calinski_harabasz_score
from sklearn.decomposition import PCA
import scipy.cluster.hierarchy as sch
```

0. Introduction

The **Anuran Calls Dataset** consists of audio recordings from 60 specimens of frogs, representing 4 families, 8 genera, and 10 species. Each recording corresponds to an individual frog, and modern processing techniques were used to derive 22 Mel-frequency cepstral coefficients (MFCCs).
The objective of this assignment is to apply advanced clustering techniques, starting with K-Means, to group the frogs based on their MFCC features, and explore the clustering performance through various evaluation methods.

1. Data Processing and Exploration

1.1 Data Preparation

Apply label encoding to non-numeric relevant features (Family, Genus, Species) and drop unnecessary column (Record ID)

In [ ]:
```
# Load the dataset
filepath = "/content/Frogs_MFCCs.csv"
data = pd.read_csv(filepath)

mfcc_data = data.copy()

# Initialize a LabelEncoder instance
label_encoder = LabelEncoder()

# Apply Label Encoding to each metadata column and add it to mfcc_data
metadata_columns = ['Family', 'Genus', 'Species']
for column in metadata_columns:
    mfcc_data[column] = label_encoder.fit_transform(data[column])

# Drop the 'RecordID' column
mfcc_data = mfcc_data.drop(columns=['RecordID'])

mfcc_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7195 entries, 0 to 7194
Data columns (total 25 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   MFCCs_ 1  7195 non-null   float64
 1   MFCCs_ 2  7195 non-null   float64
 2   MFCCs_ 3  7195 non-null   float64
 3   MFCCs_ 4  7195 non-null   float64
 4   MFCCs_ 5  7195 non-null   float64
 5   MFCCs_ 6  7195 non-null   float64
 6   MFCCs_ 7  7195 non-null   float64
 7   MFCCs_ 8  7195 non-null   float64
 8   MFCCs_ 9  7195 non-null   float64
 9   MFCCs_10  7195 non-null   float64
 10  MFCCs_11  7195 non-null   float64
 11  MFCCs_12  7195 non-null   float64
 12  MFCCs_13  7195 non-null   float64
 13  MFCCs_14  7195 non-null   float64
 14  MFCCs_15  7195 non-null   float64
 15  MFCCs_16  7195 non-null   float64
 16  MFCCs_17  7195 non-null   float64
 17  MFCCs_18  7195 non-null   float64
 18  MFCCs_19  7195 non-null   float64
 19  MFCCs_20  7195 non-null   float64
 20  MFCCs_21  7195 non-null   float64
 21  MFCCs_22  7195 non-null   float64
 22  Family    7195 non-null   int64  
 23  Genus     7195 non-null   int64  
 24  Species   7195 non-null   int64  
dtypes: float64(22), int64(3)
memory usage: 1.4 MB
```

1.2 Exploratory Data Analysis

Analyze the dataset, and visualize feature distributions.

In [ ]:
```
mfcc_data.describe()  # summary statistics
```

Out[ ]:

	MFCCs_1	MFCCs_2	MFCCs_3	MFCCs_4	MFCCs_5	MFCCs_6	MFCCs_7	MFCCs_8	MFCCs_9	MFCC
count	7195.000000	7195.000000	7195.000000	7195.000000	7195.000000	7195.000000	7195.000000	7195.000000	7195.000000	7195.00
mean	0.989885	0.323584	0.311224	0.445997	0.127046	0.097939	-0.001397	-0.000370	0.128213	0.05
std	0.069016	0.218653	0.263527	0.160328	0.162722	0.120412	0.171404	0.116302	0.179008	0.12
min	-0.251179	-0.673025	-0.436028	-0.472676	-0.636012	-0.410417	-0.538982	-0.576506	-0.587313	-0.95
25%	1.000000	0.165945	0.138445	0.336737	0.051717	0.012581	-0.125737	-0.063109	0.004648	-0.00
50%	1.000000	0.302184	0.274626	0.481463	0.161361	0.072079	-0.052630	0.013265	0.189317	0.06
75%	1.000000	0.466566	0.430695	0.559861	0.222592	0.175957	0.085580	0.075108	0.265395	0.11
max	1.000000	1.000000	1.000000	1.000000	0.752246	0.964240	1.000000	0.551762	0.738033	0.52

8 rows × 25 columns

In [ ]:
```
mfcc_data.nunique()  # number of unique values in each column
```

Out[ ]:

	0
MFCCs_ 1	249
MFCCs_ 2	7140
MFCCs_ 3	7026
MFCCs_ 4	7175
MFCCs_ 5	7195
MFCCs_ 6	7195
MFCCs_ 7	7195
MFCCs_ 8	7195
MFCCs_ 9	7195
MFCCs_10	7195
MFCCs_11	7195
MFCCs_12	7195
MFCCs_13	7195
MFCCs_14	7195
MFCCs_15	7195
MFCCs_16	7195
MFCCs_17	7195
MFCCs_18	7195
MFCCs_19	7195
MFCCs_20	7195
MFCCs_21	7195
MFCCs_22	7195
Family	4
Genus	8

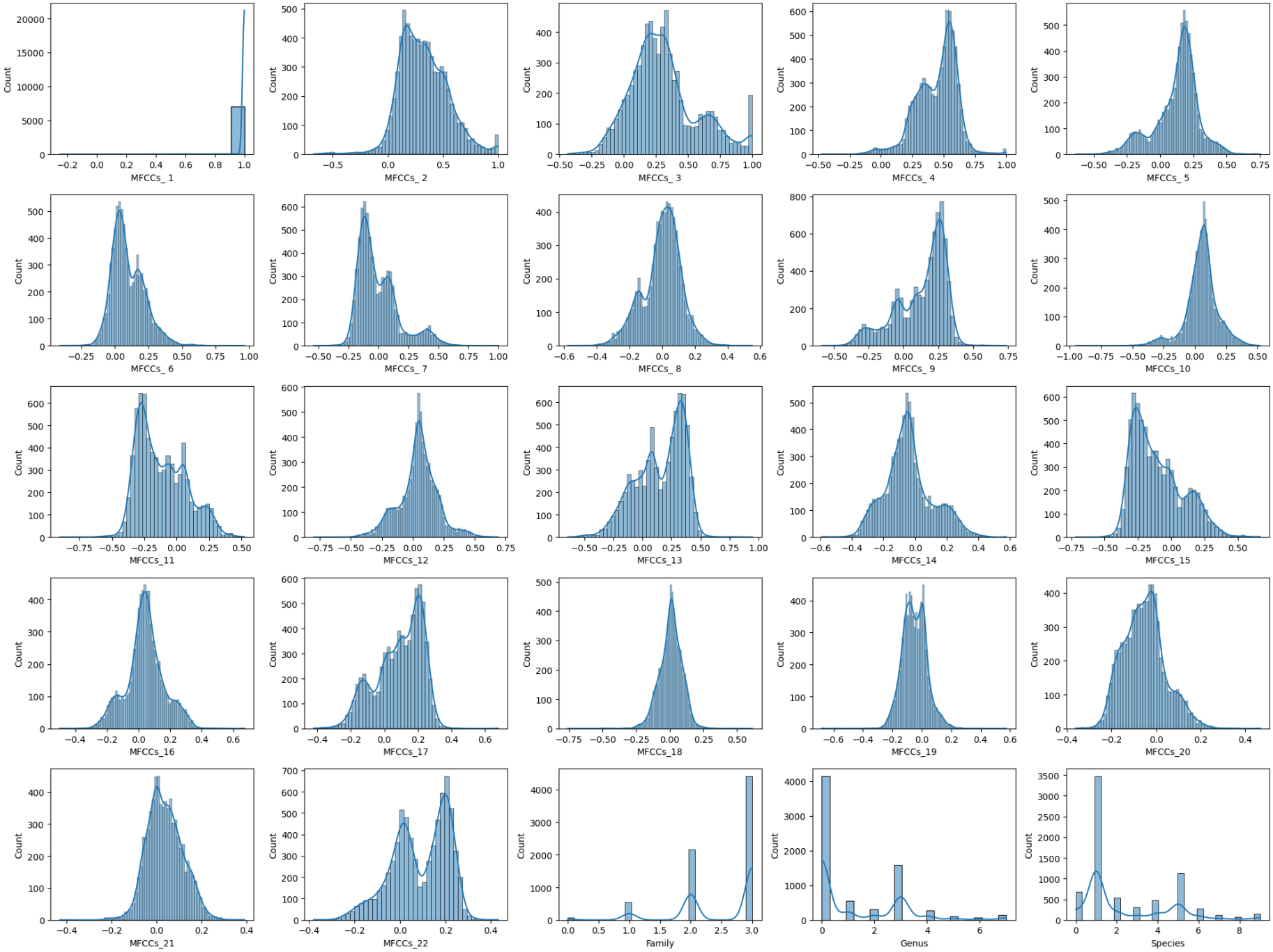
	0
Species	10

dtype: int64

Histograms and Pie Charts:

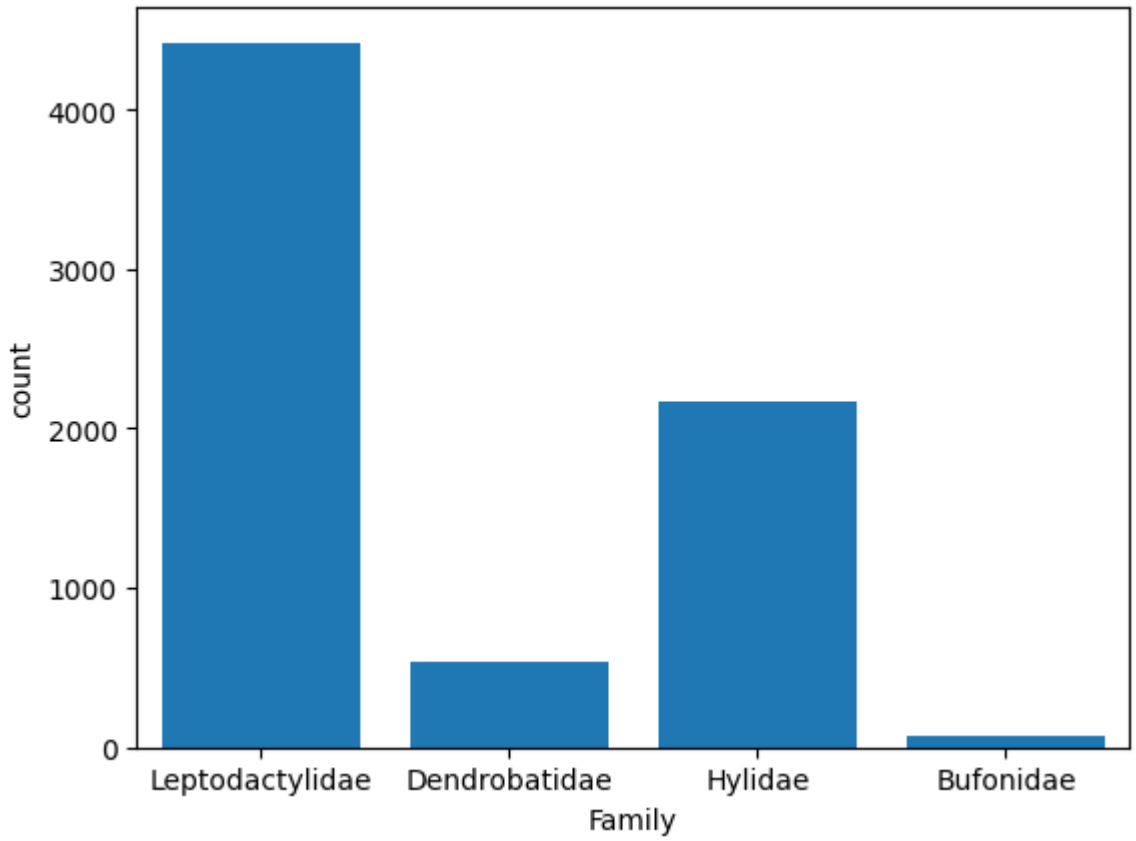
Show the frequency distribution of each feature. Ideal for identifying skewness and modality.

In [ ]:
```
def beautiful_plot(df):
    plt.figure(figsize=(20, 15))
    rows, cols = (df.shape[1] + 4) // 5, 5

    for idx, column in enumerate(df.columns):
        plt.subplot(rows, cols, idx + 1)
        sns.histplot(df[column], kde=True)

    plt.tight_layout()
    plt.show()


beautiful_plot(mfcc_data)
```


Count
MFCCs_ 1
MFCCs_ 2
MFCCs_ 3
MFCCs_ 4
MFCCs_ 5
MFCCs_ 6
MFCCs_ 7
MFCCs_ 8
MFCCs_ 9
MFCCs_10
MFCCs_11
MFCCs_12
MFCCs_13
MFCCs_14
MFCCs_15
MFCCs_16
MFCCs_17
MFCCs_18
MFCCs_19
MFCCs_20
MFCCs_21
MFCCs_22
Family
Genus
Species

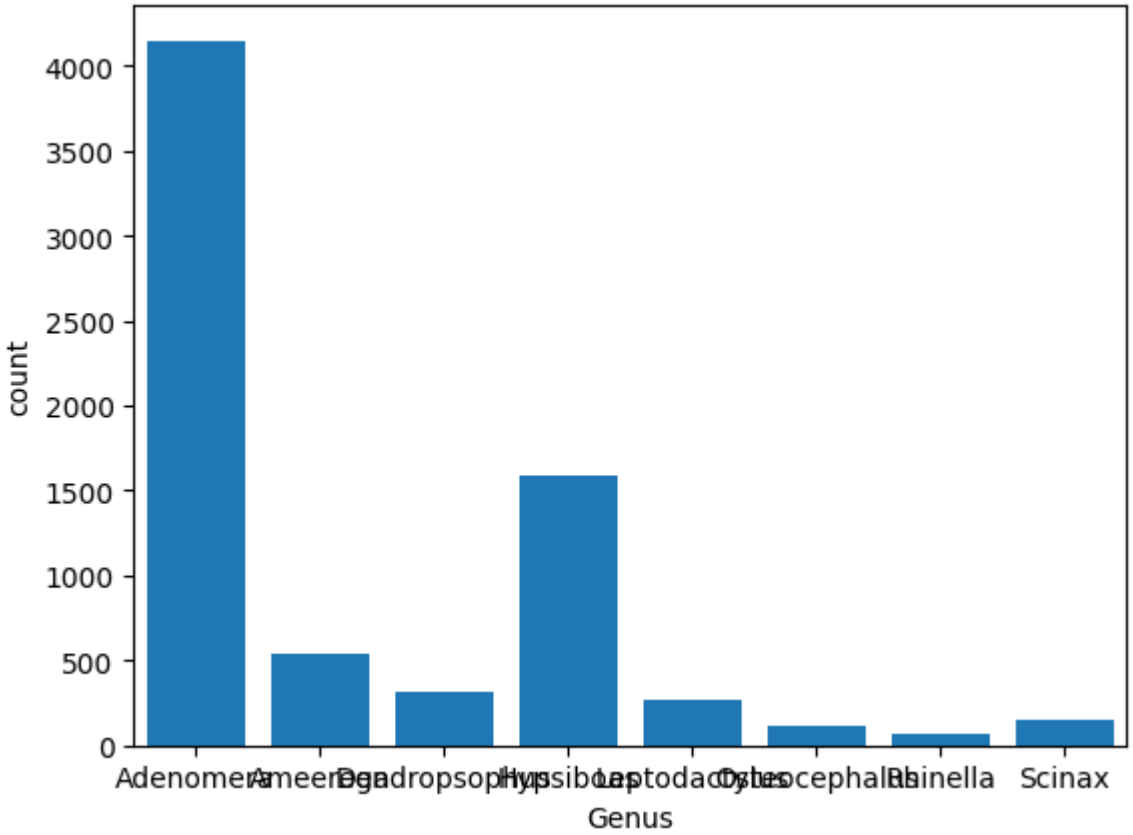
In [ ]:
```
sns.countplot(x="Family", data=data, saturation=10)
```

Out[ ]: <Axes: xlabel='Family', ylabel='count'>

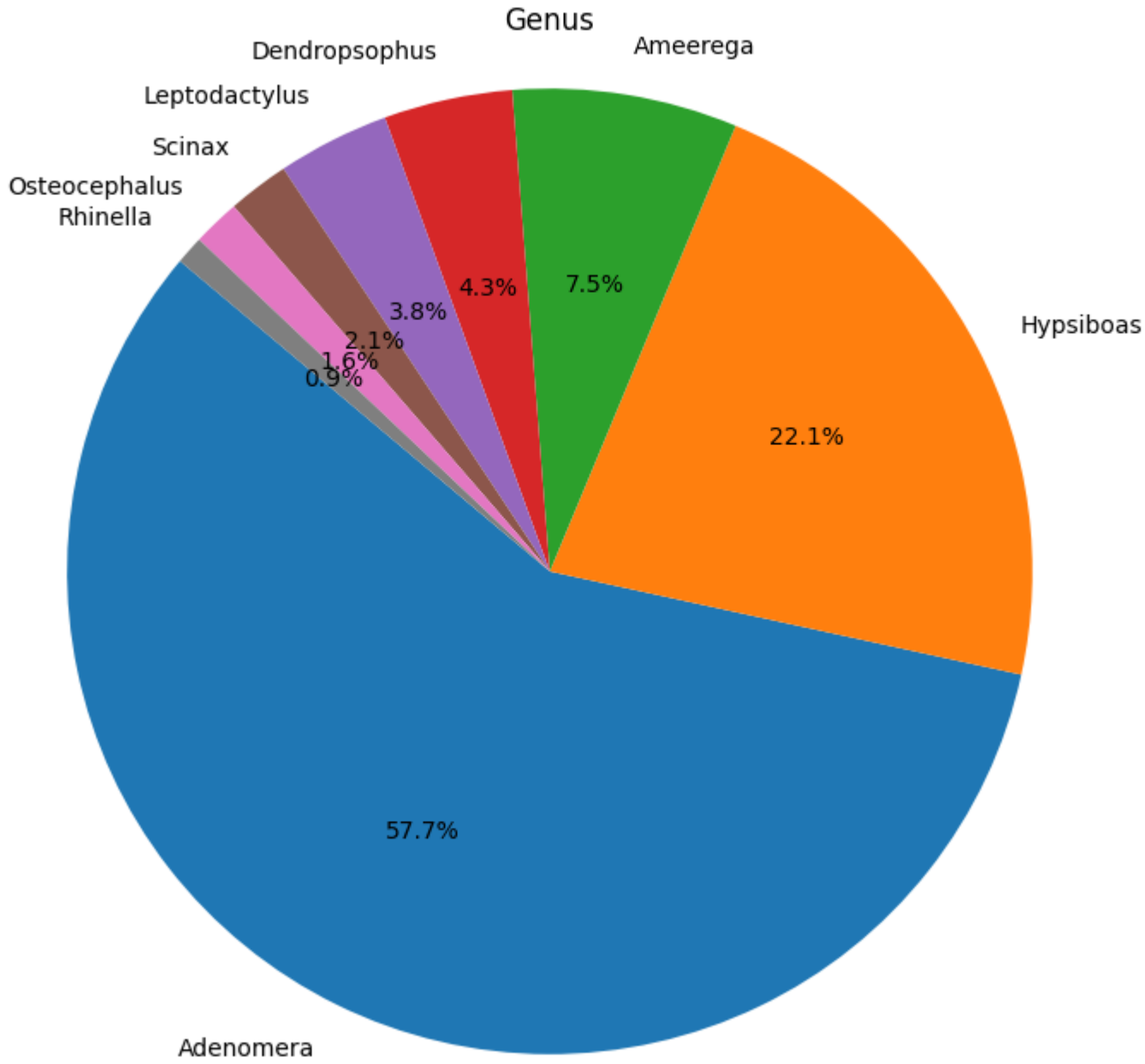


In [ ]:
```
sns.countplot(x="Genus", data=data, saturation=10)
```

Out[ ]: <Axes: xlabel='Genus', ylabel='count'>



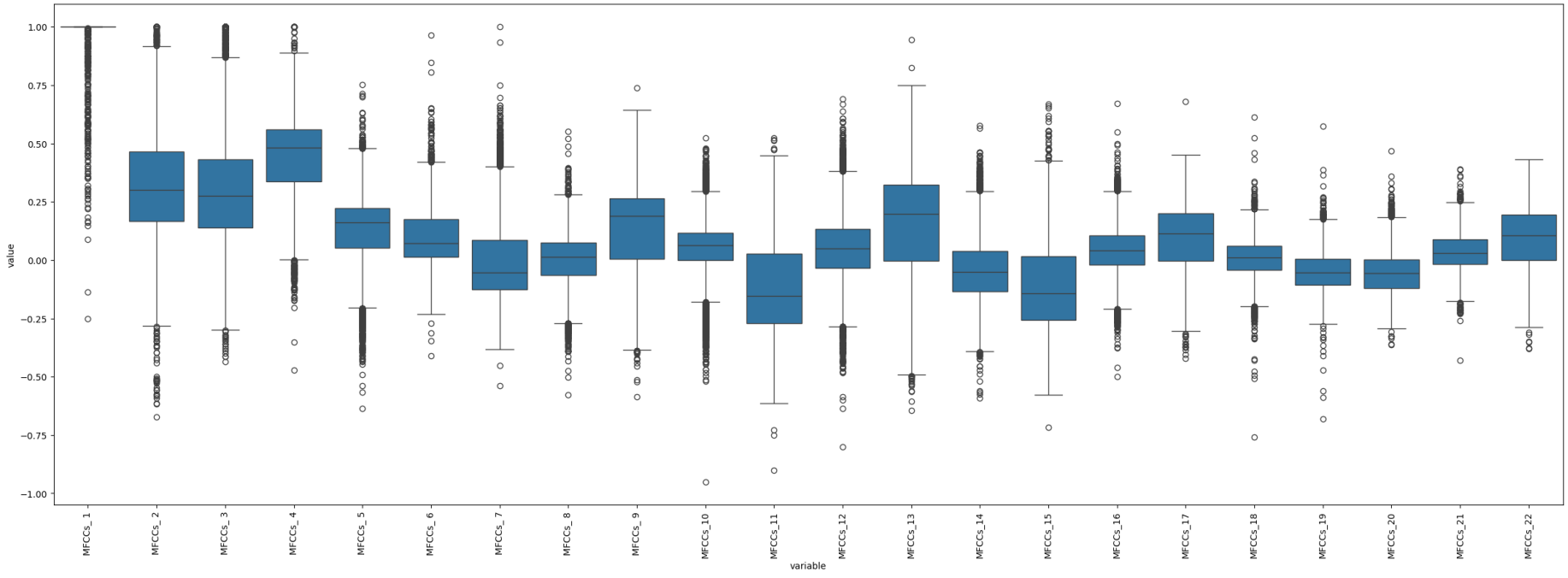
In [ ]:
```
genus_counts = data['Genus'].value_counts()

# Plotting the pie chart
plt.figure(figsize=(8, 8))
plt.pie(genus_counts, labels=genus_counts.index, autopct='%1.1f%%', startangle=140)
plt.title('Genus')
plt.axis('equal')
plt.show()
```


Genus
Dendropsophus
Ameerega
Leptodactylus
Scinax
Osteocephalus
Rhinella
Hypsiboas
4.3%
7.5%
3.8%
2.1%
1.6%
0.9%
22.1%
57.7%
Adenomera

Box Plots

Useful for spotting outliers and understanding the spread (quartiles) of the data.

In [ ]:
```
plt.figure(figsize=(30,10))
sns.boxplot(x="variable", y="value", data=pd.melt(mfcc_data.drop(columns=metadata_columns)))
plt.xticks(rotation=90)
plt.show()
```



Outliers

Check for missing values and outliers (using the **Z-score** method), as they can distort clustering results. Clean the data accordingly.

In [ ]:
```
mfcc_data.isnull().sum()  # number of missing values in each column
```

Out[ ]:

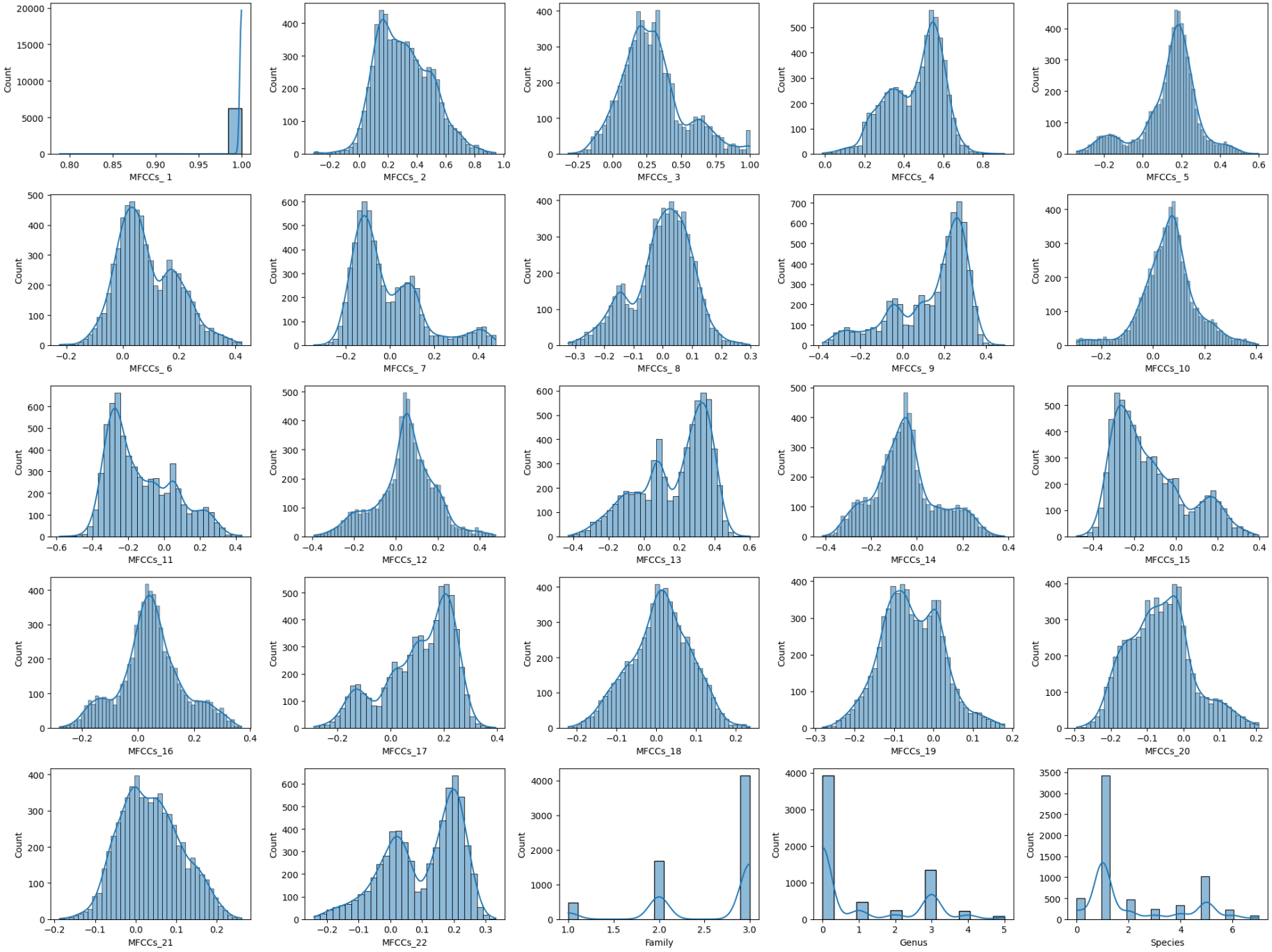
	0
MFCCs_ 1	0
MFCCs_ 2	0
MFCCs_ 3	0
MFCCs_ 4	0
MFCCs_ 5	0
MFCCs_ 6	0
MFCCs_ 7	0
MFCCs_ 8	0
MFCCs_ 9	0
MFCCs_10	0
MFCCs_11	0
MFCCs_12	0
MFCCs_13	0
MFCCs_14	0
MFCCs_15	0
MFCCs_16	0
MFCCs_17	0
MFCCs_18	0
MFCCs_19	0
MFCCs_20	0
MFCCs_21	0
MFCCs_22	0
Family	0
Genus	0

	0
Species	0

dtype: int64

In [ ]:
```python
# For each feature, remove the outliers that are more than 3 standard deviations away from the mean
def remove_outliers(df, columns, n_std=3):
    df_clean = df.copy()
    for column in columns:
        # Calculate mean and standard deviation
        mean = df_clean[column].mean()
        std = df_clean[column].std()

        # Calculate z-scores
        z_scores = (df_clean[column] - mean) / std

        # Remove outliers based on z-score threshold
        df_clean = df_clean[abs(z_scores) <= n_std]

    return df_clean

# Remove outliers from all numeric columns
print(f"Original shape: {mfcc_data.shape}")
mfcc_data_modified = remove_outliers(mfcc_data, mfcc_data.columns)
print(f"Shape after removing outliers: {mfcc_data_modified.shape}")
print(f"Number of outliers removed: {mfcc_data.shape[0] - mfcc_data_modified.shape[0]}")
beautiful_plot(mfcc_data_modified)
```

```
Original shape: (7195, 25)
Shape after removing outliers: (6295, 25)
Number of outliers removed: 900
```


MFCCs_ 1
MFCCs_ 2
MFCCs_ 3
MFCCs_ 4
MFCCs_ 5
MFCCs_ 6
MFCCs_ 7
MFCCs_ 8
MFCCs_ 9
MFCCs_10
MFCCs_11
MFCCs_12
MFCCs_13
MFCCs_14
MFCCs_15
MFCCs_16
MFCCs_17
MFCCs_18
MFCCs_19
MFCCs_20
MFCCs_21
MFCCs_22
Family
Genus
Species
Count

1.3 Feature Engineering

Derive new features from the existing MFCCs to potentially improve clustering performance.
Find the feature with least variance, and replace it with it's squarred feature

- By introducing polynomial features (like squaring a feature), non-linear relationships between the variables can be captured that might not be apparent when using the original linear features.
- The chosen feature with the lowest variance may not contribute significantly to the model's predictive power, as low variance often indicates that the feature does not vary much across the dataset. By creating a polynomial feature from this low-variance feature, we can potentially increase its variance and enhance its contribution to the model.

In [ ]:
```
mfcc_poly_df = mfcc_data_modified.copy()

"""Polynomial Features"""

# Get numeric columns only, excluding categorical features
numeric_columns = mfcc_data_modified.select_dtypes(include=['float64', 'int64']).columns

# Choose feature with lowest variance
feature_variances = mfcc_data_modified[numeric_columns].var()
lowest_variance_feature = feature_variances.sort_values(ascending=True).index[0]
print("Feature chosen for polynomial transformation (lowest variance): ", lowest_variance_feature)

# Create polynomial feature
mfcc_poly_df[lowest_variance_feature + "_squared"] = mfcc_poly_df[lowest_variance_feature] ** 2
lowest_variance_feature_squared = lowest_variance_feature + "_squared"

# Compare the variances
print("Original feature variance:", mfcc_poly_df[lowest_variance_feature].var())
print("Squared feature variance:", mfcc_poly_df[lowest_variance_feature_squared].var())

# Drop the original feature
mfcc_poly_df.drop(lowest_variance_feature, axis=1, inplace=True)

mfcc_poly_df.describe()
```

```
Feature chosen for polynomial transformation (lowest variance):  MFCCs_ 1
Original feature variance: 0.00012065697208707164
Squared feature variance: 0.00040984573928593015
```

Out[ ]:

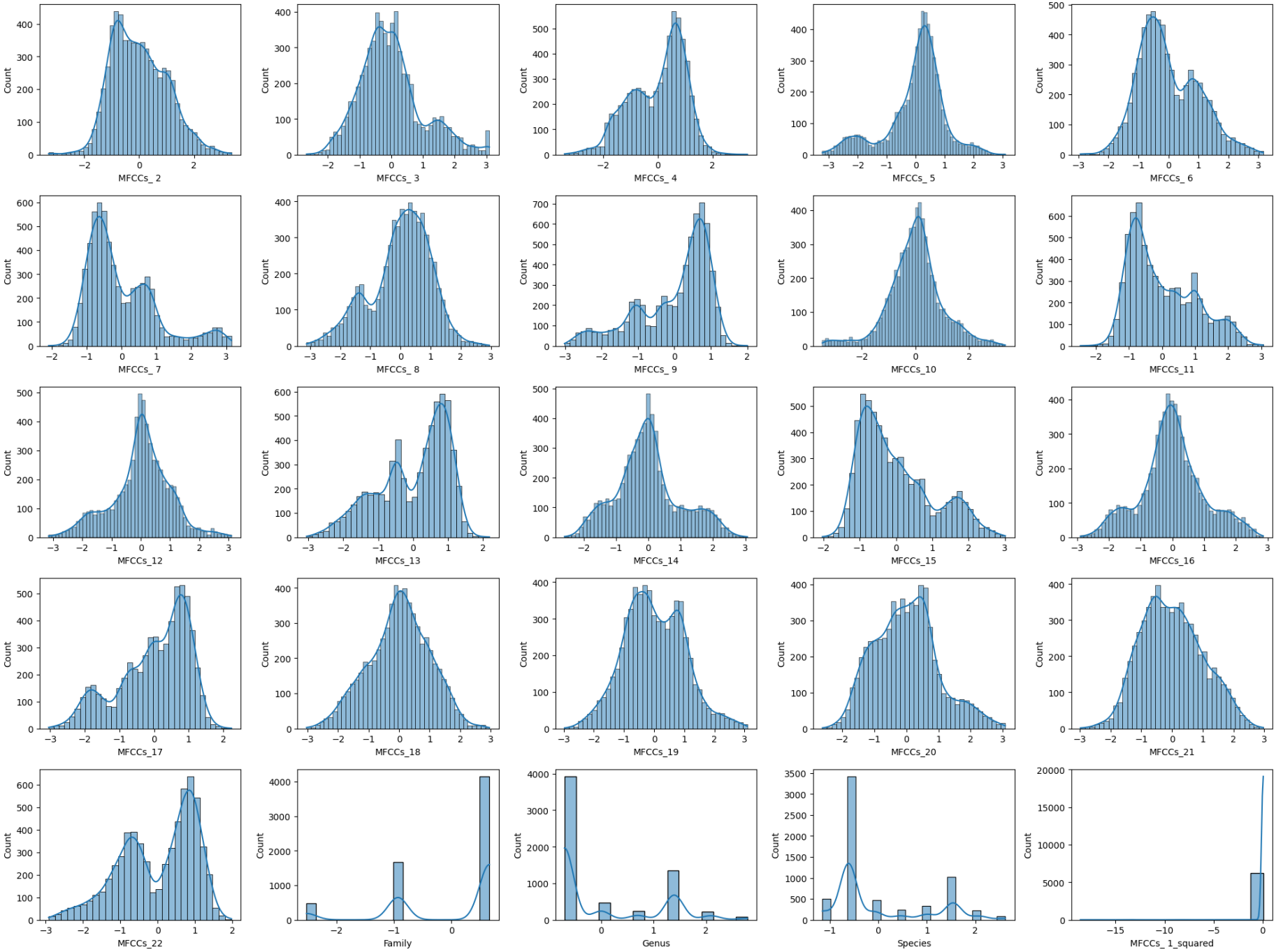
	MFCCs_ 2	MFCCs_ 3	MFCCs_ 4	MFCCs_ 5	MFCCs_ 6	MFCCs_ 7	MFCCs_ 8	MFCCs_ 9	MFCCs_10	MFCC
count	6295.000000	6295.000000	6295.000000	6295.000000	6295.000000	6295.000000	6295.000000	6295.000000	6295.000000	6295.00
mean	0.311538	0.291514	0.457376	0.138909	0.086787	-0.016620	-0.004219	0.140020	0.062947	-0.12
std	0.187884	0.229076	0.137885	0.148862	0.105777	0.156984	0.102519	0.172641	0.103389	0.18
min	-0.309681	-0.320006	-0.012733	-0.340247	-0.223474	-0.344937	-0.323505	-0.380777	-0.295007	-0.58
25%	0.164759	0.141898	0.350286	0.083624	0.010235	-0.130221	-0.059368	0.031349	0.007668	-0.27
50%	0.291677	0.262866	0.491699	0.168218	0.066066	-0.066834	0.009425	0.204725	0.065522	-0.17
75%	0.445899	0.400714	0.562472	0.223259	0.164923	0.072809	0.068171	0.269753	0.114735	0.01
max	0.944696	1.000000	0.908218	0.598381	0.421119	0.478249	0.298119	0.485847	0.409454	0.43

8 rows × 25 columns

1.4 Data Scaling

Standardize the MFCCs features to normalize them, as many clustering algorithms are sensitive to different scales.

In [ ]:
```
scaler = StandardScaler()
mfcc_scaled = scaler.fit_transform(mfcc_poly_df)
mfcc_scaled_df = pd.DataFrame(mfcc_scaled, columns=mfcc_poly_df.columns)

beautiful_plot(mfcc_scaled_df)
```


MFCCs_ 2
MFCCs_ 3
MFCCs_ 4
MFCCs_ 5
MFCCs_ 6
MFCCs_ 7
MFCCs_ 8
MFCCs_ 9
MFCCs_10
MFCCs_11
MFCCs_12
MFCCs_13
MFCCs_14
MFCCs_15
MFCCs_16
MFCCs_17
MFCCs_18
MFCCs_19
MFCCs_20
MFCCs_21
MFCCs_22
Family
Genus
Species
MFCCs_ 1_squared
Count

1.5. Feature Correlation Analysis

Investigate correlations between features and remove highly correlated features to avoid redundancy and improve clustering results.

Correlation Matrix

- The correlation matrix is a square matrix that shows the pairwise correlation coefficients for a set of variables, with values typically ranging from -1 to +1.

$$\begin{bmatrix} 1 & r_{XY} & r_{XZ} \\ r_{YX} & 1 & r_{YZ} \\ r_{ZX} & r_{ZY} & 1 \end{bmatrix}$$

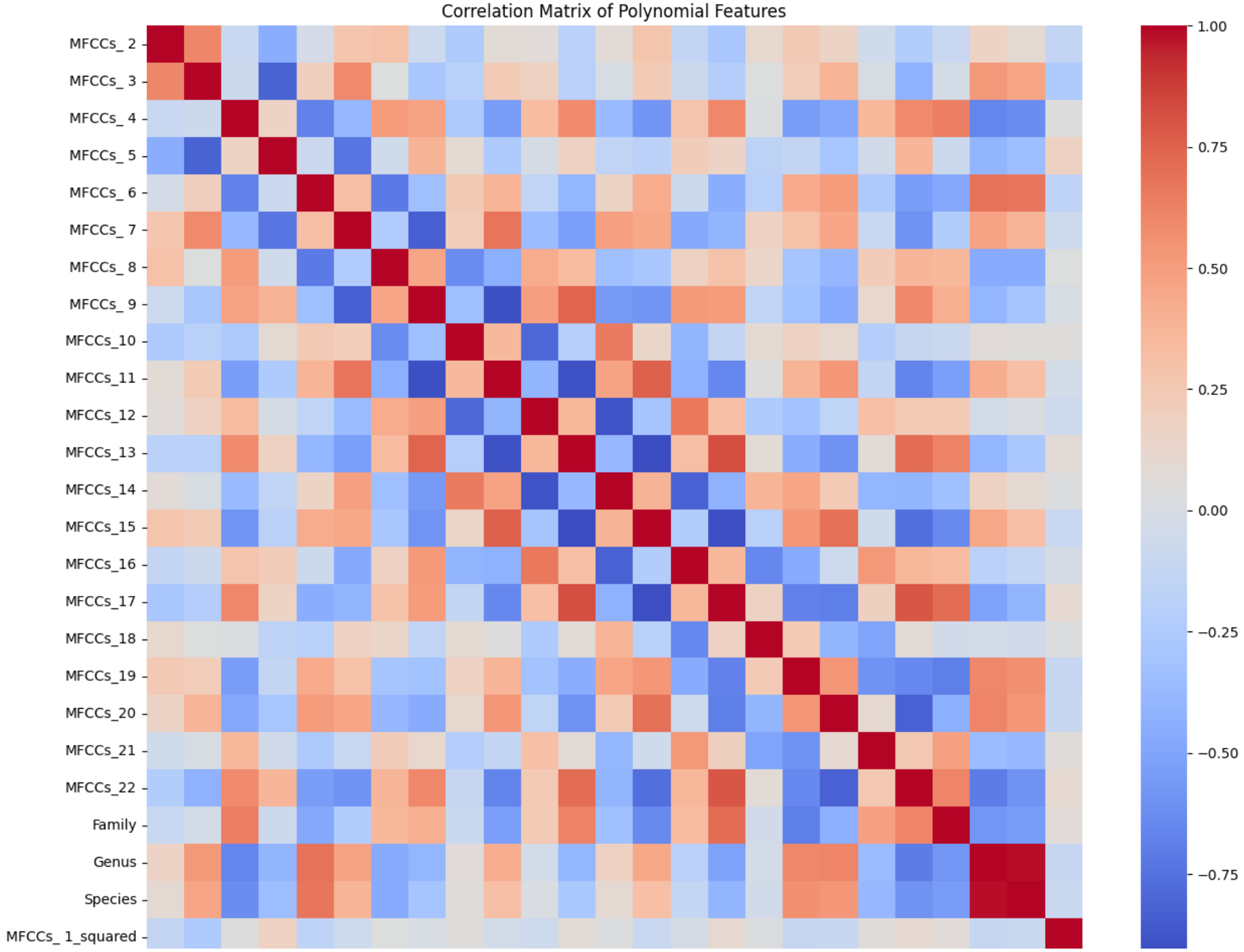
- The most common measure of correlation is Pearson's correlation coefficient, calculated as:

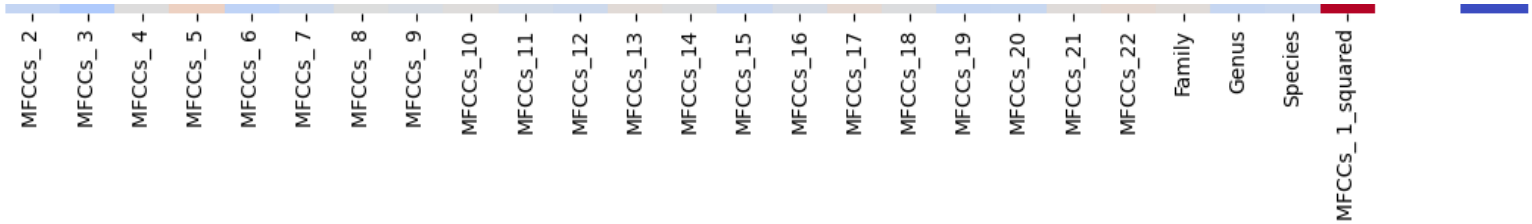
$$r = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y}$$

Where $\text{Cov}(X, Y)$ is the covariance between variables X and Y, and σ_X and σ_Y are the standard deviations of X and Y, respectively.

- A correlation coefficient close to +1 implies a strong positive correlation, meaning that as one variable increases, the other also tends to increase.
- A coefficient close to -1 indicates a strong negative correlation, suggesting that as one variable increases, the other tends to decrease.
- A coefficient around 0 suggests no linear correlation between the variables.

In [ ]:
```
# Compute correlation matrix
correlation_matrix = mfcc_scaled_df.corr()

# Plot heatmap to visualize correlations
plt.figure(figsize=(15, 12))
sns.heatmap(correlation_matrix, cmap="coolwarm", annot=False, fmt=".2f")
plt.title("Correlation Matrix of Polynomial Features")
plt.show()
```


Correlation Matrix of Polynomial Features
MFCCs_ 2
MFCCs_ 3
MFCCs_ 4
MFCCs_ 5
MFCCs_ 6
MFCCs_ 7
MFCCs_ 8
MFCCs_ 9
MFCCs_10
MFCCs_11
MFCCs_12
MFCCs_13
MFCCs_14
MFCCs_15
MFCCs_16
MFCCs_17
MFCCs_18
MFCCs_19
MFCCs_20
MFCCs_21
MFCCs_22
Family
Genus
Species
MFCCs_ 1_squared
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75



Dropping Highly Correlated Features

The correlation matrix helps identify features that are highly correlated with each other (in this case, 0.8). Highly correlated features provide similar information, leading to redundancy in the dataset. Thus, dropping them prevents multicollinearity and improves model interpretability.

In [ ]:
```
threshold = 0.8

# Select upper triangle of correlation matrix
upper_triangle = correlation_matrix.where(np.triu(np.ones(correlation_matrix.shape), k=1).astype(bool))

# Find features with correlation above the threshold
to_drop = [column for column in upper_triangle.columns if any(upper_triangle[column] > threshold)]
print(f"Features to drop (correlation > {threshold}):\n", to_drop)

# Drop the identified features
mfcc_reduced_df = mfcc_scaled_df.drop(columns=to_drop)

mfcc_reduced_df.describe()
```

```
Features to drop (correlation > 0.8):
 ['MFCCs_17', 'Species']
```

Out[ ]:

	MFCCs_ 2	MFCCs_ 3	MFCCs_ 4	MFCCs_ 5	MFCCs_ 6	MFCCs_ 7	MFCCs_ 8	MFCCs_ 9	MFC
count	6.295000e+03	6.295000e+03	6.295000e+03	6.295000e+03	6.295000e+03	6.295000e+03	6.295000e+03	6.295000e+03	6.2950(
mean	-1.805986e-17	7.223945e-17	3.070177e-16	5.417959e-17	-3.611973e-17	3.611973e-17	-1.805986e-17	-3.611973e-17	9.0299
std	1.000079e+00	1.000079e+00	1.000079e+00	1.000079e+00	1.000079e+00	1.000079e+00	1.000079e+00	1.000079e+00	1.00007
min	-3.306659e+00	-2.669721e+00	-3.409704e+00	-3.219058e+00	-2.933392e+00	-2.091566e+00	-3.114650e+00	-3.016884e+00	-3.4624(
25%	-7.812852e-01	-6.531785e-01	-7.767260e-01	-3.714140e-01	-7.237649e-01	-7.236995e-01	-5.379829e-01	-6.295127e-01	-5.3471
50%	-1.057182e-01	-1.250686e-01	2.489443e-01	1.968979e-01	-1.959086e-01	-3.198880e-01	1.331035e-01	3.748236e-01	2.4905
75%	7.151835e-01	4.767373e-01	7.622592e-01	5.666752e-01	7.387512e-01	5.697169e-01	7.061650e-01	7.515179e-01	5.0095
max	3.370204e+00	3.093050e+00	3.269959e+00	3.086814e+00	3.160973e+00	3.152600e+00	2.949323e+00	2.003311e+00	3.3517(

8 rows × 23 columns

2. K-Means Clustering

2.1 Elbow Method

The Elbow Method is a popular technique used in clustering to determine the optimal number of clusters (k) for a given dataset. It helps in selecting a suitable value of k by analyzing how the total within-cluster sum of squares (WCSS) changes as k increases.

1. Fit the Model: For a range of k values (e.g., from 1 to 10), apply the clustering algorithm (here, K-Means) to the dataset and compute the WCSS for each k.

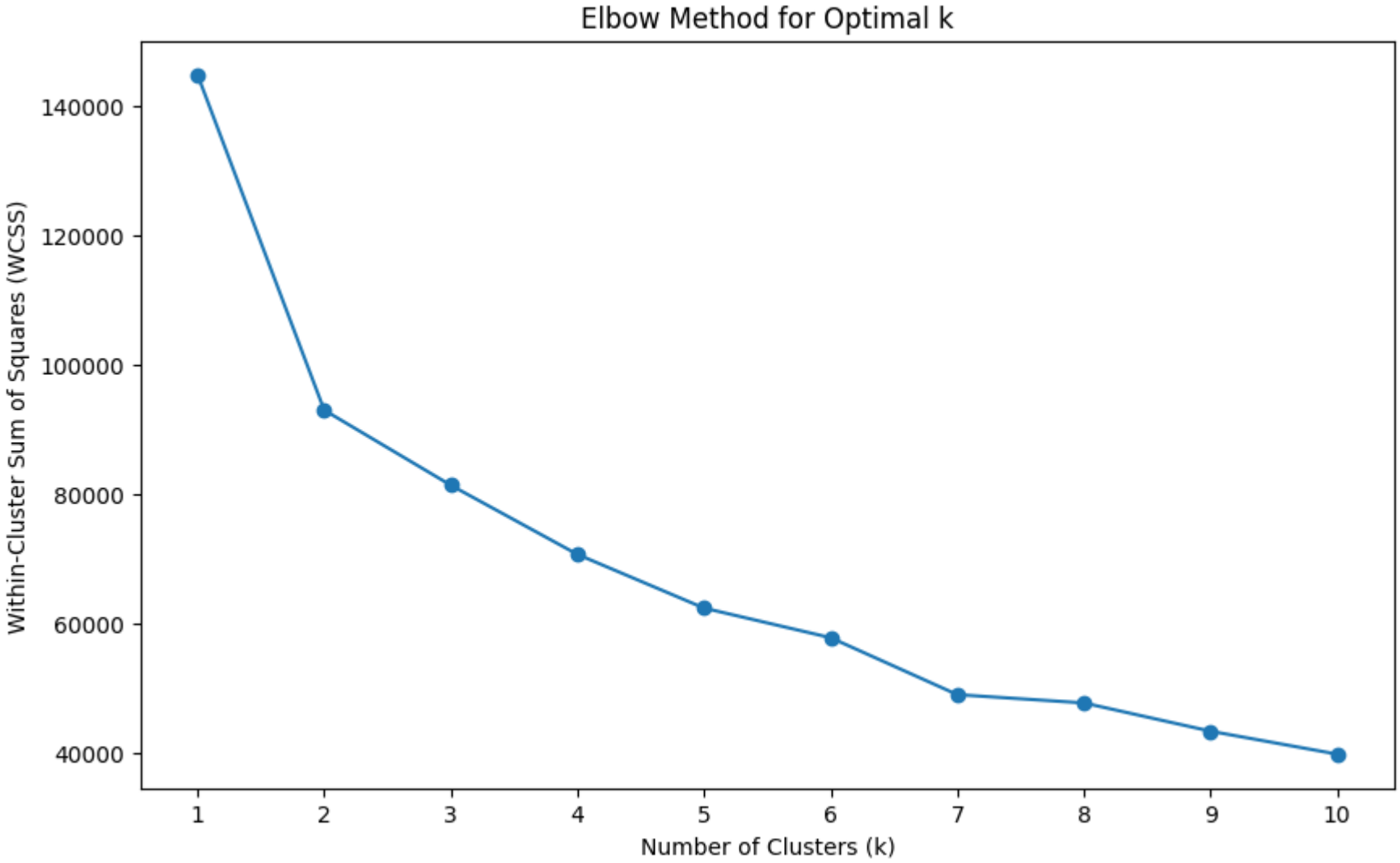
2. Calculate WCSS: WCSS measures the variance within each cluster. It is calculated as the sum of the squared distances between each data point and the centroid of its assigned cluster. A lower WCSS indicates that the data points within each cluster are closer to the centroid.

3. Plot the Results: Create a plot with the number of clusters (k) on the x-axis and the corresponding WCSS on the y-axis.

4. Identify the Elbow Point: As the number of clusters (k) increases, the WCSS generally decreases. However, at a certain point, the

rate of decrease slows down, forming an "elbow" shape in the plot of WCSS versus k. The optimal k value is typically chosen at this elbow point, which indicates a suitable balance between minimizing WCSS and avoiding overfitting by adding too many clusters.

In [ ]:
```
# Define a range of possible cluster numbers to test (e.g., 1 to 10)
k_range = range(1, 11)
wcss = []  # List to store the WCSS (Within-Cluster Sum of Squares) for each k

# Run KMeans for each k and calculate WCSS
for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(mfcc_reduced_df)
    wcss.append(kmeans.inertia_)  # Inertia is the WCSS for KMeans

# Plot the WCSS values for each k
plt.figure(figsize=(10, 6))
plt.plot(k_range, wcss, marker='o', linestyle='-')
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Within-Cluster Sum of Squares (WCSS)")
plt.title("Elbow Method for Optimal k")
plt.xticks(k_range)
plt.show()
```



2.2 Silhouette Score Evaluation

The Silhouette Score is a popular technique for evaluating the quality of clustering. This method provides insight into how well-separated and distinct the clusters are.

1. Calculation for a data point i:

- $a(i)$: Calculate the average distance from point i to all other points in the same cluster (intra-cluster distance).
- $b(i)$: Calculate the average distance from point i to all points in the nearest cluster (inter-cluster distance).
- The Silhouette Score $s(i)$ for point i is then calculated as:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

2. Aggregate Score

- The overall Silhouette Score for the clustering solution is the average of the individual scores for all points in the dataset.

3. Interpretation:

- A score close to 1 indicates that the sample is well-matched to its own cluster and poorly matched to neighboring clusters.
- A score close to 0 indicates overlapping clusters.
- A score close to -1 suggests that samples may be in the wrong clusters.

In [ ]:
```
# Set the optimal number of clusters
optimal_k = 5  # Elbow Point

# Run K-Means clustering with the optimal number of clusters
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
cluster_labels = kmeans.fit_predict(mfcc_reduced_df)

# Calculate the Silhouette Score
silhouette_avg = silhouette_score(mfcc_reduced_df, cluster_labels)
print(f"Silhouette Score for {optimal_k} clusters: {silhouette_avg:.4f}")
```

```
Silhouette Score for 5 clusters: 0.4147
```

2.3 Cluster Implementation

In [ ]:
```
# Add the cluster labels to a copy of the original DataFrame
mfcc_clustered_df = mfcc_reduced_df.copy()
mfcc_clustered_df['Cluster'] = cluster_labels
```

```
cluster_counts = mfcc_clustered_df['Cluster'].value_counts()
print("Cluster counts:\n", cluster_counts)
```

```
Cluster counts:
 Cluster
0    3454
1    1650
4     512
3     344
2     335
Name: count, dtype: int64
```

2.4 Cluster Initialization

Compare different initialization methods for K-Means

1. Random Initialization

- Randomly selects k data points from the dataset as the initial centroids.
- Can lead to suboptimal clustering results, as it may result in centroids that are too close to each other or in poor locations, potentially causing the algorithm to converge to local minima and increase the overall computational time.

2. KMeans++

- Selects initial centroids more strategically, ensuring they are spaced out by choosing the first centroid randomly and subsequent centroids based on their distance from the already chosen centroids.
- Generally leads to faster convergence and helps avoid poor local minima, resulting in better clustering performance compared to random initialization.

In [ ]:
```
# K-Means with Random Initialization
kmeans_random = KMeans(n_clusters=optimal_k, init='random', random_state=42)
labels_random = kmeans_random.fit_predict(mfcc_reduced_df)
silhouette_random = silhouette_score(mfcc_reduced_df, labels_random)

# K-Means with K-Means++ Initialization
kmeans_kmeans_plus = KMeans(n_clusters=optimal_k, init='k-means++', random_state=42)
labels_kmeans_plus = kmeans_kmeans_plus.fit_predict(mfcc_reduced_df)
silhouette_kmeans_plus = silhouette_score(mfcc_reduced_df, labels_kmeans_plus)
```

```
# Output the Silhouette Scores
print(f"Silhouette Score with Random Initialization: {silhouette_random:.4f}")
print(f"Silhouette Score with K-Means++ Initialization: {silhouette_kmeans_plus:.4f}")
```

```
Silhouette Score with Random Initialization: 0.2912
Silhouette Score with K-Means++ Initialization: 0.4147
```

3. Cluster Visualization

3.1 Dimensionality Reduction

Principal Component Analysis (PCA)

A dimensionality reduction technique used to transform a high-dimensional dataset into a lower-dimensional space while preserving as much variance (information) as possible.
Seeks to identify the directions (principal components) in which the data varies the most. These directions are orthogonal (perpendicular) to each other.

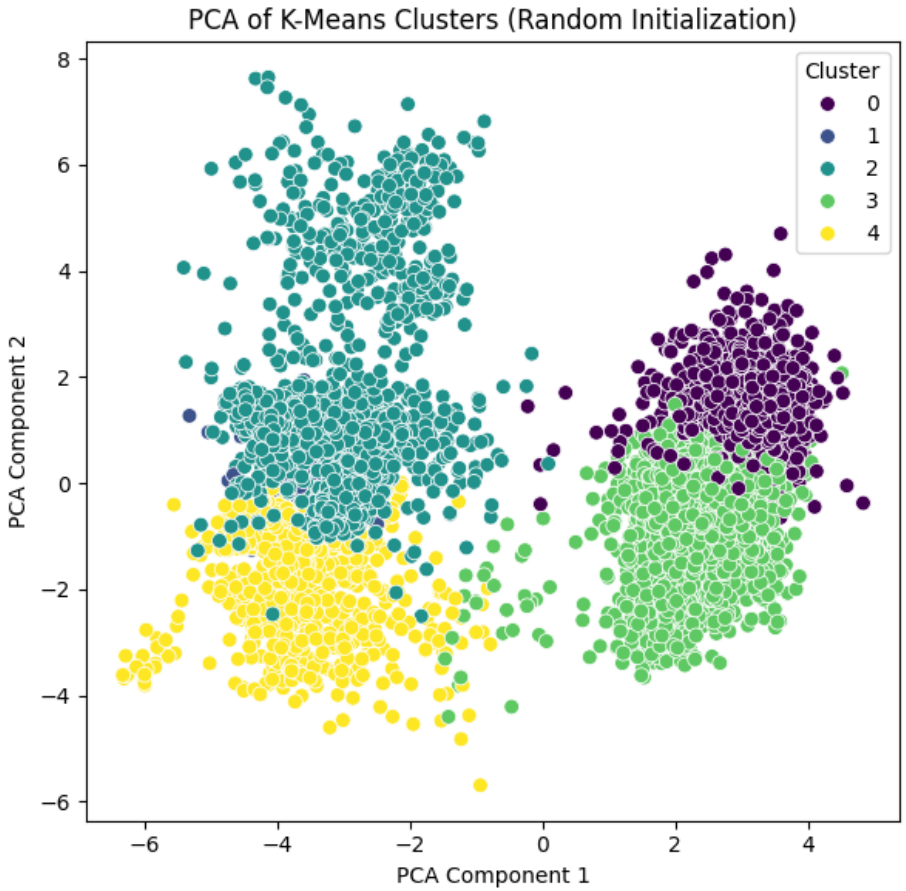
In [ ]:
```
pca = PCA(n_components=2)

# PCA for K-Means with Random Initialization
pca_result_random = pca.fit_transform(mfcc_reduced_df)
pca_df_random = pd.DataFrame(data=pca_result_random, columns=['PCA1', 'PCA2'])
pca_df_random['Cluster'] = labels_random  # Add cluster labels

# PCA for K-Means with K-Means++ Initialization
pca_result_kmeans_plus = pca.fit_transform(mfcc_reduced_df)
pca_df_kmeans_plus = pd.DataFrame(data=pca_result_kmeans_plus, columns=['PCA1', 'PCA2'])
pca_df_kmeans_plus['Cluster'] = labels_kmeans_plus  # Add cluster labels
```

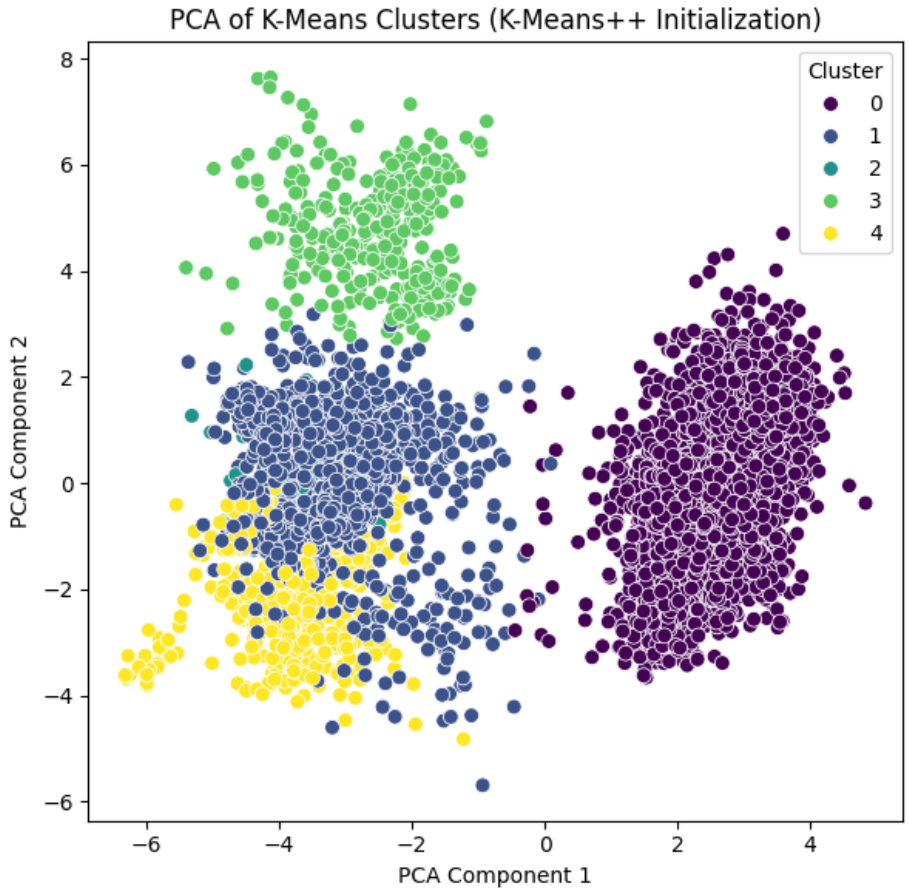
3.2 Cluster Plots

Visualize the clusters using 2D scatter plots.

In [ ]:
```
# Plot the PCA results for Random Initialization
```

```
plt.figure(figsize=(12, 6))

# Plot for Random Initialization
plt.subplot(1, 2, 1)
sns.scatterplot(data=pca_df_random, x='PCA1', y='PCA2', hue='Cluster', palette='viridis', s=50)
plt.title("PCA of K-Means Clusters (Random Initialization)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.legend(title='Cluster')

# Plot for K-Means++ Initialization
plt.subplot(1, 2, 2)
sns.scatterplot(data=pca_df_kmeans_plus, x='PCA1', y='PCA2', hue='Cluster', palette='viridis', s=50)
plt.title("PCA of K-Means Clusters (K-Means++ Initialization)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.legend(title='Cluster')

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

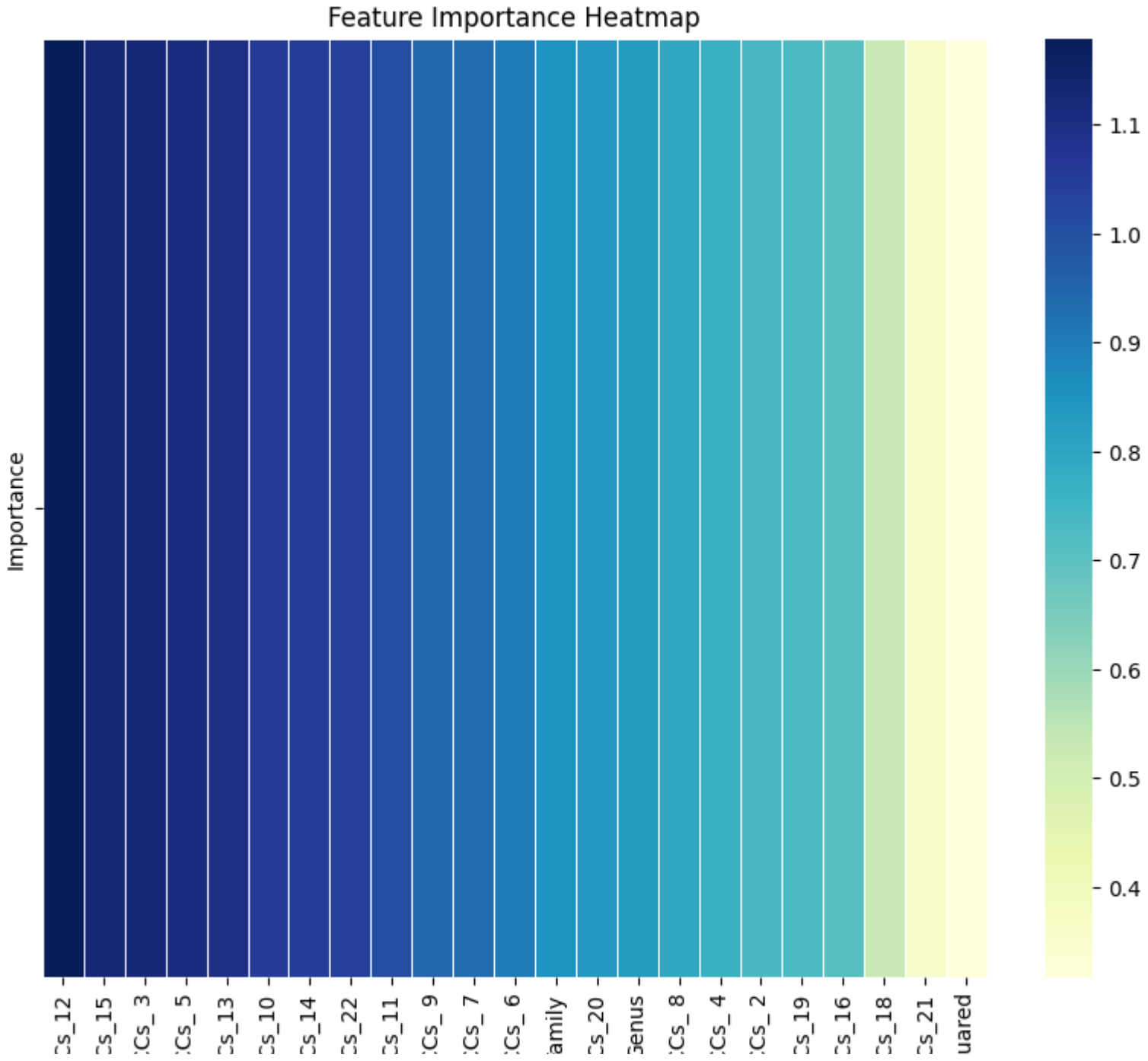




3.3 Feature Contribution to Clustering

In [ ]:
```
# Calculate the centroids of the clusters
centroids = kmeans_kmeans_plus.cluster_centers_

# Calculate the absolute mean difference of each feature across centroids
feature_importance = np.abs(centroids).mean(axis=0)

# Create a DataFrame with feature names and their importance scores
feature_importance_df = pd.DataFrame({
    'Feature': mfcc_reduced_df.columns,
    'Importance': feature_importance
```

```
})

# Sort features by importance
feature_importance_df = feature_importance_df.sort_values('Importance', ascending=False)

plt.figure(figsize=(10, 8))
sns.heatmap(feature_importance_df.set_index('Feature').T,
            cmap="YlGnBu",
            cbar=True,
            linewidths=.5)
plt.title("Feature Importance Heatmap")
plt.show()

print("\nTop 10 Most Important Features:")
feature_importance_df.head(10)
```


Feature Importance Heatmap
Importance
1.1
1.0
0.9
0.8
0.7
0.6
0.5
0.4

Top 10 Most Important Features:

Out[ ]:

	Feature	Importance
10	MFCCs_12	1.179132
13	MFCCs_15	1.127437
1	MFCCs_ 3	1.123596
3	MFCCs_ 5	1.114503
11	MFCCs_13	1.095265
8	MFCCs_10	1.048030
12	MFCCs_14	1.046022
19	MFCCs_22	1.039019
9	MFCCs_11	1.004366
7	MFCCs_ 9	0.949248

4. Cluster Evaluation Metrics

Davies-Bouldin Index

Metric used to evaluate the quality of clustering algorithms by measuring the separation and compactness of clusters.

- A lower Davies-Bouldin Index indicates better clustering quality.
- The average similarity ratio (the ratio of the within-cluster scatter (compactness) to the between-cluster separation) of each cluster with the most similar cluster.

$$DB = \frac{1}{k} \sum_{i=1}^{k} \max_{j \neq i} \left(\frac{\sigma_i + \sigma_j}{d_{ij}} \right)$$

Where σ_i is the scatter of cluster i, d_{ij} is the distance between cluster centroids i and j, and k is the number of clusters.

Calinski-Harabasz Index (Variance Ratio Criterion)

Measures the quality of clustering based on the ratio of between-cluster dispersion to within-cluster dispersion.

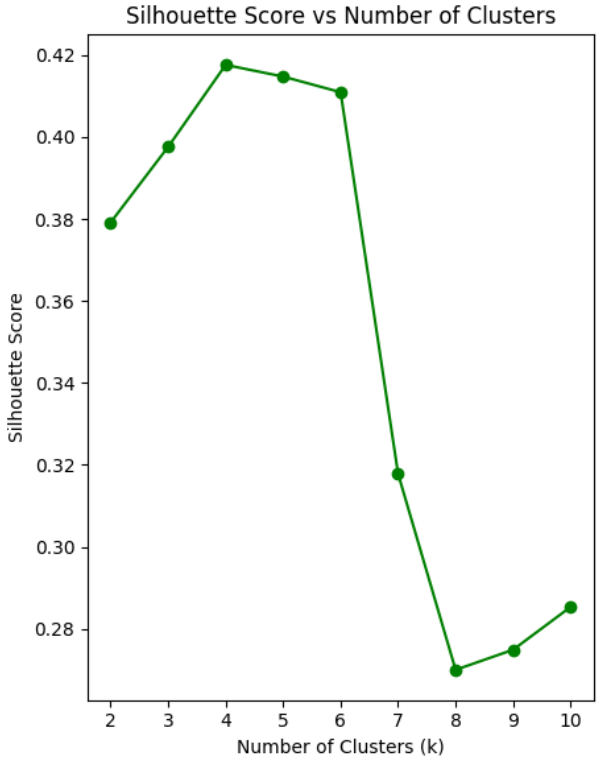
- A higher Calinski-Harabasz Index indicates better clustering performance.
- The ratio of the sum of between-cluster dispersion to the sum of within-cluster dispersion.

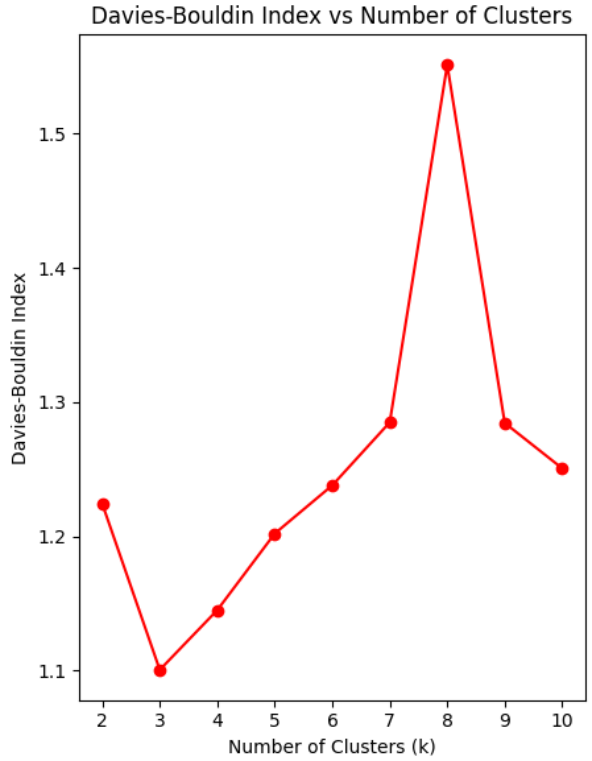
$$CH = \frac{B_k/(k-1)}{W_k/(n-k)}$$

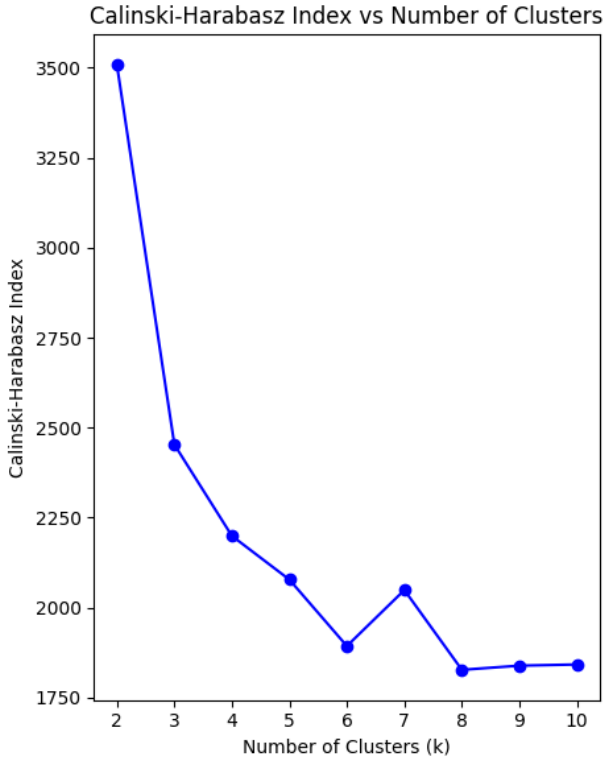
Where B_k is the between-cluster variance, W_k is the within-cluster variance, k is the number of clusters, and n is the total number of samples.

In [ ]:
```
# Lists to store metrics for each k
db_scores = []
ch_scores = []
sh_scores = []

# Define a range of possible cluster numbers to test (e.g., 1 to 10)
k_range = range(2, 11)

for k in k_range:
    # Fit K-Means clustering
    kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42)
    labels = kmeans.fit_predict(mfcc_reduced_df)

    # Calculate Davies-Bouldin Index
    db_index = davies_bouldin_score(mfcc_reduced_df, labels)
    db_scores.append(db_index)

    # Calculate Calinski-Harabasz Index
```

```
    ch_index = calinski_harabasz_score(mfcc_reduced_df, labels)
    ch_scores.append(ch_index)

    # Calculate Silhouette Score
    silhouette_avg = silhouette_score(mfcc_reduced_df, labels)
    sh_scores.append(silhouette_avg)

# Plotting the metrics
plt.figure(figsize=(14, 6))

# Silhouette Score
plt.subplot(1, 3, 1)
plt.plot(k_range, sh_scores, marker='o', color='g')
plt.title("Silhouette Score vs Number of Clusters")
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Silhouette Score")
plt.xticks(k_range)


# Davies-Bouldin Index
plt.subplot(1, 3, 2)
plt.plot(k_range, db_scores, marker='o', color='r')
plt.title("Davies-Bouldin Index vs Number of Clusters")
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Davies-Bouldin Index")
plt.xticks(k_range)

# Calinski-Harabasz Index
plt.subplot(1, 3, 3)
plt.plot(k_range, ch_scores, marker='o', color='b')
plt.title("Calinski-Harabasz Index vs Number of Clusters")
plt.xlabel("Number of Clusters (k)")
plt.ylabel("Calinski-Harabasz Index")
plt.xticks(k_range)

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







In [ ]:
```
# Evaluate Clustering for KMeans
kmeans_silhouette = silhouette_score(mfcc_reduced_df, labels_kmeans_plus)
kmeans_db_index = davies_bouldin_score(mfcc_reduced_df, labels_kmeans_plus)
kmeans_ch_index = calinski_harabasz_score(mfcc_reduced_df, labels_kmeans_plus)

# Output the evaluation metrics
print(f"Silhouette Score for K-Means Clustering: {kmeans_silhouette:.2f}")
print(f"Davies-Bouldin Index for K-Means Clustering: {kmeans_db_index:.2f}")
print(f"Calinski-Harabasz Index for K-Means Clustering: {kmeans_ch_index:.2f}")
```

```
Silhouette Score for K-Means Clustering: 0.41
Davies-Bouldin Index for K-Means Clustering: 1.20
Calinski-Harabasz Index for K-Means Clustering: 2077.90
```

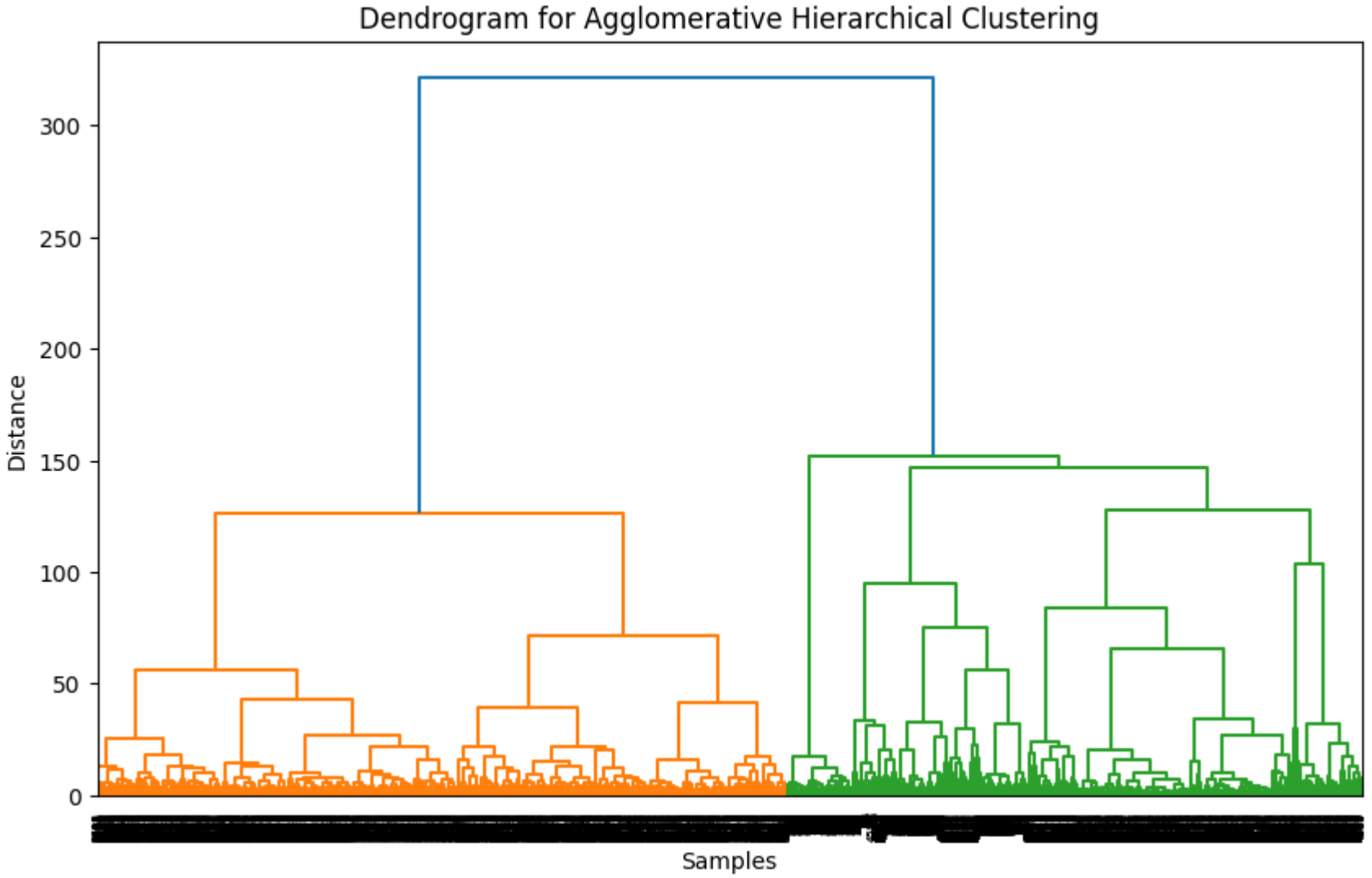
5. Comparison with other Clustering Algorithms

5.1 Agglomerative Clustering

A hierarchical clustering method that builds a tree of clusters (dendrogram) through a bottom-up approach.

- The algorithm starts with each data point as its own cluster and iteratively merges the closest pairs of clusters until a specified number of clusters is achieved or until all points are in a single cluster.
- Clusters are formed based on distance metrics, such as Euclidean distance. Different linkage criteria can be used to determine the distance between clusters, including:
  - Single Linkage: Distance between the closest points in two clusters.
  - Complete Linkage: Distance between the farthest points in two clusters.
  - Average Linkage: Average distance between all points in two clusters.
  - Ward's Method: Minimizes the variance within clusters. (Used here)

In [ ]:
```
# Apply Agglomerative Hierarchical Clustering
agg_clustering = AgglomerativeClustering(n_clusters=optimal_k)  # Use optimal_k from K-Means
agg_labels = agg_clustering.fit_predict(mfcc_reduced_df)

# Plotting Dendrogram
plt.figure(figsize=(10, 6))
dendrogram = sch.dendrogram(sch.linkage(mfcc_reduced_df, method='ward'))
plt.title("Dendrogram for Agglomerative Hierarchical Clustering")
plt.xlabel("Samples")
plt.ylabel("Distance")
plt.show()

# Evaluate Clustering
agg_silhouette = silhouette_score(mfcc_reduced_df, agg_labels)
agg_db_index = davies_bouldin_score(mfcc_reduced_df, agg_labels)
agg_ch_index = calinski_harabasz_score(mfcc_reduced_df, agg_labels)

# Output the evaluation metrics
print(f"Silhouette Score for Agglomerative Hierarchical Clustering: {agg_silhouette:.2f}")
print(f"Davies-Bouldin Index for Agglomerative Hierarchical Clustering: {agg_db_index:.2f}")
print(f"Calinski-Harabasz Index for Agglomerative Hierarchical Clustering: {agg_ch_index:.2f}")
```




```
Silhouette Score for Agglomerative Hierarchical Clustering: 0.40
Davies-Bouldin Index for Agglomerative Hierarchical Clustering: 1.32
Calinski-Harabasz Index for Agglomerative Hierarchical Clustering: 2060.79
```

5.2 DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

A density-based clustering algorithm that identifies clusters in data by examining the density of data points in a specified area.

- `eps` (ε): A distance threshold that defines the radius of neighborhood around a point.
- `min_samples` (MinPts): The minimum number of points (samples) required to form a dense region (cluster).

1. For each point in the dataset, retrieve its ε-neighborhood.
2. If the number of points in this neighborhood is greater than or equal to MinPts, it is classified as a core point and a new cluster is initiated.
3. All points within the ε-neighborhood of core points are added to the cluster (including border points).
4. The process is repeated until all points are visited.

In [ ]:
```
# Apply DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan_labels = dbscan.fit_predict(mfcc_reduced_df)

# Evaluate Clustering
dbscan_silhouette = silhouette_score(mfcc_reduced_df, dbscan_labels)
dbscan_db_index = davies_bouldin_score(mfcc_reduced_df, dbscan_labels)
dbscan_ch_index = calinski_harabasz_score(mfcc_reduced_df, dbscan_labels)

# Output the evaluation metrics
print(f"Silhouette Score for DBSCAN: {dbscan_silhouette:.2f}")
print(f"Davies-Bouldin Index for DBSCAN: {dbscan_db_index:.2f}")
print(f"Calinski-Harabasz Index for DBSCAN: {dbscan_ch_index:.2f}")
```

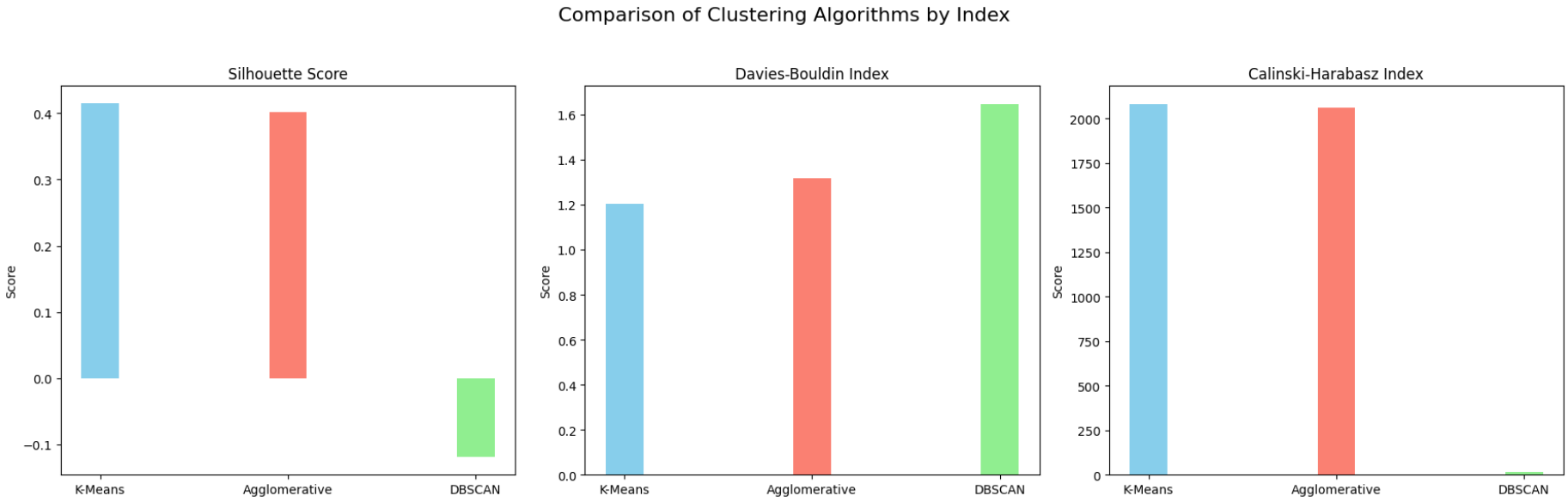
```
Silhouette Score for DBSCAN: -0.12
Davies-Bouldin Index for DBSCAN: 1.64
Calinski-Harabasz Index for DBSCAN: 16.60
```

5.3 Comparison of Cluster Results

Comparison by Index

In [ ]:
```
# Plotting indices for the three different clustering algorithms
metrics = {
    'K-Means': {
        'Silhouette Score': kmeans_silhouette,
```

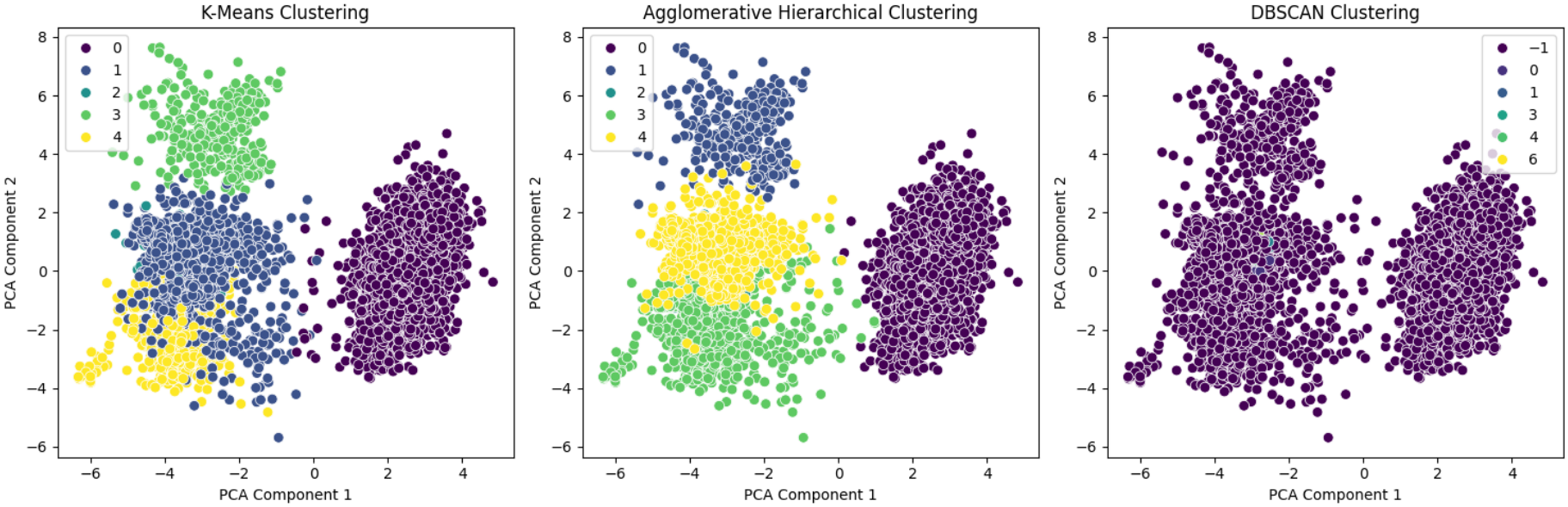
```
        'Davies-Bouldin Index': kmeans_db_index,
        'Calinski-Harabasz Index': kmeans_ch_index,
    },
    'Agglomerative': {
        'Silhouette Score': agg_silhouette,
        'Davies-Bouldin Index': agg_db_index,
        'Calinski-Harabasz Index': agg_ch_index,
    },
    'DBSCAN': {
        'Silhouette Score': dbscan_silhouette,
        'Davies-Bouldin Index': dbscan_db_index,
        'Calinski-Harabasz Index': dbscan_ch_index,
    }
}

# Prepare the data for plotting
index_names = list(metrics['K-Means'].keys())
num_algorithms = len(metrics)
x = np.arange(len(index_names))

# Create subplots
fig, axs = plt.subplots(1, 3, figsize=(18, 6))

# Loop through each index and create a bar plot
for i, index_name in enumerate(index_names):
    scores = [metrics[alg][index_name] for alg in metrics]

    axs[i].bar(x, scores, width=0.2, color=['skyblue', 'salmon', 'lightgreen'])
    axs[i].set_title(index_name)
    axs[i].set_xticks(x)
    axs[i].set_xticklabels(metrics.keys())
    axs[i].set_ylabel('Score')

# Customizing the overall plot
plt.suptitle('Comparison of Clustering Algorithms by Index', fontsize=16)
plt.tight_layout(rect=[0, 0.03, 1, 0.95])  # Adjust layout to fit the title
plt.show()
```



Visualization of the clustering results

In [ ]:
```
# Create a PCA for visualization
pca = PCA(n_components=2)
pca_result = pca.fit_transform(mfcc_reduced_df)

# Plot K-Means results
plt.figure(figsize=(15, 5))

# K-Means Clustering
plt.subplot(1, 3, 1)
sns.scatterplot(x=pca_result[:, 0], y=pca_result[:, 1], hue=labels_kmeans_plus, palette='viridis', s=50)
plt.title("K-Means Clustering")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")

# Agglomerative Clustering
plt.subplot(1, 3, 2)
sns.scatterplot(x=pca_result[:, 0], y=pca_result[:, 1], hue=agg_labels, palette='viridis', s=50)
plt.title("Agglomerative Hierarchical Clustering")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
```

```
# DBSCAN
plt.subplot(1, 3, 3)
sns.scatterplot(x=pca_result[:, 0], y=pca_result[:, 1], hue=dbscan_labels, palette='viridis', s=50)
plt.title("DBSCAN Clustering")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")

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



Strengths and Weaknesses of Each Algorithm

Algorithm	Strengths	Weaknesses
K-Means	Fast and efficient for large datasets.	Sensitive to initialization (although K-Means++ helps).
	Works well with spherical clusters.	Requires the number of clusters to be specified beforehand.
	Simple to understand and implement.	Assumes clusters are of similar sizes and densities.

Algorithm	Strengths	Weaknesses
Agglomerative Hierarchical Clustering	Does not require a predefined number of clusters (can cut the dendrogram). Produces a hierarchy of clusters, providing insights into data structure. Can capture clusters of different shapes and sizes.	Computationally expensive for large datasets ($O(n^3)$). Sensitive to noise and outliers.
DBSCAN	Can find arbitrarily shaped clusters and is robust to noise. Does not require specifying the number of clusters a priori.	Requires careful tuning of parameters (epsilon and min_samples). May struggle with clusters of varying densities. Performance can degrade with high-dimensional data.

6. Analysis and Report

- For initialization of KMeans, the KMeans++ initialization performed better than Random initialization (as expected).

Initialization Method	Silhouette Score
Random Initialization	0.2912
K-Means++ Initialization	0.4147

- A negative Silhouette Score for the DBSCAN algorithm indicates that it struggles to cluster the data points effectively with the current parameter settings (default parameters used). Meanwhile, Agglomerative clustering shows performance comparable to that of K-Means. Although DBSCAN achieves a favorable Davies-Bouldin Index for this dataset, it underperforms when evaluated with the other two metrics.

Clustering Algorithm	Silhouette Score	Davies-Bouldin Index	Calinski-Harabasz Index
K-Means	0.41	1.20	2077.90
Agglomerative Hierarchical Clustering	0.40	1.32	2060.79
DBSCAN	-0.12	1.64	16.60