Phase 2

Voice Authentication

Introduction to Machine Learning

Instructors: Dr. Dehqani, Dr. Aarabi

Authors:

Parsa Daghigh (810101419)

Kosar Shiri Jafarzadeh (810101456)

Mohammad Mahdi Samadi (810101465)

Parisa Yahyapour Fatideh (810101551)

Table of Content

Introduction to voice authentication

Description

Voice authentication, also known as voice biometrics, is a technology that uses a person's voice as a unique identifier to verify their identity. This is achieved by analyzing various characteristics of a person’s voice, such as pitch, tone, and speech patterns, which are unique to each individual. It is commonly used in applications like secure access to devices, customer authentication in call centers, and more.

Voice authentication differs from technologies like speech recognition, speech-to-text systems, and voice assistants such as Siri. While these tools focus on understanding and processing spoken language to convert it into text or execute specific commands, voice authentication emphasizes identifying the unique vocal characteristics of an individual to verify their identity. Unlike speech-to-text systems that rely solely on linguistic content, voice authentication analyzes features such as pitch, tone, and vocal patterns, making it a biometric solution rather than a linguistic one.

Importance

Voice authentication has become a vital tool in industries such as call centers, banking, healthcare, and government services, where secure and efficient identity verification is essential. By leveraging the unique characteristics of an individual’s voice, this technology provides an added layer of protection against unauthorized access and fraudulent activities.

Unlike traditional passwords or PINs, which can be forgotten, stolen, or hacked, a person’s voice is inherently unique and difficult to replicate, making voice authentication a reliable and secure method. Organizations can further strengthen their security protocols by integrating voice authentication with multi-factor authentication systems, ensuring a robust defense against cyber threats.

Additionally, voice recognition systems are highly user-friendly, offering a seamless and efficient authentication process. Users can quickly verify their identity without the need for remembering complex passwords or carrying physical tokens, enhancing convenience while reducing the risk of errors.

Usage

Below, we delve into 10 of the most promising applications of voice biometrics, with expanded descriptions of each use case:

1. Call Center Authentication: Voice biometrics is revolutionizing customer interactions in call centers by enabling seamless and secure authentication. Instead of relying on traditional methods like PIN codes, passwords, or security questions, voice biometrics verifies customers through their unique vocal characteristics. This not only enhances security but also reduces call handling time, improves customer satisfaction, and minimizes the risk of impersonation and fraud.
2. Financial Services: Financial institutions are leveraging voice biometrics to bolster account security and streamline processes such as account verification and transaction approvals. By using voice as an additional layer of security, banks can reduce the likelihood of fraudulent activities while providing a more convenient authentication method for customers, especially in mobile banking and phone-based transactions.
3. Healthcare: In the healthcare industry, voice biometrics is used for patient identification, ensuring that only authorized individuals can access sensitive medical records or systems. This technology enhances data privacy and security in compliance with regulations like HIPAA. It also simplifies patient onboarding and authentication processes, particularly in telemedicine services.
4. Government: Government agencies utilize voice biometrics for critical functions such as border control, passport verification, and secure access to e-government services. By integrating voice biometrics, authorities can streamline identification processes, improve accuracy, and prevent identity theft in sensitive applications like immigration and national security.
5. Education: In the education sector, voice biometrics is used for student identification during online exams and assessments, ensuring academic integrity. Additionally, it can facilitate personalized learning experiences by enabling voice-based interactions in educational platforms, making learning more accessible and engaging.
6. Gender Recognition: Gender recognition in voice authentication refers to the process of identifying the gender of a speaker based on vocal characteristics. This is achieved by analyzing features such as pitch, tone, speech patterns, and resonance, which tend to differ between male and female voices due to physiological and behavioral factors.

While not directly related to verifying identity, gender recognition is often used in conjunction with voice authentication systems to enhance personalization, improve system performance, or aid in demographic analysis.

Closed-Set Authentication

A biometric task which involves identifying an unknown individual who is confirmed to exist within a pre-registered database. The system's objective is to match the input voice with one of the stored voices and accurately determine the person's identity. The effectiveness of the system is evaluated based on how often the correct match appears at the top of the system's ranked list of potential identities. Higher accuracy in ranking the correct individual reflects better system performance.So, closed-set authentication assumes that all possible users of the system are known during the enrollment phase. This means the system has access to a predefined database of enrolled identities, and every input is matched against this database. If we want to consider a sequence of steps for this method, we can consider the steps below:

1. A user provides their biometric data.
2. The system compares the input with all stored profiles in the database.
3. The system determines which enrolled user matches the input.

Open-Set Authentication

Open-set authentication is a biometric recognition paradigm designed to identify whether an individual attempting authentication is part of a known group (enrolled users) or a new, unknown user. This is particularly relevant in systems that need to handle both authorized users and prevent access from unauthorized individuals. Unlike closed-set authentication, open-set systems include the concept of rejection, where the input may not correspond to any existing profile in the database. If we want to consider a sequence of steps for this method, we can consider the steps below:

1. Users provide their biometric data to create a unique profile stored in the system's database. These profiles represent the "known" group.
2. A user provides a biometric sample during authentication.
3. The system compares the sample against all stored profiles in the database.

4. If a match is found within a confidence threshold, the user is authenticated as a known individual.
5. If no match meets the threshold, the system rejects the input, classifying the user as “unknown.”

Closed-Set vs. Open-Set Authentication

Closed-set and open-set authentication can be compared across several key aspects. In the following, we will explore and contrast these two methods in six distinct areas, highlighting their differences.

Goal

The primary goal in closed-set authentication is to identify an individual from a fixed pool of known identities. If the system finds a match within the enrolled profiles, the individual is authenticated. This system does not reject users unless they are not in the database at all. The goal of open-set authentication is not only to identify known users but also to reject unknown users effectively. If the system cannot find a match in the database within the set threshold, it will classify the user as "unknown" and deny access. This provides an added layer of security by distinguishing legitimate users from potential intruders.

Complexity

Closed-set authentication tends to be simpler since it operates with a known set of enrolled users. The system’s only task is to match the incoming biometric data to the database and verify the identity. There is no need for a rejection process for unknown users, making the system easier to manage and scale in controlled environments. Open-set authentication is more complex, as it requires the system not only to identify known users but also to reject unknown users accurately. This added layer of functionality increases the system's computational complexity, requiring sophisticated algorithms to balance false rejections and false acceptances effectively. Open-set systems must manage both recognition and rejection thresholds, which can involve more advanced machine learning models or decision algorithms.

User Base

In closed-set authentication, the system assumes that all potential users are pre-enrolled and known to the system. The set of users is fixed, and only individuals whose biometric data is stored in the system can be authenticated. For example, in a corporate building access system, only employees enrolled in the system can gain access. Open-set authentication, on the other hand, works in environments where the system must distinguish between authorized users and unknown individuals. In this case, the system checks if the input biometric matches one of the known enrolled profiles. If there's no match, the system will reject the input as from an unauthorized user, making it suitable for environments where there is a mix of both known and unknown participants, like airport security systems.

Implementation of Closed-Set Authentication

Collect and Register Identities

To implement closed-set authentication, first we have to determine the feature to be used for authentication, such as passwords, biometrics (e.g., fingerprints or facial recognition), or digital keys. For user enrollment, we collect and securely store passwords if a password-based system is used or we should capture biometric data using sensors and convert it into templates for biometric authentication, or register digital credentials for key-based methods. Preprocessing biometric data to normalize, clean, and encode is an essential step for ensuring consistency and reliability for future matching.

Implement the Authentication Process

The authentication process begins with the user providing their credentials, such as entering a password, scanning a fingerprint, or presenting another form of identity. This input undergoes preprocessing to prepare it for further analysis, ensuring consistency with stored data formats. Key features are extracted from the input to create a comparable representation, such as hashing passwords or encoding biometric data. The extracted data is then compared to the stored records using appropriate matching algorithms, such as hash verification for passwords or similarity metrics for biometrics. A decision is made based on the comparison: if a match is found within an acceptable threshold, access is granted; otherwise, authentication fails.

Decision Logic

1. Exact Match: If the input matches a stored identity within an acceptable threshold, the user is authenticated.
2. Failure: If no match is found, deny access.
3. Thresholding: For biometric systems, implement thresholds for matching scores to account for minor variations

Implementation of Open-Set Authentication

Collect and Register Known Identities

To construct a database of known identities for an open-set authentication system, the process begins with identifying unique characteristics that can reliably distinguish each user. These may include biometric traits such as facial features or fingerprints, behavioral patterns like typing dynamics, or digital credentials such as access tokens. During the enrollment phase, representative data is collected from users and subjected to preprocessing to ensure it is accurate, normalized, and suitable for further analysis. Key features are then extracted from the processed data, such as embeddings for biometrics or secure hashes for credentials. The resulting templates are securely stored in a dedicated database, with robust encryption and strict access controls implemented to ensure data confidentiality and integrity. This database serves as the foundation for verifying user identities during authentication.

Establish Open-Set Thresholds

The system must define thresholds to determine whether a user belongs to the known set. Matching algorithms, such as cosine similarity or Hamming distance, are used to compare user inputs with stored templates. Two thresholds are defined: an acceptance threshold for identifying known users and a rejection threshold for unknown users. Inputs falling between these thresholds can be flagged for manual review or additional verification, ensuring a balance between security and accessibility.

Authentication Process

The authentication process involves several stages. First, user inputs, such as a fingerprint scan or facial image, are captured in real-time. These inputs are preprocessed to match the format of stored templates. Key features are extracted from the processed data and compared against stored templates using similarity metrics. Based on the comparison results:

- If the similarity score exceeds the acceptance threshold, the user is authenticated as a known identity.
- If the score falls below the rejection threshold, the user is classified as unknown, and access is denied.
- Scores in the ambiguous range trigger further checks, such as secondary verification.

Handling Unknown Users

The system incorporates mechanisms to manage unknown users effectively. When an input is classified as unknown, the system can escalate the event for manual review, log the details for further investigation, or implement dynamic learning workflows. This allows for potential expansion of the database with new user templates, subject to administrative approval, ensuring adaptability in dynamic environments.

Closed-Set and Open-Set in Voice Authentication

The two key approaches to voice authentication—closed-set and open-set—are employed based on the nature of the system, user base, and security requirements.

Closed-Set

In closed-set voice authentication, the system is designed to authenticate users whose voice profiles are pre-enrolled in a database. The system assumes that all users attempting to authenticate are part of a fixed, known group. We can divide the process into two phases:

1. Enrollment: During the enrollment phase, users provide their voice samples, and these are processed into unique biometric templates. The system extracts distinct features from the voice, such as pitch, tone, speech cadence, and pronunciation patterns, which form the voiceprint (or template) for each user.

2. Authentication: When a user attempts to authenticate, their voice is captured and processed. The system extracts similar features from the new sample and compares it to the stored templates. If the features match closely with one of the enrolled profiles, the user is authenticated. If no match is found, access is denied.

Applications

1. Corporate Access Control: Closed-set voice authentication is commonly used for access control in corporate environments where only authorized employees are allowed access to sensitive areas or information.
2. Customer Service: Many customer service centers use closed-set voice authentication to verify the identity of known customers, such as in banking or insurance.

Advantages

1. Simpler Implementation: Since the system only needs to match known users, it is less computationally complex and easier to implement.
2. Fast Authentication: Closed-set systems typically provide faster recognition since they only have to compare against a small set of profiles.

Limitations

1. Inflexibility: If a new user needs to be added, the database must be updated with their voiceprint. The system cannot handle unknown users.
2. Limited Security: While fast and efficient, closed-set authentication is vulnerable to attacks where an unauthorized person has access to a legitimate user's voiceprint.

Open-Set

In open-set voice authentication, the system is designed to both authenticate known users and reject unknown users. It not only recognizes users from a pre-enrolled set but also has the capability to identify when a speaker is not registered in the system, denying them access. We can divide the process into three phases:

1. Enrollment: Just like closed-set authentication, users provide their voice samples during the enrollment phase, creating biometric templates.

2. Authentication: When a user provides their voice sample for authentication, the system extracts features and compares them with the stored templates. However, if no match is found, the system does not simply deny access—it actively rejects the unknown user.
3. Threshold-Based Decision: The system evaluates the similarity of the input voice sample with all stored templates. A threshold is set for the similarity score, and if the score of the best match falls below the threshold, the system classifies the speaker as "unknown" and denies access. If the score is above the threshold, the system authenticates the user as known.

Applications

1. Public Security Systems: Open-set voice authentication is widely used in systems that need to differentiate between authorized individuals and intruders, such as in airports, border control, or government security systems.
2. Banking and Finance: In telephone banking, open-set voice authentication helps prevent fraud by rejecting callers who are not in the system, while still allowing enrolled customers to authenticate.

Advantages

1. Enhanced Security: Since the system can reject unknown users, open-set authentication is more secure than closed-set systems, making it ideal for high-risk applications.
2. Greater Flexibility: It can handle both known and unknown users, making it suitable for dynamic environments where new users might need to be authenticated.

Limitations

1. Increased Complexity: Implementing open-set authentication requires more sophisticated algorithms to manage the identification and rejection of unknown users.
2. Higher Computational Cost: Because it involves checking against all stored templates and determining the threshold for rejection, it requires more resources and may be slower than closed-set systems.

Challenges in Voice Authentication

Main Challenges

Diversity of Languages and Accents

This challenge we face here arises because speech characteristics vary significantly across linguistic and cultural groups. These variations affect the system's ability to recognize and authenticate voices accurately.

Languages differ in phonemes and syllable structures, For example, Chinese depend on pitch variations to extract meaning, whereas some languages like English do not. This creates complexity for systems that lack sensitivity to this feature in their training data.

Speakers of the same language can have different accents based on their region, for example each part of Iran may speak Persian with different accents.

Many speakers alternate between two or more languages in a single conversation like us when we use English and Persian together.Most datasets that are available are for major languages like english but others may lack the proper dataset which lead to poorer performance to the major ones.

High Costs of Data Collection

Implementing voice recognition technology can cost a lot of money since it requires both hardware and software support.

Note that in this task there is a trade off if the company finds out this process would be helpful because its overhead is venial.

If we want to reach high quality voice recognition our models need to have larger datasets which is associated with additional cost.

The associated costs may come from specialized recording equipment, secure storage, Data cleaning and quality control expenses and additional data for regions accents.

Data Privacy and Sensitivity

As we already know people’s voices are a part of their biometric and it's kind of like their fingerprint since voices are distinct as well so it's an individual’s privacy.
Then how can we gather the data we need for the dataset?
Often people don't want to be involved in this process without any incentives and if we use their voice against their will this would raise a huge privacy invasion.
Variability in Voice Characteristics
Voice characteristics such as pitch, tone, and resonance are actually very different among a set of people and this diversity is due to biological, physiological, and social factors.
For example, if we have a child, finding out the gender would be challenging, on the other hand if we try to recognize the gender by the fixed frequency. It will cause some error when the age, noise and other environmental things apply to this.

Overlap in Male and Female Vocal Features

Although pitch is a strong indicator of gender, other acoustic features like formant frequencies and harmonics often overlap between male and female speakers.

Acoustic and Environmental Noise

Background noise, poor microphone quality, and other acoustic distortions often mask critical vocal features like pitch and harmonics. These issues can drastically reduce the accuracy of gender classification models.
In a noisy environment, such as a cafe, the system might fail to distinguish between male and female voices due to interference from ambient sounds.

Physiological and Psychological Factors

Temporary or permanent changes in the voice due to health, emotions, or other psychological factors can make gender classification unreliable.
A cold, throat infection, stress, excitement, or sadness often can lead to misclassification.

Dataset Bias

Many voice recognition systems are trained on datasets that lack diversity in gender, age, ethnicity, and linguistic background.This problem will lead to poorer performance for underrepresented groups.

Solutions

Multilingual and Accent-Agnostic Models

These models are specifically designed to handle a wide range of languages and accents. They are trained on datasets that include diverse linguistic and regional varieties, ensuring broad coverage.

By incorporating phonetic and linguistic diversity, these models reduce the bias toward dominant languages or accents. Techniques like phoneme-based modeling and language embeddings are commonly used.

Transfer Learning and Accent Adaptation

Transfer learning uses pre-trained models and fine-tunes them with smaller datasets from underrepresented accents. Accent adaptation techniques allow models to dynamically adjust to new accents with minimal additional training.

It reduces the resource requirements for training while improving performance on less common accents and languages.

Data Augmentation

Methods such as speed perturbation, pitch shifting, and adding synthetic noise create new data points from existing recordings.

Augmentation significantly reduces the need for collecting large datasets by simulating diverse speaking conditions, accents, and environments.

Synthetic Data Generation

Advanced text-to-speech (TTS) systems are used to create realistic voice datasets. GANs (Generative Adversarial Networks) and neural vocoders are often employed to generate high-quality synthetic speech.

It minimizes the costs of collecting and annotating real-world voice data, especially for underrepresented groups.

Federated Learning (FL)

FL enables machine learning models to be trained across multiple devices without transferring data to a central server. Only model updates (e.g., gradients) are shared.

This method ensures that sensitive voice data remains on local devices, maintaining user privacy while contributing to model training.

Speaker Normalization

Techniques like Vocal Tract Length Normalization (VTLN) adjust speech features to reduce variability caused by anatomical differences in speakers.

This equalizes the acoustic representation of speech, making it easier for models to learn consistent patterns.

Multi-Level Feature Representation

Using deep learning models that extract features at multiple levels, such as pitch, energy, and spectral patterns, helps differentiate subtle variations between genders.

By capturing higher-order features, the model can discern gender-related traits even when there is significant overlap.

Ensemble Models for Classification

Combining the predictions of multiple classifiers reduces uncertainty and improves decision-making in gender classification tasks.

Ensemble approaches enhance robustness when gender-related vocal features are ambiguous.

Noise-Robust Training

Training models on datasets that include a wide range of noisy environments prepares them to handle real-world conditions.
It ensures that systems perform consistently well, regardless of environmental noise levels.

Context-Aware Processing

Incorporating contextual cues, such as the content of conversation and environmental factors, helps in understanding voice inputs affected by psychological factors.
Context-awareness enhances system accuracy when speech variations occur due to mood or cognitive state.

Bias Mitigation Techniques

Techniques such as re-sampling underrepresented classes, re-weighting data during training, and adversarial training to debias models.
These methods directly tackle the imbalances in existing datasets and model prediction

Preprocessing Audio Data

Importance of Preprocessing Audio Data in Voice Authentication and Gender Classification

Preprocessing audio data before operations like voice authentication or gender classification is important for several reasons, such as reducing noise, increasing consistency, accuracy, efficiency, etc. let's explain each more.

- By filtering out unwanted background noise, we make the voice clearer and more distinguishable from one another.
- By normalizing all sample voices to a standard and similar level, we aim to compare them more accurately and have a more reliable model.

- Some preprocessing steps reduce the complexity of the data and make the analysis faster, more stable, consistent and efficient.

Usual Steps of Preprocessing Audio Data

Noise Reduction

There exists several methods to do so, Each with a specific goal to achieve. Here are some examples:

- **Filtering**: it is performed on the frequency domain of an audio data. For example using a high-pass filter only passes high-frequency parts and as a result eliminates low-frequency sounds such as a rumble. Low-pass filters perform the same operation but on low-frequency parts. Frequency-domain methods involve taking a FFt at first and an inverse FFT at last.
- **Time-domain methods**: such methods pass only sounds within a specific range to block quieter parts (which are probably noises).
- **ML models**: for example using pre-trained DL models such as CNNs, RNNs and autoencoders to detect and remove noises is very popular these days.

Normalization

It involves adjusting the amplitude of audio signals to a standard level. As mentioned earlier, it helps to train a more reliable and consistent model later. Models that use normalized data, often converge faster and more accurately due to stability of calculations. Here are some popular methods to do so:

- **Peak Normalization**: Adjusts the amplitude of the audio signal so that the highest peak reaches a target level (usually 0 dBFS).
- **Loudness Normalization**: Adjusts the audio signal to achieve a consistent perceived loudness level. It takes into account how humans perceive loudness at different frequencies and standards the audio signal to that measure.

- **RMS Normalization**: Adjusts the root mean square level of the audio signal to a target value. This method provides a balance between the two previously mentioned methods, giving a more consistent overall level.

In python, there exists some powerful tools to normalize audio files with. Such as *Pydub* and *Pynormalize*.

Windowing

It‘s an important part of audio preprocessing that involves segmenting the audio signal into smaller, manageable parts called windows and then process each one individually. Processing smaller windows of data can be more computationally efficient and manageable. It provides us with more options of how to process a signal. But why do we use this method? Audio signals often change over time. Windowing allows for localized analysis of the signal. Another reason is that applying a windowing function can minimize discontinuities at the boundaries of each segment, reducing spectral leakage and improving the accuracy of the frequency analysis.

Here are some windowing functions:

- **Rectangular Window**: The simplest form where each segment is treated equally.
- **Hamming Window** Applies a cosine shape to the segment. Reduces spectral leakage and is commonly used in practice.
- **Gaussian Window**: Performs windowing based on the Gaussian function.

Feature Extraction Techniques

Mel-Frequency Cepstral Coefficients (MFCC)

First let's introduce mel-frequency cepstrum. It represents the short-term power spectrum of a sound, based on a linear cosine transform of a log power spectrum on a nonlinear mel scale of frequency.

MFCC is a way to represent the short-term mower spectrum of a sound which helps ML models to understand and process human voices more effectively.

MFCCs are actually mathematical representations of the vocal produced by us humans as we speak.

Steps to calculate MFC coefficients:

- **Pre-emphasis**: It boosts high-frequency samples using a high-pass filter. By applying below filter on the signal:

  $$y[n] = x[n] - \alpha\, x[n-1]$$

  Where $y[n]$ is the output sample, $x[n]$ and $x[n-1]$ are consecutive samples and α is called pre-emphasis coefficient which typically is in range of $[0.95, 0.97]$ .For low-frequencies, two consecutive samples $x[n]$ and $x[n-1]$ are close so the subtraction result is close to zero. Because high-frequency samples change more rapidly, the subtraction result is large. In conclusion this filter passes high-frequencies and attenuates low-frequency samples.

- **Framing**: It is the step of dividing the audio signal into overlapping frames. A common frame length is in range of $[20, 40]$ milliseconds. The overlap between two consecutive frames is usually 50 percent. Meaning that for frame length of 30, first one is in range $[0, 30]$ and the second frame $[15, 45]$. Analyzing just a frame at a time makes it easier to capture features and the time-varying nature of the signal. The overlapping is to capture between-frames features.
- **Windowing**: When performing a DTFT on a frame of audio, discontinuities at the edges of the frame can cause problems. Windowing reduces these

discontinuities by smoothing the transitions between frames. This is important in overlapping frames, as it ensures that the signal doesn't make sudden jumps

- **Discrete Time Fourier Transform (DTFT)**: it is the act of converting each frame from the time domain to the frequency domain. DTFT is calculated as follows:

$$\hat{x}(w) = \sum_{n=1}^{N} x[n] \exp(-i\frac{2\pi}{N}nw)$$

- **Mel-Filter**: Purpose is to transform the frequency spectrum obtained from the FFT into the mel scale, which approximates how humans perceive sound. Mel scale is calculated as follows:

  $m = 2595 \log(1 + \frac{f}{700})$

- **Logarithm**: after applying mel-filters, what we get is a set of energies, each corresponding to a mel-frequency. Taking a logarithm of these, mimics the human ear's perception of loudness, as our perception of sound is more logarithmic than linear.
- **Discrete Cosine Transform (DCT)**: It is applied to the logarithmic mel-filter bank energies to decorrelate them and pack the most significant information into the fewest coefficients.

The resulting coefficients of DCT are the MFCCs and we're done.

Fast Fourier Transform (FFT)

It uses a divide-and-conquer approach to efficiently compute the DTFT. WHat it does is dividing N samples into to groups of $\frac{N}{2}$ samples and repeat this recursively until the size of samples is small enough to calculate the DTFT fast. After that, it merges the answers. Regular DTFT multiples a $N \times N$ matrix to a $N \times 1$ one and that's done in $O(N^2)$ time complexity. But FFT archives the same goal in $O(N\,log(N))$ time complexity. It is shown using the Master Theorem:

$T(N) = T(\frac{N}{2}) + T(\frac{N}{2}) + O(N) = 2T(\frac{N}{2}) + O(N) = O(N\,log(N))$

Here's a more detailed explanation of what it does:

$$\hat{x}(w) = \sum_{n=0}^{N-1} x[n] \exp(-i\frac{2\pi}{N}nw)$$

Let's break down $\hat{x}(w)$ into its even and odd indexed values.

$$\hat{x}(w) = \sum_{n=0}^{N/2-1} x[2n] \exp(-i\frac{2\pi}{N}(2n)w) + \sum_{n=0}^{N/2-1} x[2n+1] \exp(-i\frac{2\pi}{N}(2n+1)w)$$

$$\hat{x}(w) = \sum_{n=1}^{N/2} x[2n] \exp(-i\frac{2\pi}{N}(2n)w) + \exp(-i\frac{2\pi}{N}w) \sum_{n=1}^{N/2} x[2n-1] \exp(-i\frac{2\pi}{N}(2n)w)$$

$$\hat{x}(w) = \sum_{n=1}^{N/2} x[2n] \exp(-i\frac{2\pi}{N/2}nw) + \exp(-i\frac{2\pi}{N}w) \sum_{n=1}^{N/2} x[2n-1] \exp(-i\frac{2\pi}{N/2}nw)$$

solving them recursively gives us $\hat{x}_e(w)$ and $\hat{x}_o(w)$. This is how to merge them:

$$\hat{x}(w) = \hat{x}_e(w) + \exp(-i\frac{2\pi}{N}w) \times \hat{x}_o(w)$$

The matrix representation of FFT calculations is $\hat{X} = WX$ where X is the input samples vector of size N, $\hat{X}$ is the output vector of size N and W is the $N \times N$ matrix which is known as transformation matrix. If we define w as $\exp(-i\frac{2\pi}{N})$, $W_{n,k} = w^{n \times k}$. Assuming N is a power of two, what FFT does is that it calculates WX like this:

$$\hat{X} = \begin{bmatrix} I_{N/2} & -D_{N/2} \\ I_{N/2} & -D_{N/2} \end{bmatrix} \begin{bmatrix} W_{N/2} & 0 \\ 0 & W_{N/2} \end{bmatrix} [X_e^T \; X_o^T]^T$$

Where:

- $I_{N/2}$ is the identity matrix of size $\frac{N}{2} \times \frac{N}{2}$
- $D_{N/2}$ is diagonal matrix of size $\frac{N}{2} \times \frac{N}{2}$ in which $D_{n,n}$ is $w^{n \times n} = w^{n^2}$.
- $W_{N/2}$ is the transformation matrix of size $\frac{N}{2} \times \frac{N}{2}$
- X_e is the vector of even indexed elements of X
- X_o is the vector of odd indexed elements of X

The $W_{N/2}$ matrixes are can be further broken to $W_{N/4}$ and so on. Multiplying identity and diagonal matrices are a lot less expensive than the original transformation matrix.

In summary, FFT rearranges samples in a way to avoid unnecessary and duplicate calculations.

Log Mel Spectrogram

A Log Mel Spectrogram is a visual representation of the spectrum of frequencies in a sound signal as they vary with time, but with additional processing steps to make it more useful for specific audio analysis tasks, especially in the context of speech and music. Let's break down the components and steps involved in creating a Log Mel Spectrogram:

1. **Short-Time Fourier Transform (STFT)**:
    - The audio signal is divided into short overlapping segments (frames), and a Fourier Transform is applied to each segment to convert the time-domain signal to the frequency-domain. This process creates a spectrogram, which shows how the frequency content of the signal changes over time.
    - The STFT typically uses a window function (like a Hamming or Hann window) to manage the trade-off between frequency resolution and time resolution.

$$X_m[k] = \sum_{0}^{n-1} x_m[n].w[n].e^{-j\frac{2\pi kn}{N}}$$

- $X_m[k]$ is the STFT of the mm-th segment.
- $x_m[n]$ is the mm-th windowed segment of x[n].
- $w[n]$ is the window function (e.g., Hamming window).
- N is the number of points in the FFT.
- k is the frequency bin index.

2. **Mel Filter Bank**:
    - The spectrogram is then passed through a Mel filter bank. The Mel scale is designed to mimic the human ear's perception of sound, which is more sensitive to lower frequencies than higher frequencies.
    - A set of triangular filters is applied to the spectrogram. Each filter captures a range of frequencies and produces a single value. The result is a Mel spectrogram with a lower dimensionality than the original spectrogram.

$$m[f] = 2595\, log_{10}(1 + \frac{f}{700})$$

$$S_m[i] = \sum_{k=0}^{n-1} |x_m[k]|^2.H_i[k]$$

- $S_m[i]$ is the Mel spectrogram.
- $H_i[k]$ is the i-th triangular Mel filter.

3. **Logarithmic Scaling**:
    - The Mel spectrogram values are typically on a linear scale. To make them more suitable for analysis (especially for machine learning applications), a logarithmic

scale is applied. This scaling helps compress the dynamic range of the spectrogram, making quieter sounds more noticeable relative to louder sounds.

- The logarithmic scale also better represents how humans perceive loudness.

$$L_m[i] = log(Sm[i])$$

Spectral Centroid

The spectral centroid is a measure used in digital signal processing to describe the "center of mass" of the spectrum. It indicates where the center of the mass of the spectrum is located and is often perceived as the brightness of a sound. Higher values of the spectral centroid correspond to brighter, more treble-heavy sounds, whereas lower values correspond to darker, more bass-heavy sounds.

How It Works:

1. **Framing and Windowing**: Divide the signal into overlapping frames and apply a window function to each frame to reduce edge effects.
2. **Short-Time Fourier Transform (STFT)**: Compute the STFT for each frame to obtain the spectrogram, representing the magnitude of the frequency components over time.
3. **Magnitude Spectrum**: Calculate the magnitude spectrum from the STFT.
4. **Compute Spectral Centroid**:
    - For each frame, calculate the spectral centroid using the formula:

$$C = \frac{\sum_{k=0}^{n-1} f(k) \cdot X(k)}{\sum_{k=0}^{n-1} X(k)}$$

where f(k) is the frequency of bin k, and X(k) is the magnitude of the bin k in the spectrum.

The spectral centroid provides a simple yet powerful way to capture the perceptual quality of brightness in an audio signal, making it useful for various applications in audio analysis and processing.

Chroma Features

Chroma features, also known as chroma vectors or chromagrams, are powerful tools in music analysis and audio processing. They capture the harmonic content of a sound and represent it in a way that reflects the twelve different pitch classes in Western music (C, C#, D, D#, E, F, F#, G, G#, A, A#, B). These features are particularly useful in tasks like chord recognition, key detection, and music similarity analysis.

A chroma feature vector C for a given frame of an audio signal represents the energy present in each of the twelve pitch classes. The calculation involves several steps:

1. **Short-Time Fourier Transform (STFT)**:
    - Compute the STFT of the audio signal to obtain its frequency-domain representation.
2. **Magnitude Spectrum**:
    - Calculate the magnitude spectrum |Xm[k]|X_m[k]| from the STFT.
3. **Mapping Frequencies to Pitch Classes**:
    - Each frequency bin is mapped to one of the twelve pitch classes. The frequency f in Hertz is converted to a pitch class using the following formula:

$$p(f) = 12 \cdot log_2(\frac{f}{f_{ref}})$$

where f_{ref} is a reference frequency, typically set to 440 Hz (A4).

4. **Chroma Vector Calculation**:
    - For each frequency bin, determine its corresponding pitch class and accumulate its energy into the appropriate bin of the chroma vector C:

$$C[j] = \sum_{k \in B(j)} |Xm[k]|^2$$

Where:

- C[j] is the chroma feature for the j-th pitch class.
- B(j) is the set of frequency bins corresponding to the j-th pitch class.

5 **. Normalization**

Chroma vectors are typically normalized to ensure that they are not influenced by the overall loudness of the signal:

$$C[j] = \frac{C[j]}{\sum_{i=0}^{11} C[i]}$$

Chroma features provide a compact and informative representation of the harmonic content of an audio signal. They are widely used in music information retrieval and other audio analysis tasks due to their ability to capture the essential characteristics of musical harmony.

Spectral Contrast

Spectral contrast is a feature used in audio analysis that measures the difference in amplitude between peaks and valleys in a sound spectrum. This feature is particularly useful for music genre classification, speech/music discrimination, and audio segmentation. It provides insight into the timbre and texture of the sound, complementing other features like spectral centroid and chroma features.

Spectral contrast Δ\Delta for a given frame of an audio signal is calculated by measuring the difference in amplitude between spectral peaks (high energy) and valleys (low energy) across different frequency bands.

1. **Short-Time Fourier Transform (STFT)**:
    - Compute the STFT of the audio signal to obtain its frequency-domain representation.
2. **Magnitude Spectrum**:
    - Calculate the magnitude spectrum from the STFT.
3. **Frequency Bands**:
    - Divide the spectrum into several frequency bands. The division can be done linearly or logarithmically, depending on the application.
4. **Identify Peaks and Valleys**:

- For each frequency band, identify the spectral peaks (highest energy values) and valleys (lowest energy values).

5. **Compute Spectral Contrast**:
   - Calculate the spectral contrast for each band by measuring the difference between the peaks and valleys.

$$\Delta_m[i] = 10 \cdot log10(\frac{P_m[i]}{V_m[i]})$$

Where:

- $\Delta_m[i]$ is the spectral contrast for the i-th band in the m-th frame.
- $P_m[i]$ is the peak value in the i-th band.
- $V_m[i]$ is the valley value in the i-th band.

Spectral contrast provides a valuable measure of the timbral richness and texture of an audio signal. It complements other spectral features and is widely used in various audio analysis and processing tasks.

Zero-Crossing Rate

The Zero-Crossing Rate (ZCR) is a measure of how frequently the signal changes sign (i.e., from positive to negative or vice versa) within a given time frame. It is a simple but effective feature used in various audio processing tasks, such as speech recognition, music genre classification, and audio segmentation.

The Zero-Crossing Rate for a given frame of an audio signal can be calculated as follows:

$$ZCR = \frac{1}{N-1} \sum_{n=1}^{N-1} I\{(x[n] \cdot x[n-1]) < 0\}$$

where:

- ZCR is the Zero-Crossing Rate.
- N is the number of samples in the frame.

- x[n] is the audio signal at sample n.
- I{·} is the indicator function, which returns 1 if the condition inside the braces is true, and 0 otherwise.

1. **Segmenting the Signal**:
    - Divide the audio signal into overlapping or non-overlapping frames. Let $x_m[n]$ represent the m-th frame of the signal.
2. **Zero-Crossing Detection**:
    - For each frame, count the number of times the signal changes sign. This can be done by checking the product of consecutive samples:

Zero-Crossing={1 if x[n]·x[n-1]<0 otherwise 0}

Calculate ZCR:

- Normalize the count of zero-crossings by the number of samples in the frame to obtain the Zero-Crossing Rate.

The Zero-Crossing Rate is a simple yet powerful feature for analyzing the temporal structure of an audio signal. It provides valuable information about the frequency content and can be used in various audio processing tasks.

Linear Predictive Coding (LPC)

Linear Predictive Coding (LPC) is a powerful tool used in speech processing and audio analysis. It provides a mathematical model to represent the spectral envelope of a digital signal of speech in compressed form, using the information of a linear predictive model.

LPC is based on the principle that a speech sample s(n) can be approximated as a linear combination of past speech samples:

$$s(n) = \sum_{k=1}^{p} a_k \cdot s(n-k) + e(n)$$

where:

- s(n) is the current speech sample.

- p is the order of the LPC (the number of past samples used).
- a_k are the LPC coefficients.
- e(n) is the prediction error (residual).

The objective of LPC is to minimize the prediction error over a segment of speech.

Steps to Calculate LPC Coefficients

1. **Windowing**:
    - Divide the speech signal into overlapping frames. Apply a window function (e.g., Hamming window) to each frame to reduce edge effects.
2. **Autocorrelation**:
    - Compute the autocorrelation function of each windowed frame. The autocorrelation R(m) for lag mm is given by:

$$R(m) = \sum_{n=0}^{N-1-m} s(n) \cdot s(n+m)$$

3. **Levinson-Durbin Recursion**:
    - Solve the Yule-Walker equations using the Levinson-Durbin recursion to find the LPC coefficients a_k. This is an efficient algorithm to solve the set of linear equations derived from the autocorrelation method.
4. **Compute Prediction Error**:
    - The prediction error e(n) is the difference between the actual speech sample and the predicted sample using LPC coefficients.
    - Calculate the prediction error

$$e(n) = s(n) - \sum_{k=1}^{p} a_k . s(n-k)$$

Linear Predictive Coding is a fundamental technique in speech processing, providing a compact and efficient representation of the speech signal. It is widely used in various applications, from speech synthesis and coding to speaker recognition and speech analysis.

Perceptual Linear Prediction (PLP)

Perceptual Linear Prediction (PLP) is a speech analysis technique that incorporates auditory perception models to improve the robustness and accuracy of speech features. It builds on the principles of Linear Predictive Coding (LPC) but includes additional steps to mimic the human auditory system's response. PLP is particularly useful in speech recognition and speaker identification.

PLP involves several steps that transform the speech signal into a set of coefficients that represent the perceptually relevant aspects of the signal.

Steps to Calculate PLP Coefficients

1. **Pre-emphasis**:
    - Apply a pre-emphasis filter to the speech signal to enhance high-frequency components. This can be represented as:

$$y[n] = x[n] - \alpha x[n-1]$$

    Where:

    - y[n] is the pre-emphasized signal.
    - x[n] is the original speech signal.
    - α is a pre-emphasis coefficient, typically around 0.95.
2. **Windowing and FFT**:
    - Divide the pre-emphasized signal into overlapping frames and apply a window function (e.g., Hamming window) to each frame. Compute the Short-Time Fourier Transform (STFT) to obtain the frequency-domain representation.

3. **Critical Band Analysis (Bark Scale)**:
    - Convert the frequency scale to the Bark scale to model the critical bands of human hearing. The Bark scale z(f)z(f) is a nonlinear mapping of frequency ff in Hertz:

$$z(f) = 6 \cdot sinh^{-1}(\frac{f}{600})$$

Group the spectrum into critical bands and integrate the energy within each band.

4. **Equal-Loudness Pre-emphasis**:
    - Apply an equal-loudness contour to model the frequency-dependent sensitivity of human hearing. This contour emphasizes frequencies where human hearing is more sensitive.
5. **Intensity-Loudness Conversion**:
    - Convert the integrated band energies to a loudness scale using the cubic root compression:

$$E'(z) = \sqrt[3]{E(z)}$$

where E(z) is the energy in the z-th critical band.

6. **Inverse Discrete Fourier Transform (IDFT)**:
    - Compute the IDFT of the loudness-compressed spectrum to obtain the autocorrelation sequence.
7. **Linear Prediction (LP) Analysis**:
    - Apply the Levinson-Durbin recursion to the autocorrelation sequence to obtain the LP coefficients. These coefficients are the PLP coefficients.

Perceptual Linear Prediction is a sophisticated speech analysis technique that incorporates auditory perception models to enhance the robustness and accuracy of speech features. It is widely used in various applications, including speech recognition, speaker identification, and speech synthesis.

Similarity Learning

Similarity learning is a branch of machine learning focused on training models to recognize the similarity or dissimilarity between data points. It is pivotal for tasks such as recommendation systems, image recognition, and anomaly detection, as it enables machines to uncover patterns, relationships, and structures within data.

At its core, similarity learning determines how alike or different two data points are. For instance, given two photos, the goal might be to identify if they depict the same person. Rather than analyzing every pixel, similarity learning algorithms extract key features (e.g., the shape of the eyes or the curve of the mouth) and compare them.

These algorithms typically work within feature spaces, where data points are represented as vectors. The "distance" between these vectors quantifies their similarity, with smaller distances indicating greater similarity. This approach allows efficient comparisons in large datasets.

Unlike traditional supervised learning, which predicts labels from input data, or unsupervised learning, which uncovers hidden structures, similarity learning operates between these paradigms. It doesn't always rely on explicit labels but requires references or pairs of data to assess similarity or dissimilarity. In essence, it models relationships within the data rather than focusing solely on prediction or clustering.

Voice Similarity Learning

Voice similarity learning is a subfield of similarity learning that focuses on training models to determine how similar or different two voice samples are. This technique is crucial for applications like speaker identification, verification, and voice-based biometric systems. The goal is to assess how closely one voice matches another, typically by comparing key features of speech.

Using Similarity Learning for Recognizing Similarity Between Characteristics of Audio

Similarity learning can be applied to recognize similarities between the characteristics of audio by comparing key features extracted from audio samples. This process is particularly useful in tasks like speaker identification, emotion detection, or audio classification. Below are the steps involved in using similarity learning for recognizing similarities between audio characteristics.

Audio Data Collection and Preprocessing

The first step is to collect and preprocess audio data. This involves ensuring that the audio files are in a consistent format (such as WAV or MP3) and have an appropriate sample rate (e.g., 16

kHz or 44.1 kHz). Preprocessing may include noise reduction techniques to minimize background noise, as well as segmenting longer audio clips into smaller, manageable frames for more detailed analysis.

Feature Extraction

Next, relevant features are extracted from the audio signal, which will be used to compare the characteristics of different audio samples. Commonly extracted features include:

- Mel-Frequency Cepstral Coefficients (MFCCs): These capture the spectral properties of sound, representing the power spectrum of audio and are widely used in speech and audio processing.
- Pitch and Formants: Pitch captures the fundamental frequency of the sound, and formants help distinguish different speech sounds, which can be particularly useful for voice comparison.
- Prosody: Features related to rhythm, stress, and intonation, which can provide insights into the emotional tone or speaking style of the speaker.

Feature Representation and Embedding

Once features are extracted, they are transformed into feature vectors, which represent the audio in a high-dimensional space. These vectors are mapped into a lower-dimensional embedding space, where the similarity between two audio samples can be assessed based on their relative positions in this space.

Similarity Learning Model

A similarity learning model is then trained to learn how to measure the similarity between audio samples. Techniques like Siamese Networks or Triplet Networks are commonly used:

- Siamese Networks: These networks consist of two identical neural networks that process two input audio samples and learn to produce embeddings that reflect their similarity. The model is trained to minimize the distance between similar samples and maximize it for dissimilar samples.
- Triplet Networks: Similar to Siamese Networks but with three input samples (anchor, positive, and negative) used to learn the relative similarity between the samples. The

model aims to pull the positive sample closer to the anchor and push the negative sample further apart.

- Contrastive or Triplet Loss: These loss functions guide the model by penalizing incorrect similarity measurements. The goal is to correctly distinguish similar and dissimilar pairs of audio samples.

Similarity Measurement

After training the model, the similarity between audio samples is computed by measuring the distance between their embeddings. Common distance metrics include:

- Euclidean Distance: Measures the straight-line distance between two points in the embedding space.
- Cosine Similarity: Measures the cosine of the angle between two vectors, helping to quantify their directional similarity.

A smaller distance or higher similarity score indicates that the audio samples share more similar characteristics.

Decision Threshold

To determine whether two audio samples are similar, a decision threshold is applied to the similarity score. If the distance between two embeddings is below a predefined threshold, the samples are considered similar. This threshold is typically chosen based on a balance between false positives and false negatives. Metrics such as precision, recall, and F1-score are used to evaluate the performance of the similarity learning model.

Loss Functions in Similarity Learning

The core of similarity learning lies in designing effective loss functions that ensure the model learns meaningful representations. Below are some of the most widely used loss functions in similarity learning, along with detailed explanations.

Contrastive Loss

This function is used to learn embeddings where similar pairs are close together, and dissimilar pairs are far apart.

Given a pair of samples (x_i, x_j) and a label y_{ij}, where $y_{ij} = 1$if the samples are similar and $y_{ij} = 0$ otherwise, the loss is defined as:

$$L = \frac{1}{2N}\sum_{ij}(y_{ij}d(x_i, x_j)^2 + (1 - y_{ij})max(0, m - d(x_i, x_j)^2)$$

- $d(x_i, x_j)$: The distance between embeddings of x_i and x_j, often Euclidean distance.
- m: Margin that enforces a minimum distance between dissimilar pairs.

This loss function is mostly used in face verification and person re-identification. Also it is used in speaker verification tasks, where the goal is to determine if two audio samples are from the same speaker.

Triplet Loss

This loss function ensures that the embedding of an anchor sample is closer to a positive sample (of the same class) than to a negative sample (of a different class).
Given an anchor x_a, a positive x_p, and a negative x_n, the loss is:

$$L = \frac{1}{N}\sum_{i} max(0, d(x_a^i, x_p^i) - d(x_a^i, x_n^i) + m)$$

- $d(x_a^i, x_p^i)$: Distance between anchor and positive.
- $d(x_a^i, x_n^i)$: Distance between anchor and negative.
- m: Margin ensuring separation.

This loss function is mostly used in facial recognition and image retrieval. Also it is commonly used in speaker identification and voiceprint recognition.

Hinge Loss

This loss function is used in metric learning to enforce a margin between similar and dissimilar pairs.

$$L = \frac{1}{N}\sum_{i} max(0, m - y_{ij}(d(x_i, x_j) - m))$$

This loss function is mostly used in classification with a margin. It can be used in phoneme classification or emotion detection in audio.

Center Loss

This loss function ensures that embeddings of the same class are close to their class centers.

$$L = \frac{1}{2N}\sum_{i} ||x_i - c_{y_i}||^2$$

- c_{y_i}: Center of the class y_i.

This loss function is mostly used in face recognition. It is often paired with softmax loss for speaker embedding learning.

Preprocessing and Extracting Feature

Loading the dataset

First step is to load the dataset. This code mounts Google Drive to access files in Colab and then loads all `.mp3` files from the corresponding folder. It then loads each audio file, checks its sample rate, and resamples it to 44.1 kHz if needed. Finally, it stores the resampled audio data. As we have different dataset for each part, we repeat the same segment of code 4 times but in the report, we have shown each segment once as their functionality is the same.

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
audio_folder_identity1       =       '/content/drive/My       Drive/ML/Final
Project/Identity'
audio_folder_identity2 = '/content/drive/My Drive/ML/Final Project/Identity
2'
audio_folder_identity3       =       '/content/drive/My       Drive/ML/Final
Project/Identity3'
audio_folder_gender        =        '/content/drive/My        Drive/ML/Final
Project/Gender_data'
audio_files_identity1    =    glob.glob(os.path.join(audio_folder_identity1,
"*.mp3"))
audio_files_identity2    =    glob.glob(os.path.join(audio_folder_identity2,
"*.mp3"))
audio_files_identity3    =    glob.glob(os.path.join(audio_folder_identity3,
"*.mp3"))
audio_files_gender = glob.glob(os.path.join(audio_folder_gender, "*.mp3"))

target_sr = 44100
num = 0
audio_data_identity1 = {}
audio_data_identity2 = {}
audio_data_identity3 = {}
audio_data_gender = {}
num = 0
for file in audio_files_identity1:
      sr, y = trim_audio(file)
      if sr != target_sr:
      y = librosa.resample(y, orig_sr=sr, target_sr=target_sr)
      audio_data_identity1[file] = (y, target_sr)
```

```
        num += 1
print(num, " of files are read")
```

This function is to cut at most half a minute of each file in order to keep the computations efficient. spectral centroid is a feature representing the "brightness" of the sound. By identifying the time where the centroid is highest, we can cut from 15 seconds before the centroid until 15 seconds after it.

```
def trim_audio(file_path, duration=15):
        y, sr = librosa.load(file_path, sr=None)
        spectral_centroid = librosa.feature.spectral_centroid(y=y, sr=sr)
        times  = librosa.frames_to_time(np.arange(spectral_centroid.shape[1]),
sr=sr)
        centroid_index = np.argmax(spectral_centroid)
        centroid_time = times[centroid_index]
        start_time = max(0, centroid_time - duration)
        end_time = min(len(y) / sr, centroid_time + duration)
        start_sample = int(start_time * sr)
        end_sample = int(end_time * sr)
        trimmed_y = y[start_sample:end_sample]
        return sr, trimmed_y
```

Datasets

As we want to test models with different datasets for identifying identity and some files were not named correctly, we splitted the main dataset into 4 sub datasets:

1. Identity 1: 42 audio files of 6 different people (3 females, 3 males)
2. Identity 2: 42 audio files of 6 different people (3 females, 3 males)
3. Identity 3: 42 audio files of 6 different people (3 females, 3 males)
4. Gender: 242 audio files (121 females, 121 males)

Detect Noises

To identify noisy signals, we used three thresholds, each for a feature.

Spectral Centroid

As mentioned earlier, it represents the "brightness" of the sound. Higher values often indicate more high-frequency content, which can be a possible noise.

Spectral Rolloff

Indicates the frequency below which a certain percentage of the total spectral energy lies. Higher values can suggest a noisier, more chaotic signal which probably contains noises.

Zero-Crossing Rate

Measures how often the signal changes sign by counting the number of times it crosses zero. A high rate often is a sign of noise since noises tend to go up and down over and over again.
First we need to extract some information from the audio signal such as spectral centroid, spectral rolloff, zero-crossing rate, and MFCCs (Mel-frequency cepstral coefficients). They've been explained in the first phase of the project. At the end we take the mean of each feature. It is a good approximation of the vector and also not all the audio files are the same length. By using averages we map all the audios into a similar feature space.

```
def extract_features(y, sr):
      spectral_centroid       =       librosa.feature.spectral_centroid(y=y,
sr=sr).mean()
      spectral_rolloff        =        librosa.feature.spectral_rolloff(y=y,
sr=sr).mean()
      zero_crossing_rate = librosa.feature.zero_crossing_rate(y).mean()
      mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13).mean(axis=1)
      return         np.hstack([spectral_centroid,         spectral_rolloff,
zero_crossing_rate, mfccs])
```

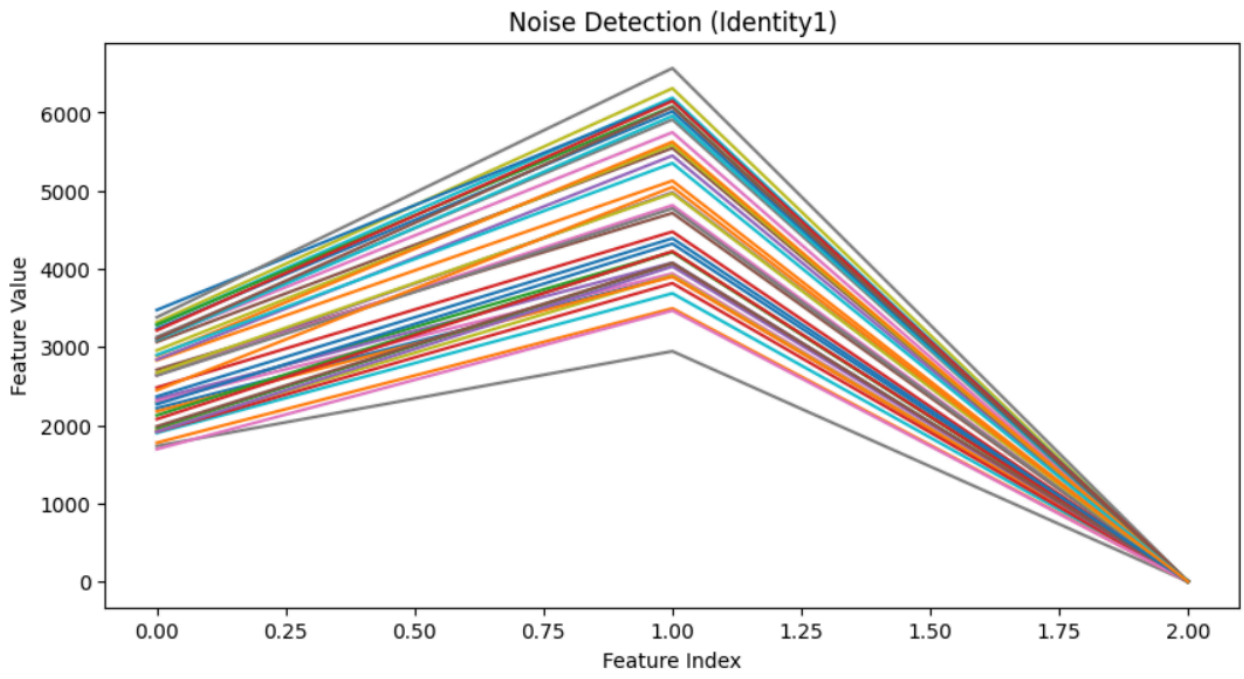
For recognizing noise we need some threshold.

```
NOISE_THRESHOLD_CENTROID = 3000
NOISE_THRESHOLD_ROLLOFF = 5000
NOISE_THRESHOLD_ZCR = 0.05
```

For each dataset we have to find those audio files that are noisy. This segment of code recognizes noisy files and prints the address of files. Also it draws the amount of features based on their indexes.

```
features_indentity1={}
audio_results_identity1={}
for file,value in audio_data_identity1.items():
      y,sr=value
      feature = extract_features(y, sr)
      features_indentity1[file] = feature
      spectral_centroid, spectral_rolloff, zero_crossing_rate = feature[:3]
      is_noisy = (spectral_centroid > NOISE_THRESHOLD_CENTROID or
                  spectral_rolloff > NOISE_THRESHOLD_ROLLOFF or
                  zero_crossing_rate > NOISE_THRESHOLD_ZCR)

      audio_results_identity1[file] = "Noisy" if is_noisy else "Clean"

for file, status in audio_results_identity1.items():
      print(f"{os.path.basename(file)} → {status}")

plt.figure(figsize=(10, 5))

for file, feature in features_indentity1.items():
      plt.plot(feature[:3], label=os.path.basename(file))

plt.xlabel("Feature Index")
plt.ylabel("Feature Value")
plt.title("Noise Detection (Identity1)")
plt.show()
```

This is one of the outputs:



Reduce Noises

After detecting noises, we need to reduce them as much as possible. It's not always possible to entirely remove noises. Two methods are mentioned in the project description. We'll describe both of them and choose one to use.

Spectral subtraction

First it takes a Short-Time Fourier Transform to convert the audio signal into the frequency domain. The transform separates magnitude and phase components. Then it estimates the noise spectrum by averaging the magnitude of the first 10 frames (usually assuming they're noise-dominant). This is done across the columns of the magnitude matrix. The magnitude spectrum is adjusted by subtracting a scaled version of the noise estimate (noise_factor * noise_estimate). The result is clamped to zero to avoid negative values, as magnitude can't be negative. The denoised magnitude is multiplied by the original phase, and the inverse STFT is applied to transform it back to the time domain.

```
def spectral_subtraction(y, sr, noise_factor=0.1):
      D = librosa.stft(y)
      magnitude, phase = np.abs(D), np.angle(D)
```

```
    noise_estimate = np.mean(magnitude[:, :10], axis=1, keepdims=True)
    magnitude = np.maximum(magnitude - noise_factor * noise_estimate, 0)
    D_denoised = magnitude * np.exp(1j * phase)
    y_denoised = librosa.istft(D_denoised)
    return y_denoised
```

Pros

- Effective in reducing broadband noise.
- Works well for stationary noise.

Cons

- Can introduce artifacts (musical tones, distortion).
- Not ideal for non-stationary or complex noise.

Bandpass Filter

This method reduces noise by keeping only the frequencies within the specified low cut (300 Hz) and high cut (3400 Hz) range, which is typically the range of human speech. Frequencies outside this range, such as low-frequency rumble or high-frequency noise, are filtered out.

```
def butter_bandpass(lowcut, highcut, sr, order=4):
    nyquist = 0.5 * sr
    low = lowcut / nyquist
    high = highcut / nyquist
    b, a = signal.butter(order, [low, high], btype='band')
    return b, a

def apply_bandpass_filter(y, sr, lowcut=300, highcut=3400):
    b, a = butter_bandpass(lowcut, highcut, sr)
    y_filtered = signal.filtfilt(b, a, y)
    return y_filtered
```

Pros

- Focuses on a specific frequency range (usually voice range), preserving speech clarity.
- Simple and computationally efficient.

Cons

- Can distort signals outside the targeted frequency range if misapplied.
- Less effective for noise across a wide range of frequencies.

We tried both methods separately and simultaneously and the result of using the bandpass filter was better than the others.

Why does bandpass work better for our dataset?

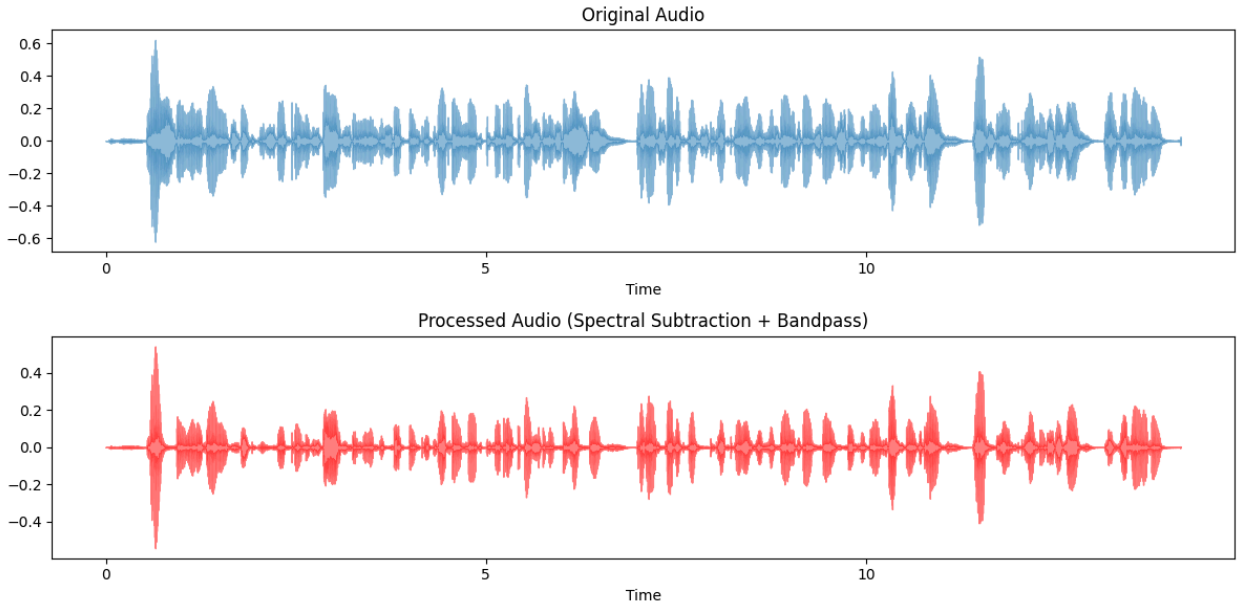
If the noise in the recordings lies outside the typical voice frequency range (around 85 Hz to 255 Hz for males and 165 Hz to 255 Hz for females), bandpass filtering would effectively remove noise while preserving the speech signal. Spectral subtraction might struggle in these cases, especially if the noise is non-stationary or varies in frequency.

```
num = 0
for file,value in audio_results_identity1.items():
      if value == "Noisy":
      num += 1
      y, sr = audio_data_identity1[file]
      y_denoised=spectral_subtraction(y,sr)
      y_filtered = apply_bandpass_filter(y, sr)
      audio_data_identity1[file] = (y_filtered , sr)
      features_indentity1[file] = extract_features(y_filtered, sr)
      plt.figure(figsize=(12, 6))
      plt.subplot(2, 1, 1)
      librosa.display.waveshow(y, sr=sr, alpha=0.5)
      plt.title("Original Audio")
      plt.subplot(2, 1, 2)
      librosa.display.waveshow(y_filtered, sr=sr, alpha=0.5, color='r')
      plt.title("Processed Audio (Spectral Subtraction + Bandpass)")
      plt.tight_layout()
      plt.show()
print("number of noisy data: ", num)
```

With this segment of code we apply noise handling techniques that we mentioned and draw the plot related to the voice before and after of noise handling. Also this segment prints the number of noisy files related to each dataset. Here is the result:

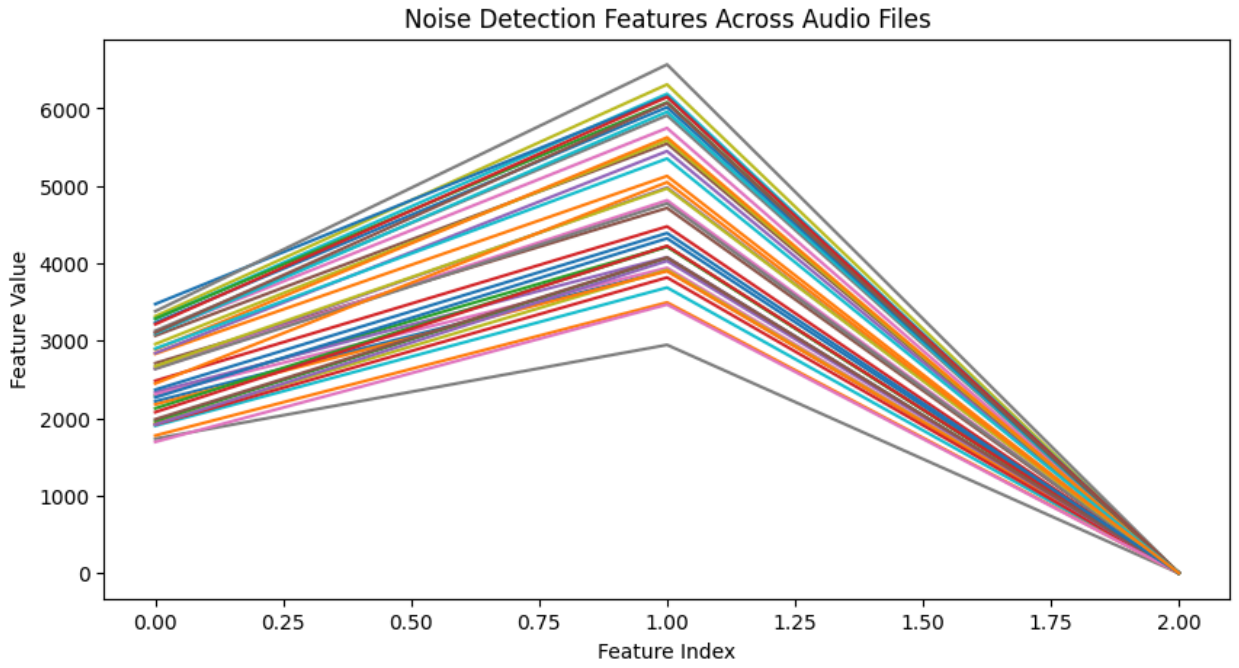
1. Identity 1: 30 noisy files
2. Identity 2: 26 noisy files
3. Identity 3: 32 noisy files
4. Gender: 169 noisy files

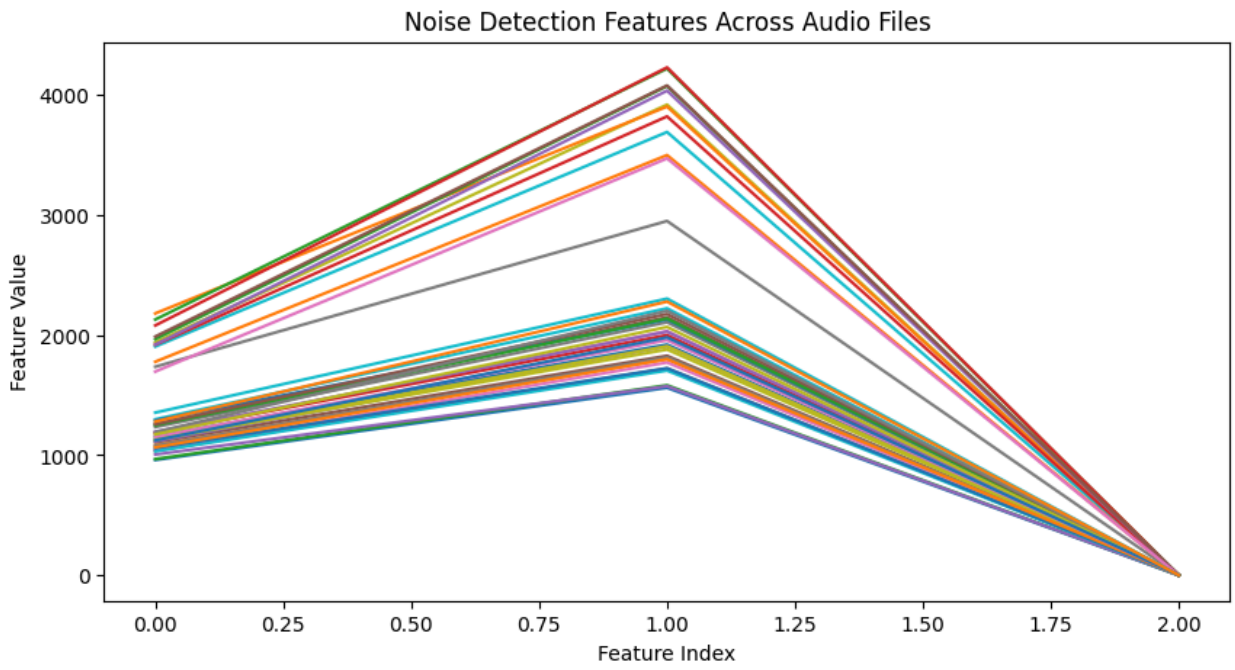
Here's a plot of a noisy audio signal before and after noise reduction. As we can see, the original signal has irregular risings and fallings, indicating the presence of noise, which can be seen as high-amplitude peaks and fast changes in the signal. On the second plot, The noise has been significantly reduced. The waveform appears smoother with fewer high-amplitude, rapid risings and fallings compared to the original signal.



After this, most of the previously noisy signals are clear now based on the three measures explained earlier. We can delete those that are still noisy.

Here's a plot of spectral centroid, spectral rolloff and zero crossing rate for the audio signals before and after reducing noises.





After noise handling we check the dataset again to see if there is any file that remains noisy even after noise handling or not. This segment of code analyses the result of noise handled data and counts the number of noisy files for each dataset.

```
num = 0
for file,value in audio_data_identity1.items():
      y,sr=value
      feature = extract_features(y, sr)
      spectral_centroid, spectral_rolloff, zero_crossing_rate = feature[:3]
      is_noisy = (spectral_centroid > NOISE_THRESHOLD_CENTROID or
                  spectral_rolloff > NOISE_THRESHOLD_ROLLOFF or
                  zero_crossing_rate > NOISE_THRESHOLD_ZCR)

      audio_results_identity1[file] = "Noisy" if is_noisy else "Clean"

for file, status in audio_results_identity1.items():
      if audio_results_identity1[file] == "Noisy":
      num += 1
print(num, " voices remained noisy")
```

The results are as follows:

1. Identity 1: 0 noisy files
2. Identity 2: 5 noisy files
3. Identity 3: 1 noisy files

4. Gender : 20 noisy files

As we applied noise handling and they are still noisy we delete them from datasets with this segment of code:

```
files = []
for file, value in audio_results_identity1.items():
  if audio_results_identity1[file] == "Noisy":
      files.append(file)
      print(file)
print(len(audio_data_identity1))
for file in files:
      audio_results_identity1.pop(file)
      audio_data_identity1.pop(file)
      features_indentity1.pop(file)
print(len(audio_data_identity1))
```

At first we extract the keys that should be deleted from all dictionaries and we will delete them and print the result of deleting which is the length of datasets:

1. Identity 1: 42 -> 42
2. Identity 2: 42 -> 37
3. Identity 3: 42 -> 41
4. Gender : 242 -> 222

Normalizing Audio Signals

Why do we normalize audio signals?

It ensures that the audio signal has a consistent amplitude, preventing clipping or distortion during processing. It standardizes the volume levels so that the signal can be compared or processed without drastic fluctuations.

How to normalize audio signals?

`librosa.util.normalize(y)` normalizes an audio signal by scaling its amplitude within a given range, usually between -1 and 1.

Mathematical explanation:

$$y_{norm} = \frac{y}{\max(|y|)}$$

```
for file, value in audio_data_identity1.items():
  y, sr = audio_data_identity1[file]
  y_norm = librosa.util.normalize(y)
  audio_data_identity1[file] = (y_norm,sr)
```

We normalize all data in the related dictionary of each dataset and then save in the corresponding key of this dictionary as we don't need unnormalized data anymore.

Feature Extraction

Two types of features are extracted. Some are in the frequency domain and some in the time domain.

Frequency-Domain Features

Log Mel Spectrogram

Represents the power of frequencies across time using Mel-scaled filters, converted to decibels.

MFCC (Mel-frequency cepstral coefficients)

Captures the spectral characteristics of the signal, useful for speech and sound recognition.

Spectral Centroid

Indicates the "center of mass" of the spectrum, often related to brightness.

Spectral Bandwidth

Measures the width of the spectrum, indicating timbral texture or clarity.

Spectral Contrast

Measures the difference in amplitude between peaks and valleys in the spectrum.

Time-Domain Features

Zero Crossing Rate

Counts how many times the signal crosses zero, indicating noisiness or percussiveness.

Energy (RMS)

Measures the signal's energy or loudness by calculating its root mean square.

These features together explain the signal such that we'd be able to perform classification and clustering tasks.

```
def extract_audio_features(file_path, y, sr, n_mfcc=13, n_mels=128):
      log_mel_spec                                                    =
librosa.power_to_db(librosa.feature.melspectrogram(y=y,               sr=sr,
n_mels=n_mels))
      mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=n_mfcc)
      spectral_centroid = librosa.feature.spectral_centroid(y=y, sr=sr)
      bandwidth = librosa.feature.spectral_bandwidth(y=y, sr=sr)
      spectral_contrast = librosa.feature.spectral_contrast(y=y, sr=sr)
      zero_crossing_rate = librosa.feature.zero_crossing_rate(y=y)
      energy = librosa.feature.rms(y=y)
      gender = detect_gender(file_path)
      id = split_segments(file_path)[2]
      return {
      "Log_Mel_Spectrogram": log_mel_spec,
      "MFCC": mfcc,
      "Spectral_Centroid": spectral_centroid,
      "Bandwidth": bandwidth,
      "Spectral_Contrast": spectral_contrast,
      "Zero_Crossing_Rate": zero_crossing_rate,
      "Energy": energy,
      "Gender": gender,
      "STID": id
      }
```

This function extracts the features from audio and returns a dictionary of information that we need for each voice. In this segment there are some self defined functions.

```
def detect_gender(text):
      text = text.lower()
      if "female" in text:
      return 1
      elif "male" in text:
      return 0
      else:
      return None
```

This function converts the gender of each voice to a binary number which is 1 or 0. This function actually checks the file path of the voice and finds the word “female” or “male”.

```
def split_segments(s):
      return re.split(r'[_\.]', s)
```

This function splits the parts of the file path for extracting the student number that we need for identifying the identity of voice.

```
audio_details_identity1 = []
for file, (y, sr) in audio_data_identity1.items():
      details = extract_audio_features(file, y, sr)
      audio_details_identity1.append(details)
```

In this loop we extract the features and save in a list for using in the future steps.

As we need data frames as the inputs of models, we convert the lists that we created to data frames with this segment of code:

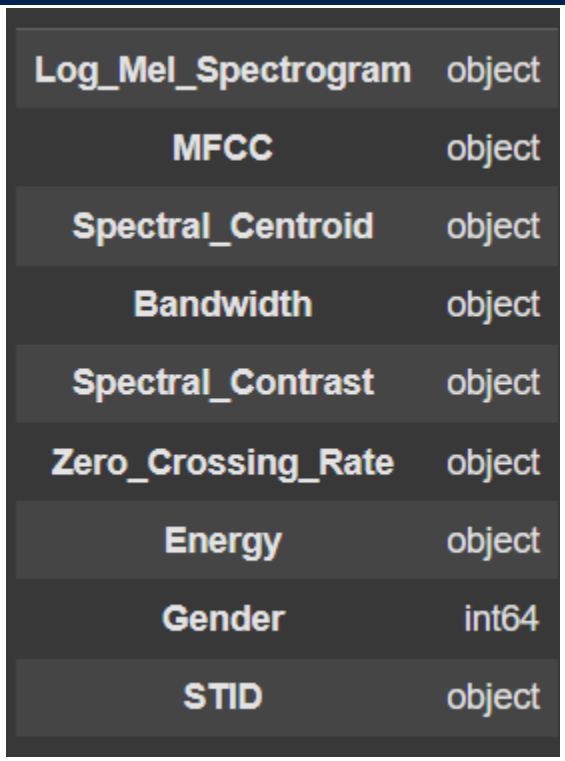
```
df_identity1 = pd.DataFrame(audio_details_identity1)
df_identity1.head()
```

This is the head of Identity 1 dataset:

	Log_Mel_Spectrogram	MFCC	Spectral_Centroid	Bandwidth	Spectral_Contrast	Zero_Crossing_Rate	Energy	Gender	STID
0	[[-22.57332182809629, -28.295782386398628, -45...	[[-286.8047790134551, -303.5843110329075, -329...	[[1146.1331635081046, 954.7189611576457, 746.6...	[[2267.070738642417, 1677.6365556799803, 519.2...	[[3.5186802079973027, 3.8887808955787246, 20.7...	[[0.01220703125, 0.0185546875, 0.02490234375, ...	[[0.08719148, 0.104803905, 0.12385833, 0.11492...	1	810101551
1	[[-24.235916311380606, -30.202685034768884, -4...	[[-419.2930621469898, -401.1195911649325, -298...	[[1971.2330992884492, 1780.7239541012564, 2119...	[[3799.021858973857, 2657.608123851938, 1148.8...	[[2.2567123369106605, 2.0450193596796904, 19.9...	[[0.01025390625, 0.02978515625, 0.037109375, 0...	[[0.0038286496, 0.02268922, 0.03702562, 0.0388...	1	810101551
2	[[-19.952038613150066, -26.062534918949815, -4...	[[-393.8493250612197, -393.74782433396797, -41...	[[1553.8345083144245, 1278.6710544786797, 913....	[[3266.590137977108, 2576.8609722856845, 834.5...	[[1.7395067051531865, 1.9255691505765506, 19.9...	[[0.009765625, 0.01611328125, 0.0224609375, 0....	[[0.009393975, 0.009794941, 0.0104258545, 0.00...	1	810101551
3	[[-11.009839914010593, -17.091016422962657, -4...	[[-294.0582574271171, -310.68520493885626, -33...	[[1437.832122501708, 1155.1326738740245, 951.3...	[[2536.7698767508955, 1840.9050695014637, 723....	[[1.2003316641704305, 0.9625453915492814, 29.6...	[[0.015625, 0.02294921875, 0.03173828125, 0.03...	[[0.04648404, 0.060950246, 0.0706342, 0.073679...	1	810101551
4	[[-35.39879457892274, -40.06223991438229, -45....	[[-349.35254820248247, -351.10903672295063, -3...	[[1391.4211047375684, 1056.8978881933415, 803....	[[2827.6690741455495, 2043.1806983477766, 550....	[[5.036595122429661, 3.054562637726441, 20.890...	[[0.01513671875, 0.02294921875, 0.0341796875, ...	[[0.027652983, 0.0280223, 0.028096933, 0.01009...	1	810101551

Also we can see the type of each feature in this dataset wit this code:

```
df_identity1.dtypes
```



Log_Mel_Spectrogram	object
MFCC	object
Spectral_Centroid	object
Bandwidth	object
Spectral_Contrast	object
Zero_Crossing_Rate	object
Energy	object
Gender	int64
STID	object

Aggregating Features

We took the average of each feature to represent it. That's because the dataset was 3-dimensional and we wanted it to be 2-dimensional for use in further steps.

For each of the columns, it aggregates that feature if it contains lists or numpy n-dimensional arrays.

```
for col in df_identity1.columns:
      if df_identity1[col].dtype == 'object':
      df_identity1[col] = df_identity1[col].apply(lambda x: np.mean(x) if
isinstance(x, (list, np.ndarray)) else x)
```

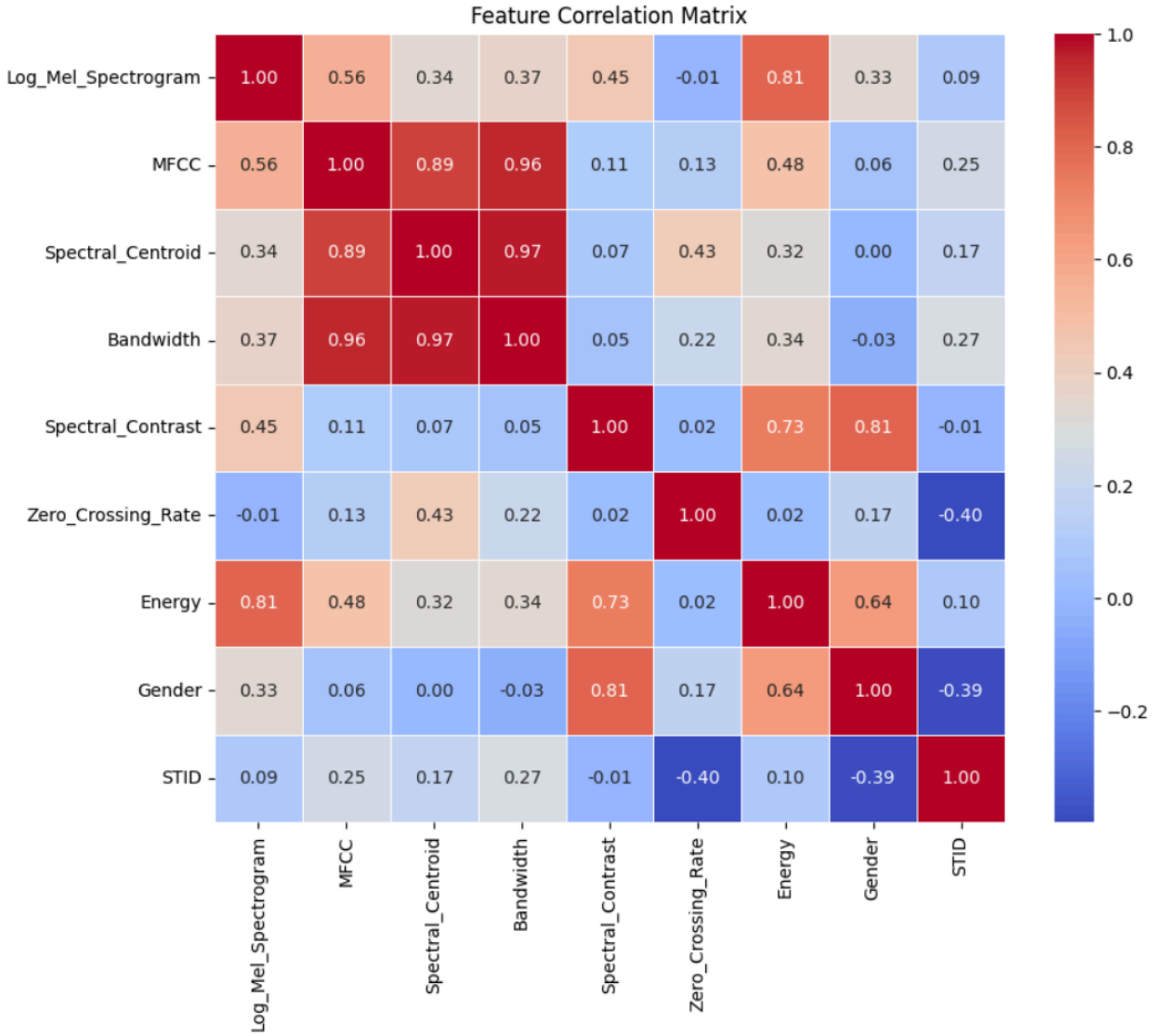
Correlation Matrix

Now that the features are ready, let's see how they correlate to the target variable and one another.

As shown in the heatmap, there are some strongly correlated features such as MFCC, spectral centroid, bandwidth, etc. Also, some of them are nearly linearly explainable by one of the others. For example bandwidth and spectral centroid or bandwidth and MFCC. It leads to redundant information. We'll drop bandwidth to avoid it and also reduce the feature space.

```
correlation_matrix = df_identity1.corr()
correlation_with_gender = correlation_matrix["Gender"].drop("Gender")
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix,  annot=True,  cmap="coolwarm",  fmt=".2f",
linewidths=0.5)
plt.title("Feature Correlation Matrix")
plt.show()
```

The output for Identity 1 is as follows:



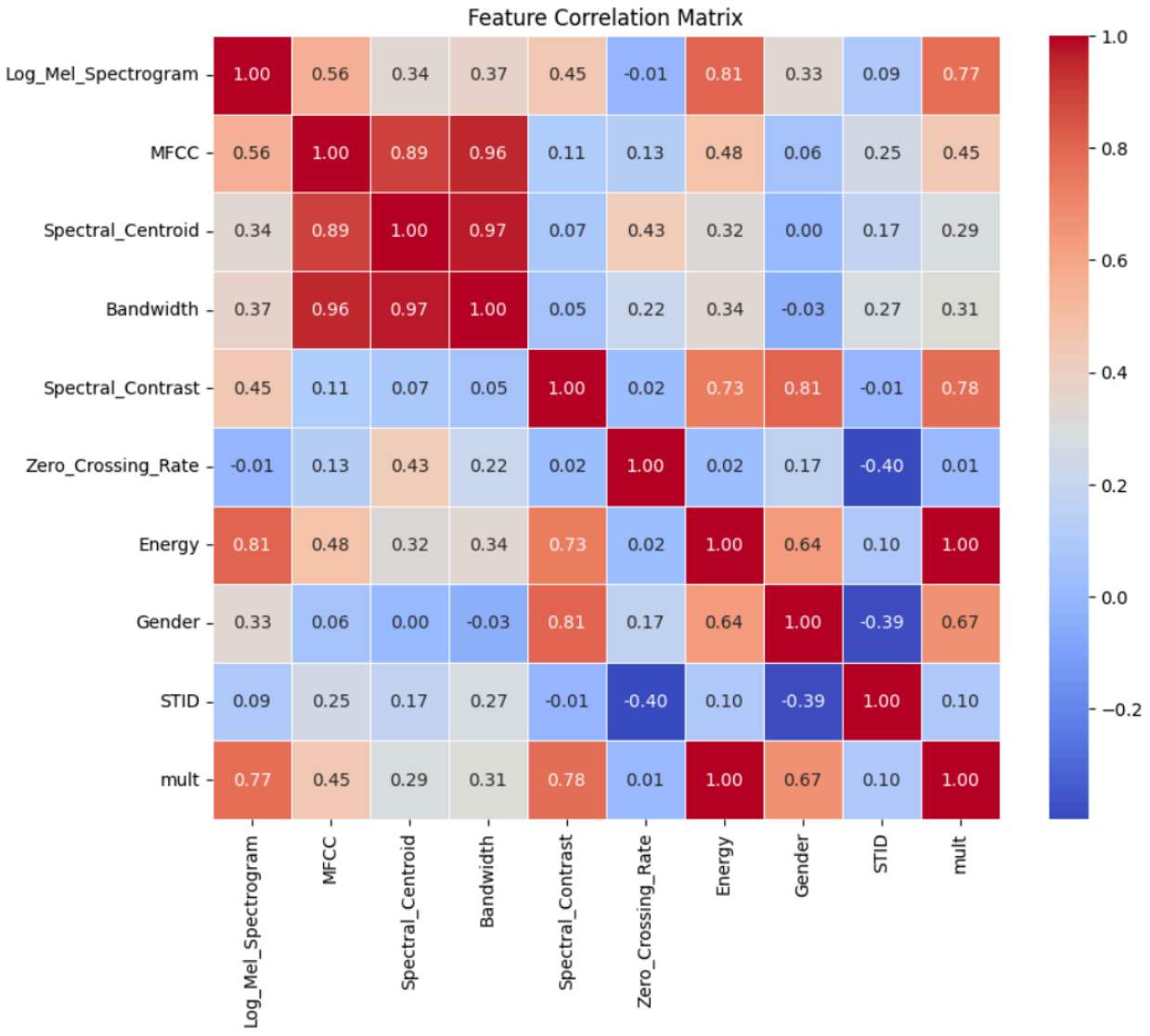
Feature Engineering

Combining time and frequency features captures more complex patterns in the data that might not be visible when analyzing each domain separately. Multiplying Spectral_Contrast (frequency feature) and Energy (time feature) gives us a new feature that captures both the intensity and texture of the signal. This can help better represent how the sound evolves and its loudness, which is useful for tasks like speech recognition or sound classification.

For each dataset, we should multiply different features based on the correlation matrix.

```
df_identity1["mult"]       =       df_identity1["Spectral_Contrast"]       *
df_identity1["Energy"]
correlation_matrix = df_identity1.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix,   annot=True,   cmap="coolwarm",   fmt=".2f",
linewidths=0.5)
plt.title("Feature Correlation Matrix")
plt.show()
```

The output for Identity 1 is as follows:



The new feature which is called “mult” is added to the correlation matrix.

Encoding

As models work with numbers, we have to convert categorical data to numbers. In this dataset we want to identify the identity of people based on the features we have extracted from voices. Student numbers are such a big number that if we use them it will cause the model to have large weights. So we encode each student number to a specific small number starting from 0 with this segment of code.

```
df_identity1['id'], _ = pd.factorize(df_identity1['STID'])
df_identity1 = df_identity1.drop(columns=['STID'])
df_identity1
```

This code encodes the STID and creates a new column called id and deletes the STID column as we don't need that anymore. As an example the converted Identity 1 dataset is as follows:

	Log_Mel_Spectrogram	MFCC	Spectral_Centroid	Bandwidth	Spectral_Contrast	Zero_Crossing_Rate	Energy	Gender	mult	id
0	-31.329070	-34.433218	959.431630	747.019561	23.162315	0.032430	0.089498	1	2.072978	0
1	-29.601908	-32.197572	1083.270853	809.046399	23.818447	0.037752	0.093794	1	2.234034	0
2	-28.612330	-31.966869	967.154661	759.277793	24.256841	0.031868	0.122557	1	2.972839	0
3	-30.236470	-33.344300	1036.169844	772.820384	23.655880	0.035957	0.091743	1	2.170267	0
4	-31.078236	-33.471657	1004.957474	732.702344	22.905368	0.036042	0.081256	1	1.861201	0
5	-29.116251	-32.240628	1107.063508	843.440461	22.269527	0.039705	0.084402	1	1.879582	0
6	-29.727754	-32.152841	1043.838227	801.898679	24.182528	0.035681	0.102258	1	2.472861	0
7	-30.555500	-16.939999	1735.266790	2264.690042	23.685460	0.043969	0.115550	1	2.736858	1
8	-30.818607	-19.233486	1938.130426	2643.147040	24.177649	0.039152	0.101492	1	2.453842	1
9	-33.286488	-21.394291	1902.863728	2612.412137	23.755614	0.042410	0.080150	1	1.904011	1
10	-35.201746	-38.521562	1155.269591	743.665628	23.278599	0.042054	0.072114	1	1.678711	1
11	-28.727554	-9.060782	2178.925297	2872.393098	23.504716	0.049179	0.087402	1	2.054356	1
12	-29.138613	-15.657375	2128.036306	2787.475175	23.550043	0.044069	0.131722	1	3.102064	1
13	-34.484745	-21.350460	2078.819437	2615.608015	23.347911	0.046503	0.068384	1	1.596620	1
14	-33.788895	-36.869430	1114.699895	778.190114	22.438953	0.038940	0.052201	1	1.171340	2
15	-33.010336	-35.740664	1259.064315	870.010768	22.968982	0.045031	0.059839	1	1.374436	2

Visualizing Audio Signals

Spectrogram Plot

We used a spectrogram plot. It shows how the frequency content of a signal changes over time.

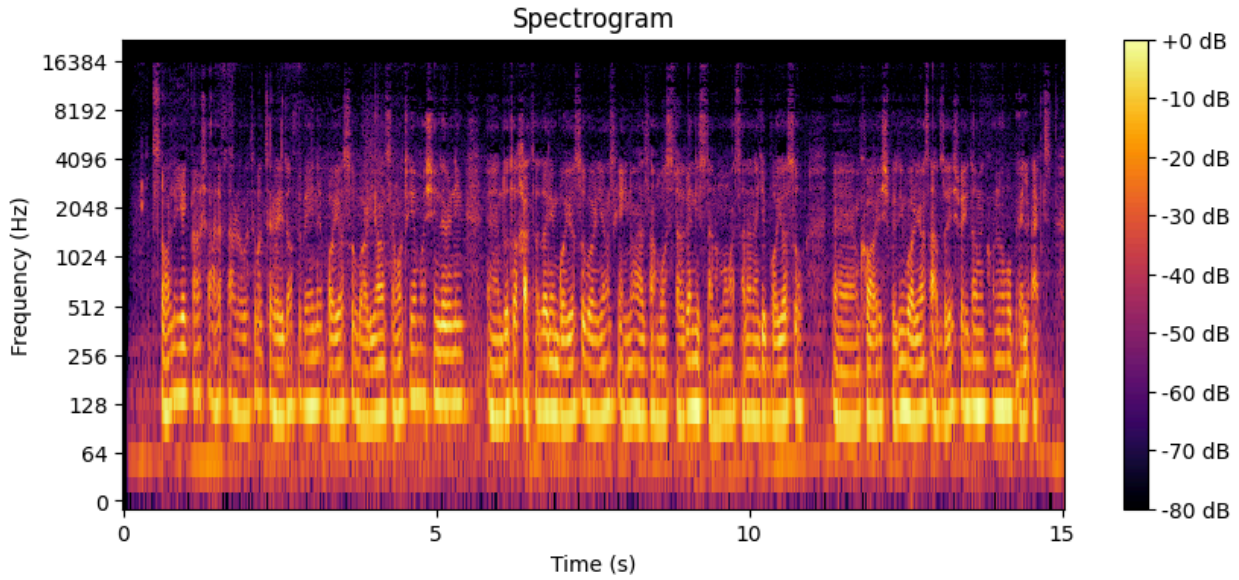
This function will plot the voice for us:

```
def plot_spectrogram(y, sr):
      D = np.abs(librosa.stft(y))
      D_db = librosa.amplitude_to_db(D, ref=np.max)
      plt.figure(figsize=(10, 4))
      librosa.display.specshow(D_db,  sr=sr,  x_axis='time',  y_axis='log',
cmap='inferno')
      plt.colorbar(format="%+2.0f dB")
      plt.title(f"Spectrogram")
      plt.xlabel("Time (s)")
      plt.ylabel("Frequency (Hz)")
      plt.show()
```

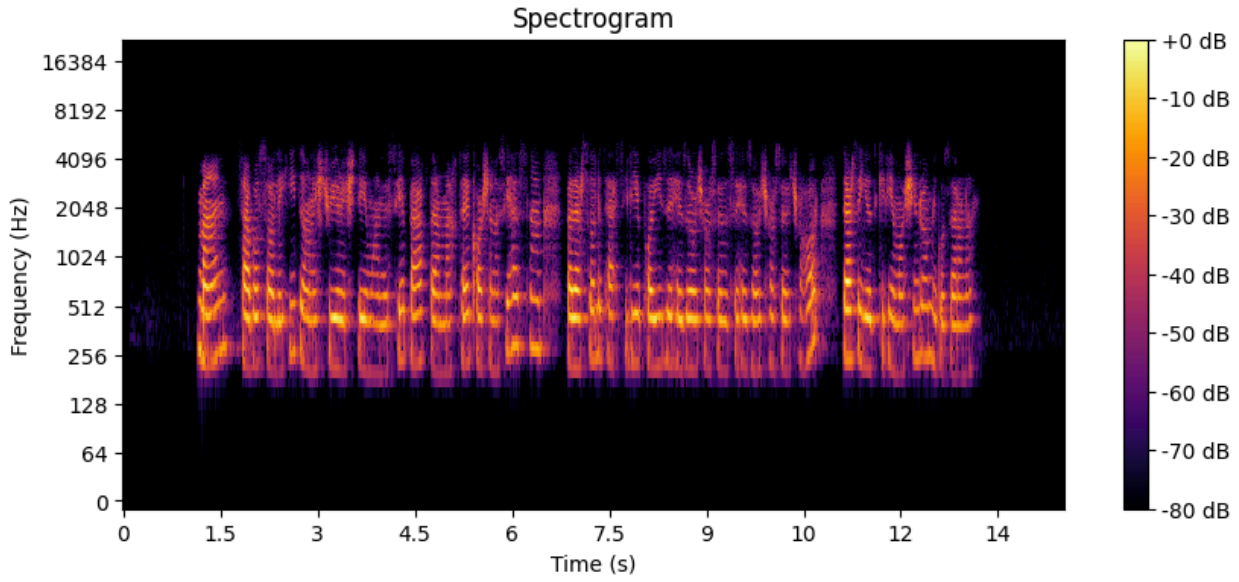
For using this function we should use a loop. Then we print some of the plots as we have many voices.

```
index = 0
for file, (y, sr) in audio_data_identity1.items():
      if index % 20 == 0:
      plot_spectrogram(y, sr)
      index += 1
```

As an example:



Color on the plot indicates the signal's amplitude at each time-frequency point. Brighter colors (yellow/orange) represent higher amplitudes, while darker colors (purple) indicate lower amplitudes. From this plot, we can see the signal has a consistent frequency pattern, likely a periodic or rhythmic structure.



Like the previous signal, This one has periodic and repeating patterns. The higher-frequency components appear to vary more rapidly than in the previous spectrogram, suggesting more complexity or additional higher-frequency noise. The overall energy distribution seems more evenly spread across time, compared to the previous spectrogram, indicating a more consistent signal without sharp spikes or drops.

Flatten data

```
def flatten_features(df):
      new_df = df.copy()
      for col in df.columns:
      if isinstance(df[col][0], np.ndarray):
            arr_shape = df[col][0].shape
            if len(arr_shape) == 1:
                  for i in range(arr_shape[0]):
                  new_df[f"{col}_{i}"] = df[col].apply(lambda x: x[i] if
len(x) > i else 0)
            else:
                  new_df[f"{col}_mean"]     =     df[col].apply(lambda     x:
np.mean(x))
                  new_df[f"{col}_std"] = df[col].apply(lambda x: np.std(x))
                  new_df[f"{col}_max"] = df[col].apply(lambda x: np.max(x))
            new_df.drop(columns=[col], inplace=True)
      return new_df
```

This function takes a DataFrame and flattens any columns containing NumPy arrays into separate scalar features. It loops through all columns in the DataFrame. It assumes the first element represents the column's data type. If the first element is a NumPy array, it processes that column. If the array is one-dimensional, it creates new columns for each element in the array. If an array has n elements, it creates n new columns: col_0, col_1, ..., col_n-1. If an array is shorter in some rows, missing indices are filled with 0. If the array is more than 1D (e.g., an image matrix), it computes statistical features:

- Mean (col_mean)
- Standard deviation (col_std)
- Maximum value (col_max)

This reduces dimensionality while preserving meaningful information.

Why Should We Flatten Data in Machine Learning?

Flattening is important in machine learning because many models expect structured, numerical features instead of arrays or matrices. Here's why:

1. Most ML Models Expect Tabular Data: Models like Logistic Regression, SVM, and Random Forest require numerical vectors as input. Flattening ensures compatibility.
2. Reduces Complexity: Multi-dimensional arrays increase computational cost.
   By extracting key statistics (mean, std, max), we make the data more manageable.
3. Avoids Issues with Inconsistent Array Sizes: Some rows may have different array sizes.
   The function handles this by creating fixed-size features.
4. Helps Feature Engineering: Aggregated values like mean and max capture essential information. This simplifies feature selection and improves model interpretability.

Scatter Plot

The feature space is more than 3-dimensional which is the maximum number of dimensions understandable by human vision. We need to apply dimensionality reduction methods like PCA to map the feature space into 2 or 3 dimensions.

PCA

PCA uses a linear mapping to convert the covariance matrix into a diagonal matrix. In other words, the features would be uncorrelated with one another. Values on the main diagonal of the resulting scatter correspond to the eigen-values of the original features matrix. In other words, variance of new features are eigen-values of original features. As you know, in general, a good feature has more separability. To find such features we can sort eigen-values in descending order and take from first of the list. That's where the features with more variance take place. So, PCA separates n top ones if n<d, otherwise it takes all the d features. Here we separated the first two principal components for a 2-dimensional scatter plot. Here's a short

Mathematical explanation:

$$S = (N-1) \times \sum_{i=1}^{N} (x_i - \hat{\mu})(x_i - \hat{\mu})^T$$

$$SV = VD \rightarrow S = VDV^T \ and \ D = V^T SV$$

$$\bar{x} = V^T x \rightarrow \bar{\mu} = V^T \mu$$

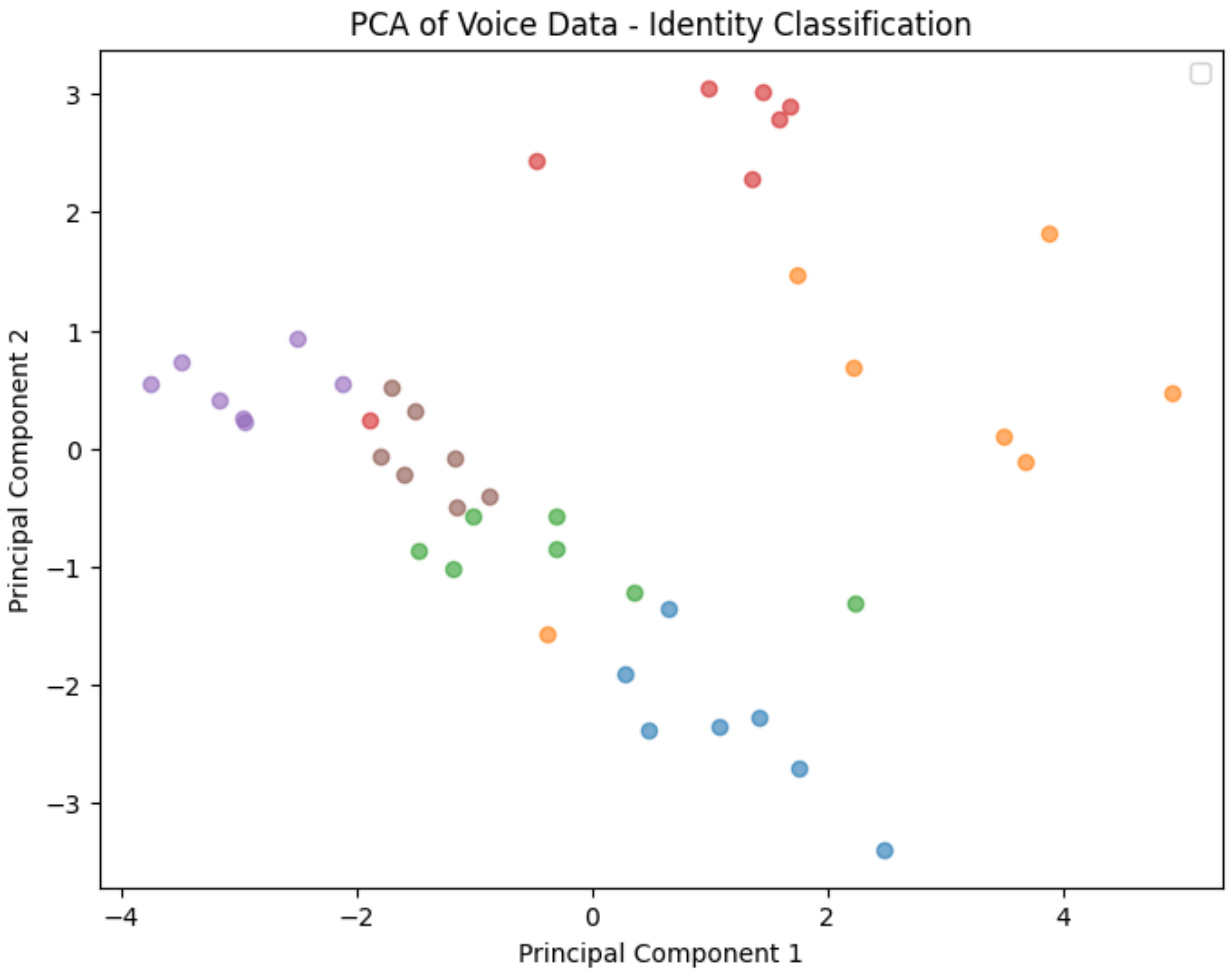
$$\bar{S} = (N-1) \times \sum_{i=1}^{N} (V^T x_i - V^T \hat{\mu})(x_i^T V - \hat{\mu}^T V) \quad = V^T (N-1) \times \sum_{i=1}^{N} (x_i - \hat{\mu})(x_i - \hat{\mu})^T V = V^T S V = D$$

Where:

- V is the vector of eigen-vectors. It's an orthogonal matrix V-1=VT
- D is a diagonal matrix containing eigen-values.

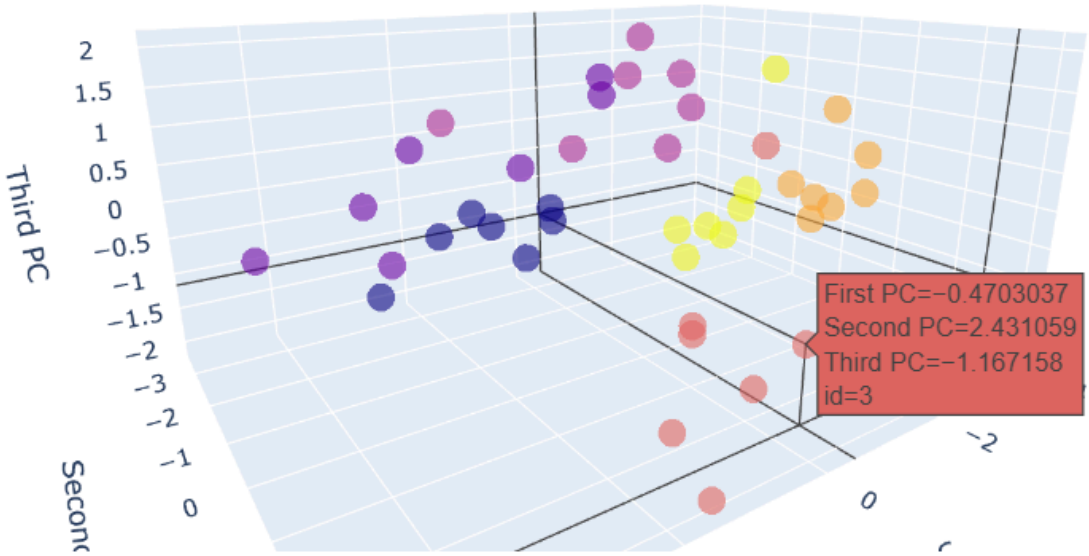
```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
plt.figure(figsize=(8, 6))
for label in np.unique(y):
      plt.scatter(X_pca[y == label, 0], X_pca[y == label, 1],alpha=0.6)
plt.title('PCA of Voice Data - Identity Classification')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()
```

x-axis and y-axis are the first two components and data points are colored based on student ID's. As we can see, audio signals belonging to the same person take place close together.

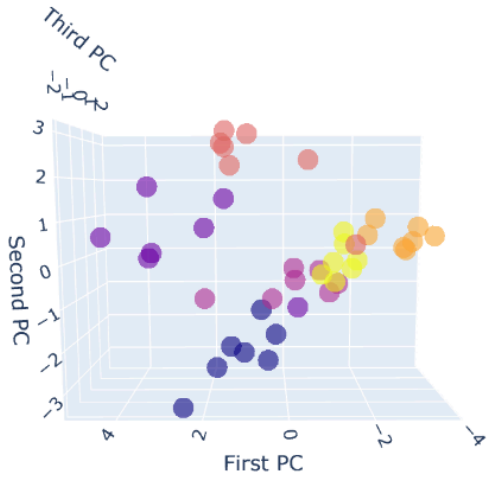


Let's try that again for the first three principal components. To visualize data points on a three dimensional scatter plot we used `plotly.express`.

```
pca = PCA(n_components=3)
X_pca = pca.fit_transform(X)
df_pca = pd.DataFrame(X_pca, columns=['PC1', 'PC2', 'PC3'])
df_pca['id'] = y
fig = px.scatter_3d(df_pca, x='PC1', y='PC2', z='PC3', color='id',
                    title='PCA of Voice Data - Identity Classification (3D)',
                    labels={'PC1': 'First PC', 'PC2': 'Second PC', 'PC3':
'Third PC'},
                    opacity=0.6)
fig.show()
```



If we look at the 3D PCA plot by fixing the third principal component to a constant value, we would obtain a 2D plot that corresponds to the 2D PCA version, showing the data in the plane formed by PC1 and PC2. Something like this point of view:



Classification – Supervised Learning

In this part, we perform gender classification and closed-set authentication on different subsets of the dataset, each with several models (Logistic Regression, KNN, SVM).

Gender Classification

The goal is to train a model to classify new unseen data as man or woman. We use features extracted from the previous part. First we separate the y vector from the X matrix. Then, 25% of the dataset is set apart as a test set and the rest will be used as a training set. Before training any model, we need to normalize data. Since we do not know the probability distribution of each feature, we have to estimate its parameters using a set of available samples (training set). For each of the models, we reported the results separated by class, overall result, the confusion matrix and ROC curve with AUROC.

Logistic Regression

Logistic Regression is a supervised learning algorithm used for classification tasks. Despite its name, it is a classification algorithm, not a regression algorithm. It predicts the probability of an instance belonging to a particular class.
Logistic Regression estimates the probability of an outcome using the sigmoid function (logistic function):

$$P(y = 1|X) = \frac{1}{1 + e^{-(wX+b)}}$$

Where:

- w = weight (coefficient)
- X = input features
- b = bias term

Why did we use logistic regression?

Using Logistic Regression for gender classification from a voice dataset can be a good choice, depending on the nature of your data and problem requirements.

Binary Classification Problem

Since gender classification is a binary classification task (Male vs. Female), Logistic Regression is a natural fit because:

- It directly models the probability of an instance belonging to one class.
- It outputs a probability score, allowing for confidence-based decision-making

Interpretability

Logistic Regression provides easily interpretable coefficients, which help in understanding which voice features (e.g., pitch, formants, MFCCs) are more important for distinguishing between male and female voices.

Computational Efficiency

Logistic Regression is lightweight and fast, making it ideal for datasets that do not require heavy computation. It works well with small to moderately large datasets.

Works Well with Linearly Separable Data

If the extracted voice features (e.g., pitch, fundamental frequency, MFCCs) are linearly separable between genders, Logistic Regression performs well.

Code of Logistic Regression

```
df = flatten_features(df_gender)
X = df.drop(columns=['Gender'])
y = df['Gender']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
y_pred_logreg = log_reg.predict(X_test)
print("Logistic Regression Results:")
print(classification_report(y_test, y_pred_logreg))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_logreg))
y_prob = log_reg.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

This code performs gender classification using Logistic Regression on a processed dataset. First, it flattens array-like features in `df_gender` using the `flatten_features` function. The dataset is then split into features (X) and labels (y), where Gender is the target variable. Using train-test splitting (75%-25%), the data is partitioned into training and testing sets. To ensure proper feature scaling, StandardScaler normalizes `X_train` and `X_test.` A Logistic Regression model is then trained on the processed `X_train` and evaluated on `X_test`. The classification results, including a classification report and confusion matrix, are printed to assess model performance. Finally, the ROC curve is plotted to visualize the model's ability to distinguish between classes, with the AUC (Area Under the Curve) providing a performance metric. This approach ensures a well-structured pipeline for binary classification.

Result of Logistic Regression

The result of this model is as follows:

```
Logistic Regression Results:
              precision    recall  f1-score   support

           0       0.71      0.68      0.69        25
           1       0.75      0.77      0.76        31

    accuracy                           0.73        56
   macro avg       0.73      0.73      0.73        56
weighted avg       0.73      0.73      0.73        56

Confusion Matrix:
[[17  8]
 [ 7 24]]
```

Classification Report

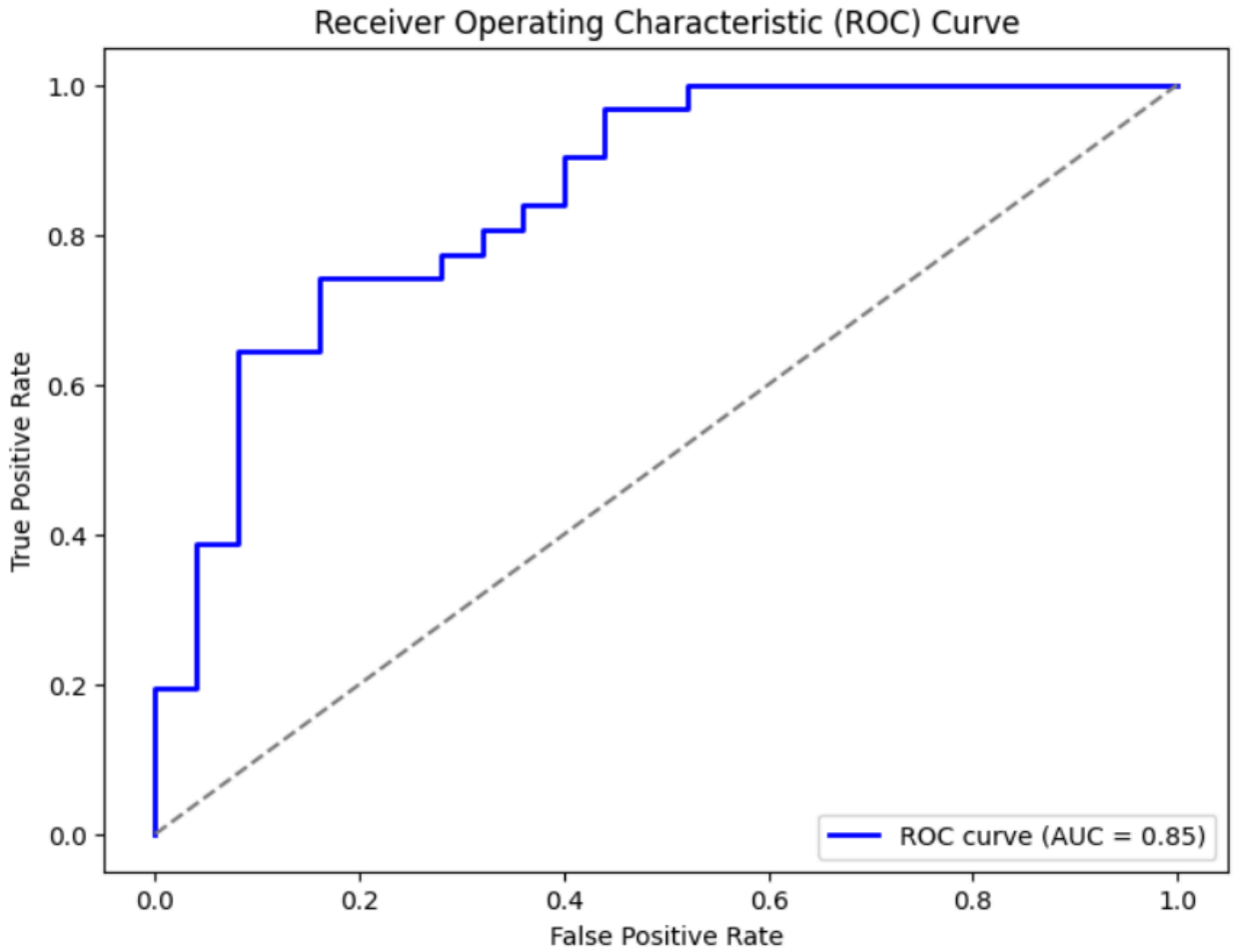
- Precision (for class 1 = 0.75): When the model predicts class 1, it is correct 75% of the time.
- Recall (for class 1 = 0.77): Out of all actual class 1 instances, the model correctly identifies 77%.
- F1-score balances precision and recall, showing class 1 has slightly better classification performance.
- Overall Accuracy: 73% (Correct predictions out of total 56 test samples)
- Macro Avg (Unweighted Mean): 73% (Average of both classes)
- Weighted Avg (Considering Class Imbalance): 73% (Weighted based on the number of samples per class)

Confusion Matrix

- True Positives (TP) = 24 → Correctly classified class 1 instances.
- True Negatives (TN) = 17 → Correctly classified class 0 instances.
- False Positives (FP) = 8 → Class 0 misclassified as class 1.
- False Negatives (FN) = 7 → Class 1 misclassified as class 0.

- Class 1 has slightly better performance than Class 0. The recall for class 0 (0.68) is lower, meaning some class 0 instances are misclassified.

ROC



- The x-axis represents the False Positive Rate (FPR), which indicates how often negative samples are incorrectly classified as positive.
- The y-axis represents the True Positive Rate (TPR) or Recall, showing how well the model identifies actual positives.
- The diagonal dashed line represents a random classifier (AUC = 0.5), meaning no predictive power.

KNN

The K-Nearest Neighbors (KNN) algorithm is a supervised machine learning algorithm used for classification and regression tasks. It is non-parametric, meaning it does not assume any specific distribution of the data.

KNN classifies a new data point based on the majority vote of its k-nearest neighbors in the feature space. The key steps are:

1. Choose a value for "k" (number of neighbors).
2. Calculate the distance between the new data point and all existing data points in the dataset. Common distance metrics include:
    - Euclidean Distance (most commonly used)
    - Manhattan Distance
    - Minkowski Distance
3. Find the k-nearest neighbors (the k closest points in the dataset).
4. Assign the class label based on the majority vote among the k neighbors.
5. The model makes the prediction.

Why did we use KNN?

Using K-Nearest Neighbors (KNN) for gender classification from voice data can be beneficial due to several reasons:

KNN is a Simple and Effective Classifier

KNN is a non-parametric algorithm, meaning it makes no assumptions about the underlying data distribution. It classifies a sample based on the majority vote of its k-nearest neighbors, making it intuitive and easy to implement.

Suitable for Voice Feature-Based Classification

Voice datasets typically contain features like pitch, formants, MFCCs (Mel-Frequency Cepstral Coefficients), and spectral properties, which can form distinct clusters for male and female voices. KNN is effective when there is a clear distinction between classes, as seen in gender-based voice features.

Works Well with Small to Medium-Sized Datasets

If your dataset is not extremely large, KNN can be a good choice since it doesn't require training—it just stores the dataset and classifies based on similarity. Unlike deep learning models, it doesn't need long training times or large labeled datasets.

Distance-Based Classification is Effective for Voice Data

KNN relies on distance metrics (e.g., Euclidean distance, Manhattan distance) to classify data points. Since male and female voices often have distinct frequency and pitch distributions, distance-based separation works well.

Adjustable Model Complexity

The number of neighbors (k) can be tuned to balance bias and variance:

- Low k (e.g., 1-3) → More sensitive to noise (overfitting).
- Higher k (e.g., 5-10) → More generalized predictions (better stability).

Code of KNN

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
print("KNN Results:")
print(classification_report(y_test, y_pred_knn))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_knn))
y_prob = knn.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

This code implements the K-Nearest Neighbors (KNN) algorithm for a binary classification task. It begins by creating a KNN classifier with 5 neighbors and trains it on the training dataset (X_train, y_train). After training, the model predicts labels for the test set (X_test) and evaluates performance using a classification report and confusion matrix, which measure accuracy, precision, recall, and F1-score. Additionally, the code computes predicted probabilities (predict_proba) for the positive class and generates a Receiver Operating Characteristic (ROC) curve, which plots the true positive rate (TPR) vs. false positive rate (FPR). The Area Under the Curve (AUC) value is calculated to assess the model's ability to distinguish between classes, where a higher AUC indicates better performance. Finally, the ROC curve is plotted, comparing the model's performance against a random classifier.

Result of KNN

Result of KNN is as follows:

```
KNN Results:
              precision    recall  f1-score   support

           0       0.73      0.64      0.68        25
           1       0.74      0.81      0.77        31

    accuracy                           0.73        56
   macro avg       0.73      0.72      0.73        56
weighted avg       0.73      0.73      0.73        56

Confusion Matrix:
[[16  9]
 [ 6 25]]
```

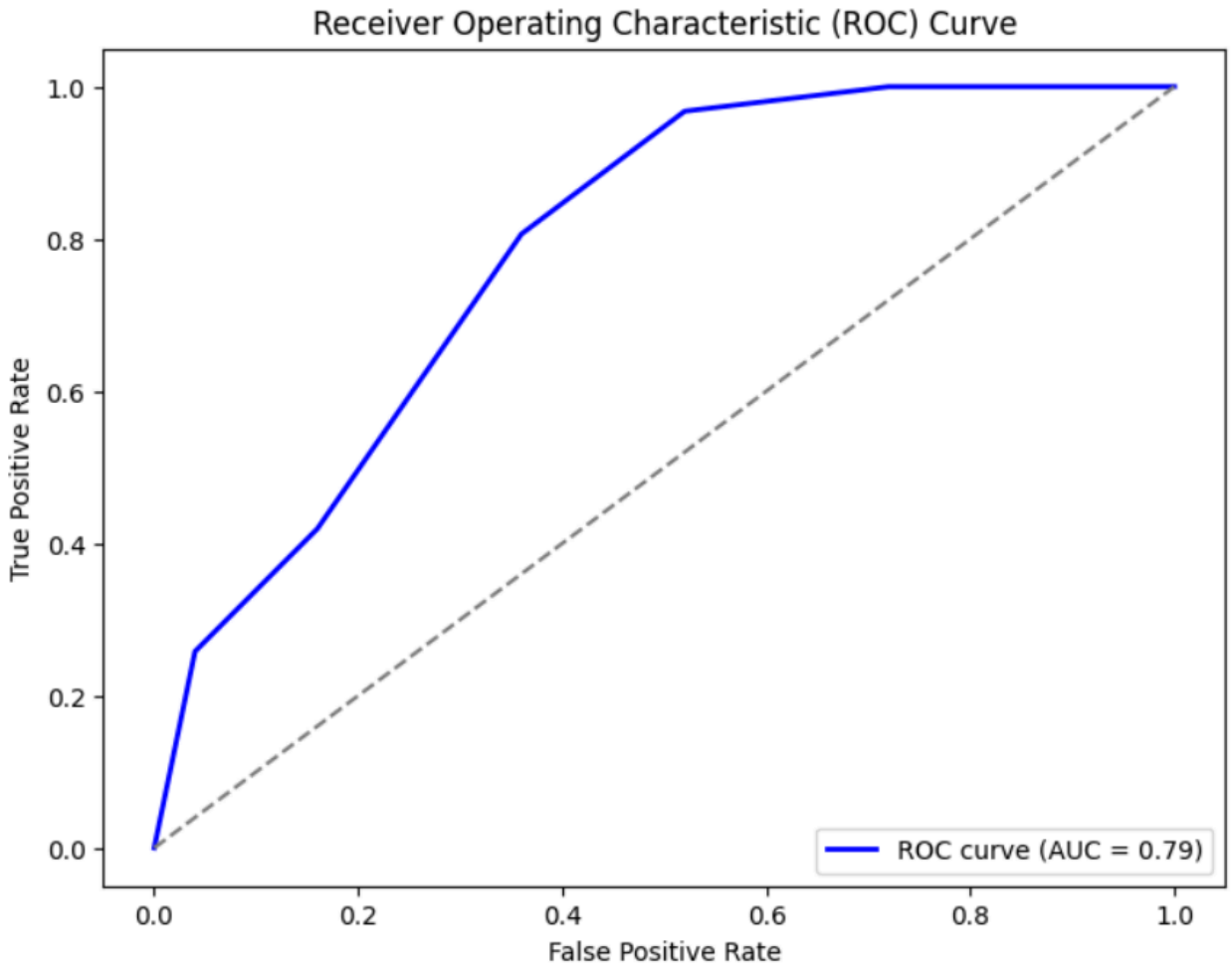
Classification Report

- Accuracy: 73% (Correct predictions out of all predictions)
- Macro Avg Precision & Recall: 73% & 72% (Balanced measure across both classes)
- Weighted Avg F1-Score: 73% (Overall balance between precision & recall)
- Class 0:
    - Precision: 0.73 → When predicting class 0, 73% of predictions are correct.
    - Recall: 0.64 → Only 64% of actual class 0 samples were correctly identified.
    - F1-Score: 0.68 → A balance between precision and recall (not too strong).
- Class 1:
    - Precision: 0.74 → When predicting class 1, 74% of predictions are correct.
    - Recall: 0.81 → 81% of actual class 1 samples were correctly identified.
    - F1-Score: 0.77 → Better performance than class 0.

Confusion Matrix

- False Negatives (FN) = 9: Class 0 misclassified as class 1 (missed detections).
- False Positives (FP) = 6: Class 1 misclassified as class 0 (wrong gender assignment).

ROC



This ROC (Receiver Operating Characteristic) curve evaluates the performance of the K-Nearest Neighbors (KNN) classifier. The blue curve represents the trade-off between the true positive rate (TPR) and false positive rate (FPR) at various classification thresholds. The AUC (Area Under the Curve) is 0.79, which suggests that the model has a reasonable ability to distinguish between the two classes, though it is not perfect. An AUC of 0.79 indicates that the classifier correctly differentiates between positive and negative classes 79% of the time, which is decent but could be improved. Compared to a random classifier (AUC = 0.5, shown by the dashed line),

this model performs significantly better, but it still leaves room for enhancement, possibly by tuning the hyperparameters or using feature selection techniques.

SVM

Support Vector Machine (SVM) is a powerful supervised learning algorithm used for classification and regression tasks. It works by finding the optimal hyperplane that best separates different classes in the feature space. The key idea behind SVM is to maximize the margin between the closest data points (support vectors) of different classes. This helps improve the generalization of the model.

SVM can work with linear and non-linear data. If the data is linearly separable, SVM finds a straight hyperplane. For non-linear cases, SVM uses kernel functions (such as polynomial, radial basis function (RBF), or sigmoid) to transform the data into a higher-dimensional space where a linear separation is possible.

SVM is particularly useful for high-dimensional spaces and is effective in handling small datasets with a clear margin of separation. However, it can be computationally expensive for very large datasets and may require careful tuning of hyperparameters like the kernel type and regularization parameter (C).

Why did we use SVM?

Using Support Vector Machine (SVM) for gender classification from a voice dataset can be a great choice for several reasons:

Effective in High-Dimensional Spaces

Voice data often has multiple extracted features (e.g., MFCCs, pitch, formants), making the feature space high-dimensional. SVM is well-suited for handling such data efficiently.

Robust to Small and Medium-Sized Datasets

SVM performs well when the dataset is not extremely large, making it a strong candidate for voice-based gender classification where datasets may be limited.

Handles Non-Linearly Separable Data

Voice features may not always be linearly separable. By using kernel functions (e.g., RBF kernel), SVM can transform the data into a higher-dimensional space where it finds a better decision boundary.

Code of SVM

```
X = df.drop(columns=['Gender'])
y = df['Gender']
scaler = StandardScaler()
X = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)
model = SVC(kernel='linear', probability=True)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")
print(classification_report(y_test, y_pred))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
y_prob = model.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='blue', lw=2, label=f'ROC curve (AUC =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```

This code performs gender classification using a Support Vector Machine (SVM) with a linear kernel. First, it separates the features (X) from the target variable (y, representing gender) and standardizes the features using StandardScaler to normalize the data. It then splits the dataset into training (75%) and testing (25%) sets. The SVM model is trained using the linear kernel, and predictions are made on the test set. The model's accuracy is computed and printed, along with a classification report (showing precision, recall, and F1-score) and a confusion matrix

(showing correct and incorrect classifications). To evaluate the model further, it calculates ROC (Receiver Operating Characteristic) and AUC (Area Under the Curve) scores to measure the model's ability to distinguish between classes. Finally, the ROC curve is plotted, illustrating the trade-off between the true positive rate (sensitivity) and the false positive rate.

Result of SVM

Classification Resort

```
Model Accuracy: 0.77
              precision    recall  f1-score   support

           0       0.94      0.57      0.71        28
           1       0.69      0.96      0.81        28

    accuracy                           0.77        56
   macro avg       0.82      0.77      0.76        56
weighted avg       0.82      0.77      0.76        56

Confusion Matrix:
[[16 12]
 [ 1 27]]
```

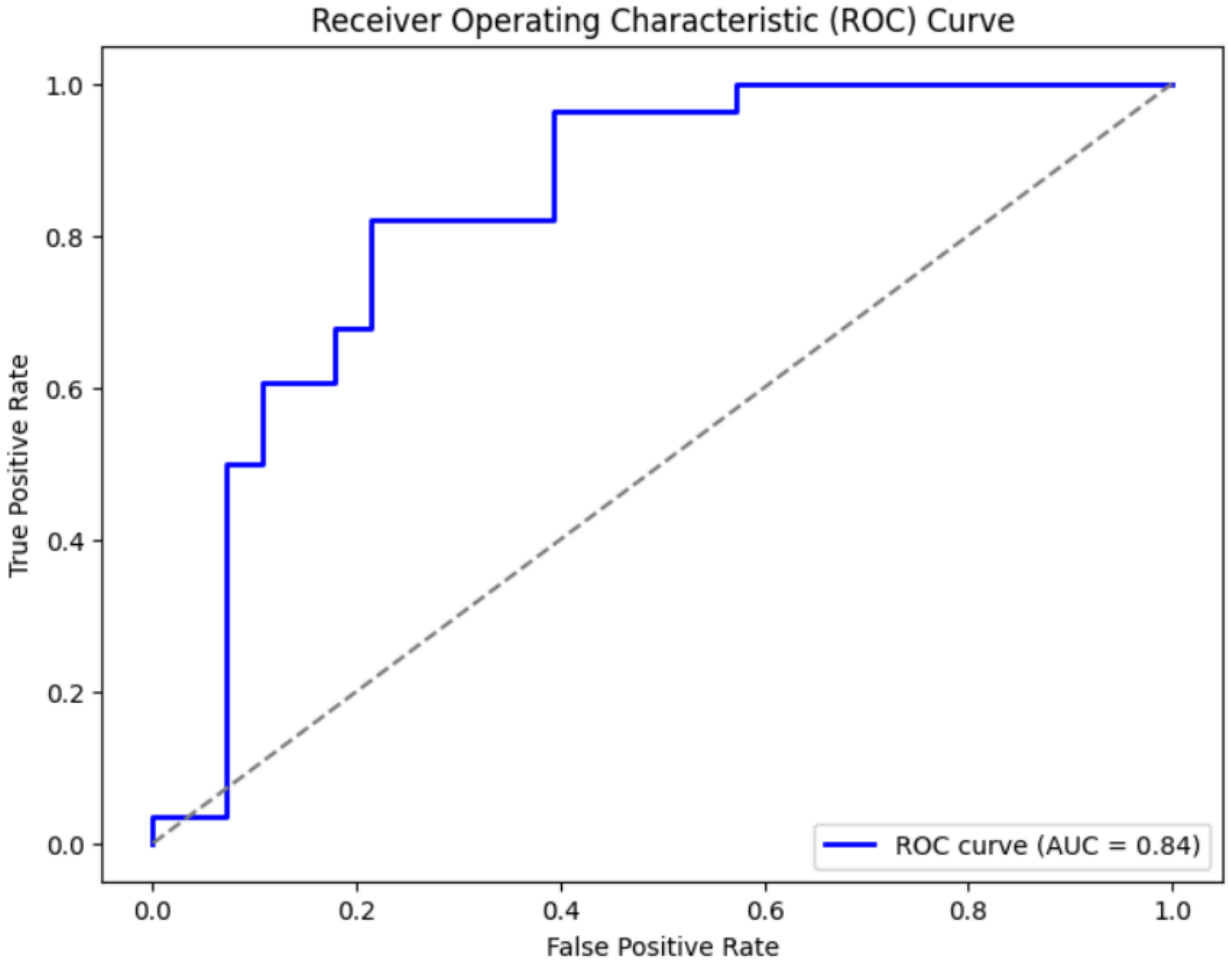
- Accuracy: 0.77 → The model correctly predicted 77% of the total instances.
- Precision (TP / (TP + FP)):
    - Class 0: 0.94 (high; very few false positives)
    - Class 1: 0.69 (lower; more false positives)
- Recall (TP / (TP + FN)):
    - Class 0: 0.57 (low; many false negatives)
    - Class 1: 0.96 (high; most positive cases are detected correctly)
- F1-score (harmonic mean of precision & recall):
    - Class 0: 0.71
    - Class 1: 0.81

Confusion Matrix

- True Positives (TP): 27 (Correctly classified as class 1)

- True Negatives (TN): 16 (Correctly classified as class 0)
- False Positives (FP): 12 (Incorrectly classified as class 1)
- False Negatives (FN): 1 (Incorrectly classified as class 0)

ROC



- The model has good discrimination ability (AUC = 0.84).
- There is a reasonable balance between True Positives and False Positives.

Identity Classification

The goal is to train a model to classify new unseen data as one of the previously observed students. We use a set containing the voices of 10 different students. Like the previous task we separate the y vector from the X matrix. Then, the dataset is set apart as a test set and the rest

will be used as a training set. Before training any model, we need to normalize data. It's just like the previous task. For each of the models, we reported the results separated by class, overall result, the confusion matrix and ROC curve with AUROC.

As we want to use a closed-set authentication, we use a self defined function which separates the train and test in a way that we have one person's voices in both train and test datasets.

```
def split_identity(df3):
      df = df3.copy()
      df['idx'] = df.index
      df['idx'] = df['idx'].apply(lambda x: 1 if x % 7 == 0 else 0)
      df1 = df[df['idx'] == 0]
      df2 = df[df['idx'] != 0]
      df1 = df1.drop(columns=['idx']).reset_index().drop(columns=["index"])
      df2 = df2.drop(columns=['idx']).reset_index().drop(columns=["index"])
      return [df1, df2]
```

Why did we use logistic regression?

Fast Training & Low Computational Cost

Works well when the dataset is linearly separable and training speed is a priority.

Probability Outputs

Unlike KNN and SVM, logistic regression provides probability scores, which can be useful for setting confidence thresholds.

Feature Importance Analysis

It allows an understanding of which voice features contribute most to classification.

Code of Logistic Regression

```
trainn, testt = split_identity(df3 = df_identity1)
X_train = trainn[[col for col in trainn.columns if col != 'id']]
X_test = testt[[col for col in testt.columns if col != 'id']]
y_train = trainn[['id']]
y_test = testt[['id']]
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
```

```
X_test = scaler.transform(X_test)
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
y_pred_logreg = log_reg.predict(X_test)
print("Logistic Regression Results:")
print(classification_report(y_test, y_pred_logreg))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_logreg))
```

This code performs identity classification using Logistic Regression on a dataset (df_identity1). It first splits the dataset into training (trainn) and testing (testt) sets using split_identity(). It then separates the features (X_train, X_test) from the target variable (y_train, y_test), where the 'id' column represents identity labels. The feature data is standardized using StandardScaler to ensure consistent scaling. A Logistic Regression model (log_reg) is then trained on the scaled training data and used to make predictions (y_pred_logreg) on the test set. Finally, the model's performance is evaluated using classification_report() (showing precision, recall, and F1-score) and confusion_matrix() (showing the count of correct and incorrect predictions).

Result of Logistic Regression

Identity 1:

```
Logistic Regression Results:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1
           2       1.00      1.00      1.00         1
           3       1.00      1.00      1.00         1
           4       1.00      1.00      1.00         1
           5       1.00      1.00      1.00         1

    accuracy                           1.00         6
   macro avg       1.00      1.00      1.00         6
weighted avg       1.00      1.00      1.00         6

Confusion Matrix:
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
```

Classification Report

- The model achieved perfect precision, recall, and F1-score (1.00) across all classes (0 to 5).
- Accuracy is 1.00 (100%), indicating that the model correctly classified all instances.
- Support column shows that each class had only one sample (6 total samples).
- Macro Avg & Weighted Avg are also 1.00, confirming perfect classification across all classes.

Confusion Matrix

- Each row represents actual class labels.
- Each column represents predicted class labels.
- All values along the diagonal are 1, indicating that every sample was classified correctly with zero misclassifications.

Identity 2:

```
Logistic Regression Results:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1
           2       1.00      1.00      1.00         1
           3       1.00      1.00      1.00         1
           4       1.00      1.00      1.00         1
           5       1.00      1.00      1.00         1

    accuracy                           1.00         6
   macro avg       1.00      1.00      1.00         6
weighted avg       1.00      1.00      1.00         6

Confusion Matrix:
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
```

Identity 3:

```
Logistic Regression Results:
              precision    recall  f1-score   support

           0       0.50      1.00      0.67         1
           1       1.00      1.00      1.00         1
           2       0.00      0.00      0.00         1
           3       0.00      0.00      0.00         1
           4       1.00      1.00      1.00         1
           5       0.50      1.00      0.67         1

    accuracy                           0.67         6
   macro avg       0.50      0.67      0.56         6
weighted avg       0.50      0.67      0.56         6

Confusion Matrix:
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [1 0 0 0 0 0]
 [0 0 0 0 0 1]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
```

KNN

Why did we use KNN?

Non-Parametric & Instance-Based Learning

KNN makes predictions based on the closest labeled examples, making it useful for voice data, where similar voices have similar feature vectors.

Works Well with Small & Medium-Sized Datasets

If the dataset isn't too large, KNN can effectively classify identities without extensive training.

Adaptability to Complex Patterns

Voice features can have non-linear relationships, and KNN naturally handles non-linearity.

Code of KNN

```
trainn, testt = split_identity(df3 = df_identity1)
X_train = trainn[[col for col in trainn.columns if col != 'id']]
X_test = testt[[col for col in testt.columns if col != 'id']]
y_train = trainn[['id']]
y_test = testt[['id']]
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
print("KNN Results:")
print(classification_report(y_test, y_pred_knn))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_knn))
```

This code performs identity classification using K-Nearest Neighbors (KNN) on a dataset (df_identity1). It first splits the data into training (trainn) and testing (testt) sets using split_identity(). The features (X_train, X_test) are separated from the target variable (y_train, y_test), where the 'id' column represents identity labels. The feature data is then standardized using StandardScaler to ensure that all features have a uniform scale. A KNN classifier with n_neighbors=5 is trained on the scaled training data and used to make predictions (y_pred_knn) on the test set. Finally, the model's performance is evaluated using classification_report() (providing precision, recall, and F1-score) and confusion_matrix() (showing correct and incorrect classifications).

Result of KNN

Identity 1:

```
KNN Results:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1
           2       1.00      1.00      1.00         1
           3       1.00      1.00      1.00         1
           4       1.00      1.00      1.00         1
           5       1.00      1.00      1.00         1

    accuracy                           1.00         6
   macro avg       1.00      1.00      1.00         6
weighted avg       1.00      1.00      1.00         6

Confusion Matrix:
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
```

Classification Report:

- Precision: How many of the predicted labels for a class were correct.
- Recall: How many of the actual labels for a class were correctly identified.
- F1-score: The harmonic mean of precision and recall.
- Support: The number of true instances for each class.

Confusion Matrix

- indicates that every class was perfectly classified without any misclassification.

Identity 2:

```
KNN Results:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1
           2       0.00      0.00      0.00         1
           3       1.00      1.00      1.00         1
           4       0.50      1.00      0.67         1
           5       0.00      0.00      0.00         1

    accuracy                           0.67         6
   macro avg       0.58      0.67      0.61         6
weighted avg       0.58      0.67      0.61         6

Confusion Matrix:
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 0 0 0 1]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]
 [0 0 0 0 1 0]]
```

Identity 3:

```
KNN Results:
              precision    recall  f1-score   support

           0       0.50      1.00      0.67         1
           1       1.00      1.00      1.00         1
           2       0.00      0.00      0.00         1
           3       0.00      0.00      0.00         1
           4       1.00      1.00      1.00         1
           5       0.50      1.00      0.67         1

    accuracy                           0.67         6
   macro avg       0.50      0.67      0.56         6
weighted avg       0.50      0.67      0.56         6

Confusion Matrix:
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [1 0 0 0 0 0]
 [0 0 0 0 0 1]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
```

SVM

Why did we use SVM?

Effective in High-Dimensional Spaces

Voice data is often transformed into feature vectors using techniques like MFCC (Mel-Frequency Cepstral Coefficients), Spectrograms, or Mel-Spectrograms. SVM works well with high-dimensional feature spaces, making it suitable for voice-based identity recognition.

Robust to Small Datasets

If your dataset is relatively small, deep learning models like CNNs may overfit, while SVMs can generalize well with limited samples.

Handles Non-Linear Decision Boundaries

Voice data can have complex patterns, and Kernel SVM (using RBF, polynomial kernels, etc.) can model non-linear relationships between different speaker identities.

Works Well with Noisy Data

SVMs maximize the margin between different classes, making them robust against some level of noise in the dataset.

Code of SVM

```
trainn, testt = split_identity(df3=df_identity1)
X_train = trainn[[col for col in trainn.columns if col != 'id']]
X_test = testt[[col for col in testt.columns if col != 'id']]
y_train = trainn[['id']]
y_test = testt[['id']]
scaler = StandardScaler()
model = SVC(kernel='linear', probability=True)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")
print(classification_report(y_test, y_pred))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

This code performs identity recognition using an SVM classifier with a linear kernel. First, the dataset df_identity1 is split into training (trainn) and testing (testt) sets using split_identity(). The feature matrix (X_train, X_test) is extracted by removing the 'id' column, while the target variable (y_train, y_test) contains only the 'id' values (which represent different identities). A StandardScaler() is initialized (but not used in the code) for potential feature scaling. An SVM classifier (SVC) with a linear kernel is created and trained on X_train and y_train. The model then predicts identities on X_test, and its accuracy is evaluated using accuracy_score. Finally,

the model's performance is displayed using classification_report() and confusion_matrix(), which provide detailed metrics such as precision, recall, F1-score, and misclassifications.

Result of SVM

Identity 1:

```
Model Accuracy: 0.83
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1
           2       0.50      1.00      0.67         1
           3       1.00      1.00      1.00         1
           4       0.00      0.00      0.00         1
           5       1.00      1.00      1.00         1

    accuracy                           0.83         6
   macro avg       0.75      0.83      0.78         6
weighted avg       0.75      0.83      0.78         6

Confusion Matrix:
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 1 0 0]
 [0 0 1 0 0 0]
 [0 0 0 0 0 1]]
```

Classification Report

- The overall accuracy of the model is 0.83 (83%), meaning 83% of the test samples were correctly classified.
- Precision: The proportion of correctly predicted instances out of total predictions for that class.
- Recall: The proportion of correctly predicted instances out of actual occurrences of that class.
- F1-score: The harmonic mean of precision and recall.

- Support: The number of actual instances for each identity in the test set.
- Classes 0, 1, 3, and 5 have perfect precision, recall, and F1-score (1.00), meaning they were classified correctly.
- Class 2 has a precision of 0.50 but recall of 1.00, meaning all actual instances were detected, but some incorrect predictions were made.
- Class 4 has precision, recall, and F1-score of 0.00, indicating that the model failed to recognize any instances of this identity.
- Macro avg (simple average across classes) shows a precision of 0.75, recall of 0.83, and F1-score of 0.78.
- Weighted avg considers class imbalance but gives the same values as macro avg due to equal support.

Confusion Matrix

- The diagonal elements represent correctly classified instances.
- The off-diagonal elements indicate misclassifications.
- The row for class 4 has [0 0 0 0 0 0], meaning class 4 was never predicted.

Identity 2:

```
Model Accuracy: 1.00
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1
           2       1.00      1.00      1.00         1
           3       1.00      1.00      1.00         1
           4       1.00      1.00      1.00         1
           5       1.00      1.00      1.00         1

    accuracy                           1.00         6
   macro avg       1.00      1.00      1.00         6
weighted avg       1.00      1.00      1.00         6

Confusion Matrix:
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
```

Identity 3:

```
Logistic Regression Results:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1
           2       1.00      1.00      1.00         1
           3       1.00      1.00      1.00         1
           4       1.00      1.00      1.00         1
           5       1.00      1.00      1.00         1

    accuracy                           1.00         6
   macro avg       1.00      1.00      1.00         6
weighted avg       1.00      1.00      1.00         6

Confusion Matrix:
[[1 0 0 0 0 0]
 [0 1 0 0 0 0]
 [0 0 1 0 0 0]
 [0 0 0 1 0 0]
 [0 0 0 0 1 0]
 [0 0 0 0 0 1]]
```

Clustering – Unsupervised Learning

We aim to split data points into a chosen number of groups (clusters) without looking at their label.

Optimal Number of Clusters

To find it, we can perform a simple clustering algorithm like k-means on different values of k and compare them to find a good k. But how to determine which clustering is the best?

Silhouette Score

We use Silhouette Score to compare clusterings. It is a measure of how well each point in a cluster is matched to its own cluster compared to other clusters. It ranges from -1 to 1, where:

- A score close to +1 indicates that points are well clustered.
- A score close to 0 means the point is on or near the decision boundary between clusters.
- A score close to -1 indicates that points may have been assigned to the wrong cluster.

For a data point i, the silhouette score si is defined as:

$$s_i = \frac{b_i - a_i}{max(b_i, a_i)}$$

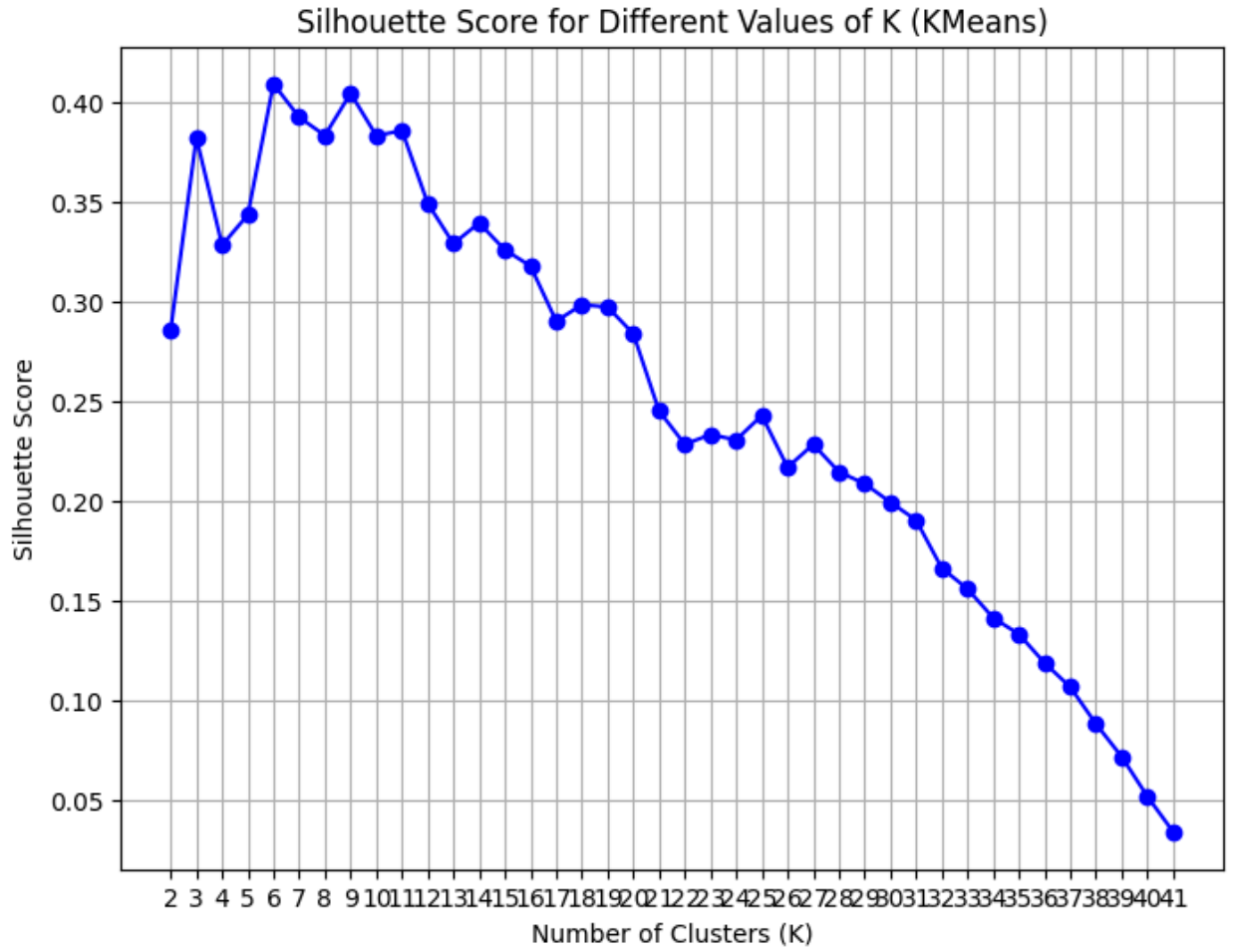
Where:

- ai is the average distance from point i to all other points in the same cluster.
- bi is the average distance from point i to all points in the nearest different cluster.

```
sil_scores = []
K_range = range(2, len(df))
for k in K_range:
      kmeans = KMeans(n_clusters=k, random_state=42)
      kmeans.fit(X)
      score = silhouette_score(X, kmeans.labels_)
      sil_scores.append(score)
plt.figure(figsize=(8, 6))
plt.plot(K_range, sil_scores, marker='o', linestyle='-', color='b')
plt.title('Silhouette Score for Different Values of K (KMeans)')
plt.xlabel('Number of Clusters (K)')
```

```
plt.ylabel('Silhouette Score')
plt.xticks(K_range)
plt.grid(True)
plt.show()


best_K = K_range[np.argmax(sil_scores)]
print(f"Best K (Number of Clusters) based on Silhouette Score: {best_K}")
```

Here's the plot of Silhouette Score for different values of k in range of 2 to number of data points. Notice that we performed clustering on the Identity detection dataset which contains 42 voices that are 7 voices of 6 different students. As shown below, maximum Silhouette Score happens at k=6, which is exactly the number of students and what we expected it to be.



Elbow Method

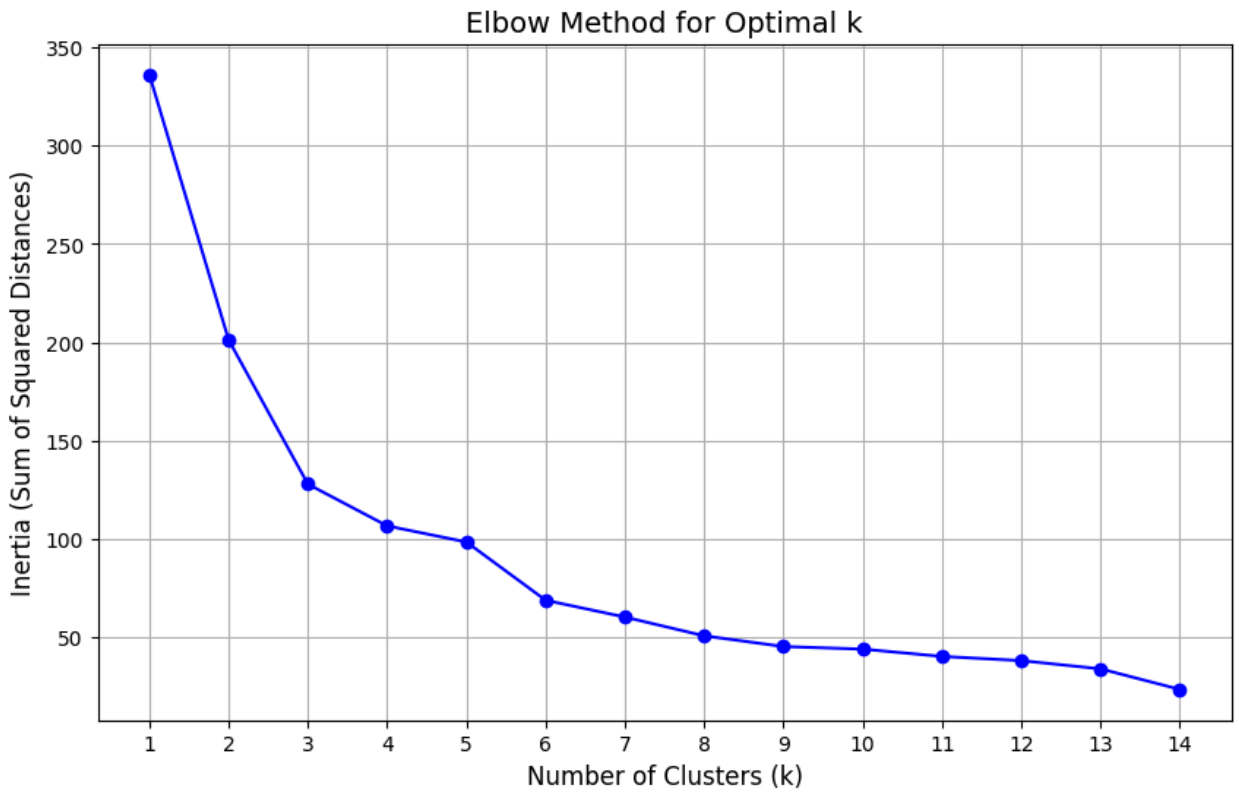
Interia tends to decrease as the number of clusters increase. But from some point, changes get lower and lower. The starting point of this process is called an elbow point. That's a good number of clusters to choose.

$$Interia = \sum_{i=1}^{n} \sum_{j=1}^{k} (y_i == j) \times ||x_i - c_j||^2$$

Where:

- n is the number of data points
- k is the chosen number of clusters
- y are the cluster labels
- x are the data points
- Leaving id and gender out to avoid

```
X = df.drop(columns=['id', 'Gender'])
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
k_range = range(1, len(df))
inertia_values = []
for k in k_range:
      kmeans = KMeans(n_clusters=k, random_state=42)
      kmeans.fit(X_scaled)
      inertia_values.append(kmeans.inertia_)
plt.figure(figsize=(10, 6))
plt.plot(k_range, inertia_values, marker='o', linestyle='-', color='b')
plt.title('Elbow Method for Optimal k', fontsize=14)
plt.xlabel('Number of Clusters (k)', fontsize=12)
plt.ylabel('Inertia (Sum of Squared Distances)', fontsize=12)
plt.grid(True)
plt.xticks(k_range)
plt.show()
```



This method confirms the result of the previous method as k=6 is the optimal value.

Identity Classification

K-Means

K-Means is an unsupervised machine learning algorithm used for clustering data points into K distinct groups based on their features. It is widely used in pattern recognition, customer segmentation, image compression, and anomaly detection.

How K-Means Works

1. Choose K (Number of Clusters): Decide the number of clusters the algorithm should form.
2. Initialize Centroids: Randomly select K initial cluster centroids from the dataset.
3. Assign Data Points to Clusters:
    - Compute the Euclidean distance between each data point and the centroids.
    - Assign each point to the nearest centroid, forming K clusters.
4. Update Centroids:
    - Compute the new centroid of each cluster by taking the mean of all points assigned to that cluster.
5. Repeat Steps 3 and 4:
    - Continue updating clusters and centroids until centroids no longer change (or a stopping condition is met, such as a max number of iterations).
6. Final Cluster Formation: The algorithm converges when cluster assignments stabilize.

Why did we use K-Means?

Using K-Means for identity recognition from a voice dataset is generally not the best choice, but it can be useful in specific scenarios. Here's when and why you might use K-Means for voice-based identity recognition:

When You Don't Have Labeled Data (Unsupervised Learning)

K-Means is an unsupervised algorithm, meaning it does not require labeled data.

When You Need Speaker Clustering Instead of Classification

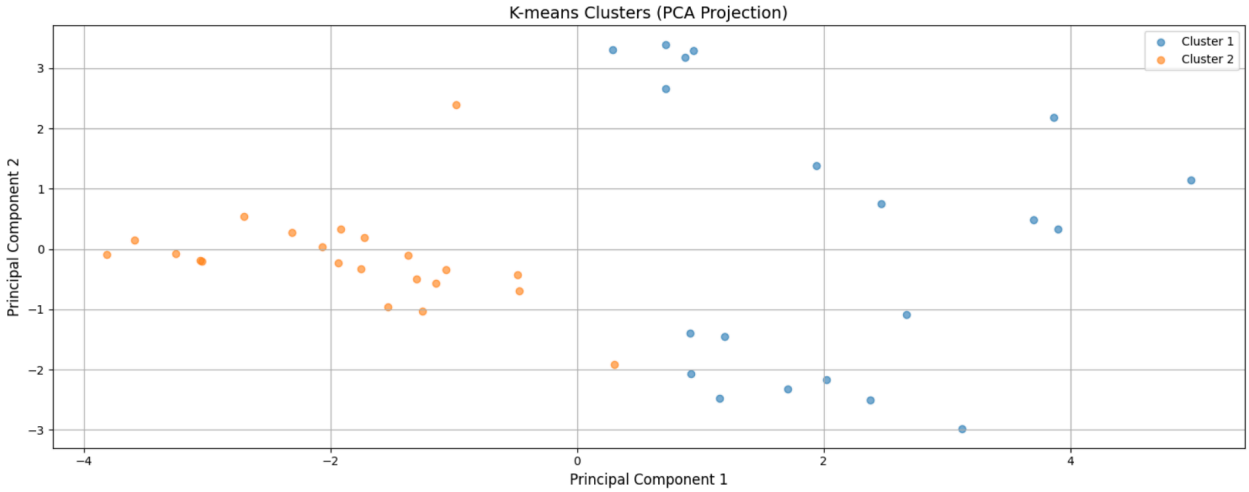
If the goal is to identify groups of speakers without knowing who they are, K-Means can help cluster similar voices together.

Code of K-Means

```
X = df_identity1.drop(columns=['id'])
y = df_identity1['id']
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
k = 2
kmeans = KMeans(n_clusters=k)
kmeans.fit(X_scaled)
df_identity1['cluster'] = kmeans.labels_
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
plt.figure(figsize=(15, 6))
for cluster in range(k):
      plt.scatter(X_pca[df_identity1['cluster']     ==     cluster,     0],
X_pca[df_identity1['cluster'] == cluster, 1],
                  label=f'Cluster {cluster + 1}', alpha=0.6)
plt.title('K-means Clusters (PCA Projection)', fontsize=14)
plt.xlabel('Principal Component 1', fontsize=12)
plt.ylabel('Principal Component 2', fontsize=12)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

kmeans_silhouette = silhouette_score(X_scaled, df_identity1['cluster'])
print(f"Silhouette Score for K-means clustering: {kmeans_silhouette}")
```

This code performs K-Means clustering on a voice identity dataset (df_identity1) to group similar voices into two clusters. First, it separates the features (X) from the identity labels (y), then applies standardization using StandardScaler() to normalize the feature values. The K-Means algorithm is initialized with k=2 clusters and fitted to the scaled data, assigning each data point a cluster label stored in the 'cluster' column. To visualize the clusters, Principal Component Analysis (PCA) reduces the feature space to two principal components, making it easier to plot in 2D. A scatter plot is generated to show the clustered data points with different colors. Finally, the Silhouette Score is calculated to measure the clustering quality, where a

higher score (closer to 1) indicates well-separated and compact clusters. This approach is useful for unsupervised speaker clustering when identity labels are unknown.

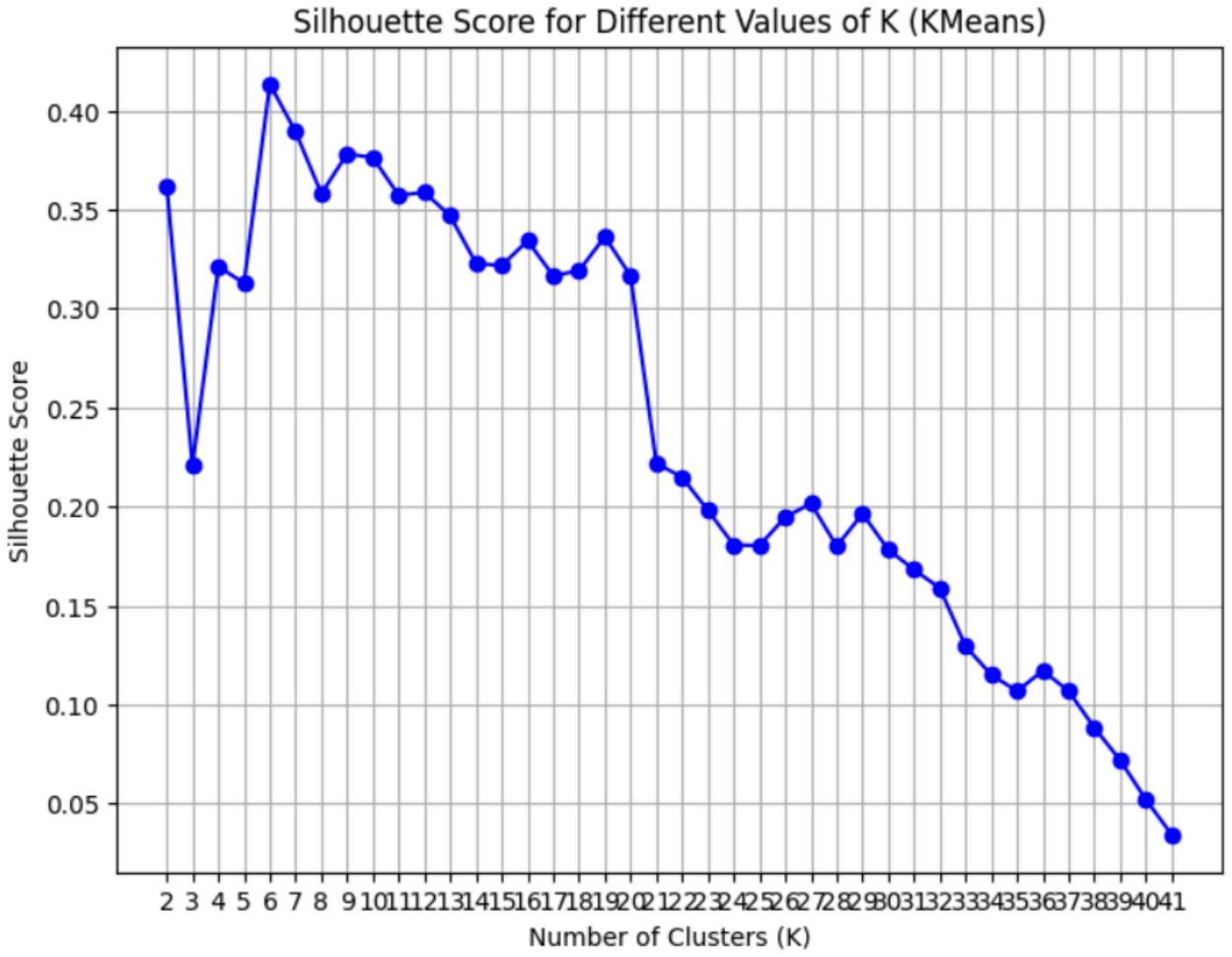


Silhouette Score for K-means clustering: 0.3502710683354807

```
sil_scores = []
K_range = range(2, 42)
for k in K_range:
      kmeans = KMeans(n_clusters=k, random_state=42)
      try:
      kmeans.fit(X_scaled)
      score = silhouette_score(X_scaled, kmeans.labels_)
      sil_scores.append(score)
      except ValueError as e:
      print(f"Error during clustering with k={k}: {e}")
      sil_scores.append(np.nan)
plt.figure(figsize=(8, 6))
plt.plot(K_range, sil_scores, marker='o', linestyle='-', color='b')
plt.title('Silhouette Score for Different Values of K (KMeans)')
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Silhouette Score')
plt.xticks(K_range)
plt.grid(True)
plt.show()
best_k_index = np.nanargmax(sil_scores)
best_K = K_range[best_k_index]
print(f"Best K (Number of Clusters) based on Silhouette Score: {best_K}")
```

This code performs K-Means clustering with different values of K (number of clusters) and evaluates clustering quality using the Silhouette Score. It iterates over a range of K values (from

2 to 41), fitting a KMeans model to the standardized feature data (X_scaled). After training, it computes the Silhouette Score, which measures how well-separated the clusters are (higher values indicate better clustering). If a ValueError occurs during clustering, it catches the exception and stores NaN for that K. The Silhouette Scores for each K are plotted to visualize clustering performance across different values of K. Finally, the best K is determined by finding the K value with the highest Silhouette Score, which is printed as the optimal number of clusters for K-Means.

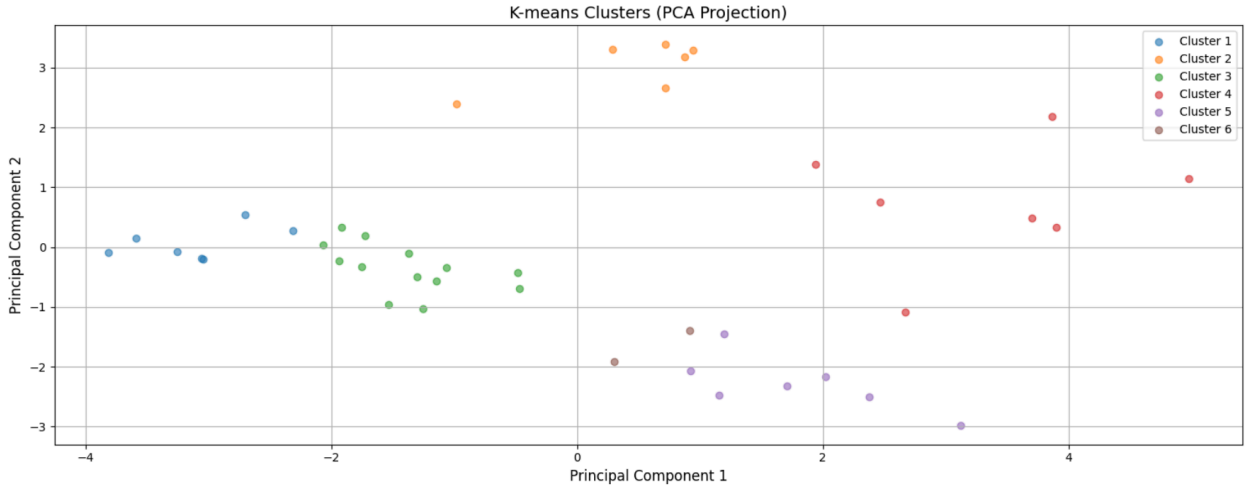


Best K (Number of Clusters) based on Silhouette Score: 6

```
kmeans = KMeans(n_clusters=best_K)
kmeans.fit(X_scaled)
df_identity1['cluster'] = kmeans.labels_
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)
plt.figure(figsize=(15, 6))
for cluster in range(best_K):
```

```
      plt.scatter(X_pca[df_identity1['cluster']     ==     cluster,     0],
X_pca[df_identity1['cluster'] == cluster, 1],
                  label=f'Cluster {cluster + 1}', alpha=0.6)
plt.title('K-means Clusters (PCA Projection)', fontsize=14)
plt.xlabel('Principal Component 1', fontsize=12)
plt.ylabel('Principal Component 2', fontsize=12)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

This code applies K-Means clustering using the optimal number of clusters (best_K), which was previously determined based on the Silhouette Score. It first fits the K-Means model on the standardized feature set (X_scaled) and assigns each data point a cluster label, storing it in the 'cluster' column of df_identity1. To visualize the clusters, Principal Component Analysis (PCA) reduces the feature dimensions to two principal components, enabling a 2D scatter plot of the clustered data. Each cluster is plotted with a distinct color, labeled accordingly, and displayed with proper formatting, including grid lines, a title, and axis labels. This visualization helps interpret the clustering structure and how well-separated the data points are in a lower-dimensional space.

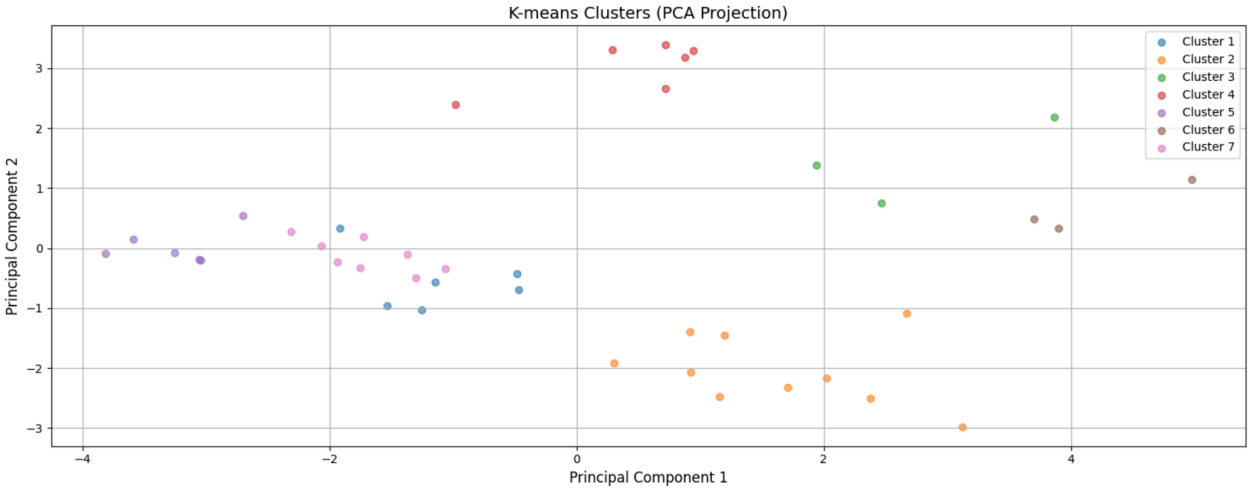


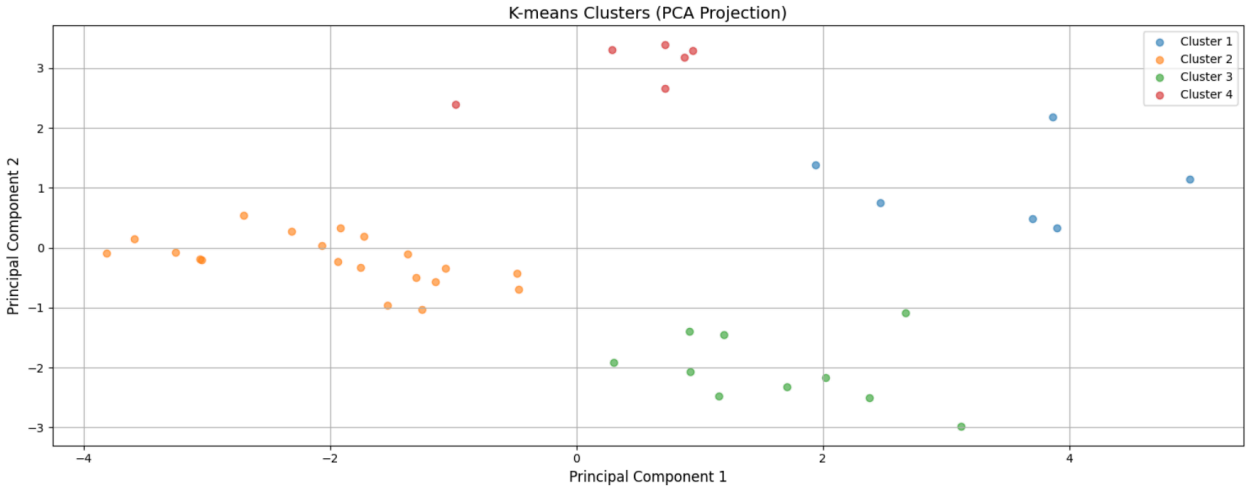
Silhouette Score for K-means clustering: 0.31831727687077

```
kmeans = KMeans(n_clusters=7)
kmeans.fit(X_scaled)
df_identity1['cluster'] = kmeans.labels_
pca = PCA(n_components=2)
```

```
X_pca = pca.fit_transform(X_scaled)
plt.figure(figsize=(15, 6))
for cluster in range(7):
      plt.scatter(X_pca[df_identity1['cluster']     ==     cluster,     0],
X_pca[df_identity1['cluster'] == cluster, 1],
                  label=f'Cluster {cluster + 1}', alpha=0.6)
plt.title('K-means Clusters (PCA Projection)', fontsize=14)
plt.xlabel('Principal Component 1', fontsize=12)
plt.ylabel('Principal Component 2', fontsize=12)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

This code applies K-Means clustering with 7 clusters on a standardized dataset (X_scaled) and assigns each data point a cluster label stored in the 'cluster' column of df_identity1. To visualize the clusters in a 2D space, it uses Principal Component Analysis (PCA) to reduce the feature dimensions to two principal components. A scatter plot is then generated, where each cluster is represented with a different color and labeled accordingly. The plot is formatted with a title, axis labels, a legend, and a grid to improve readability. This visualization helps analyze how well the K-Means algorithm has grouped the data points based on their similarity in a lower-dimensional space.





Gaussian Mixture

GMM is a probabilistic model that assumes the data is generated from a mixture of several normal distributions with unknown parameters. Each cluster in the data is modeled by a normal distribution, and the model estimates the mean, variance, and probability function of each Gaussian.

Expectation-Maximization (EM): E-step is to calculate the probability that each data point belongs to each Gaussian cluster. M-step is to update the parameters (mean, variance, weight) of each Gaussian based on the probabilities from the E-step. These two steps are repeated iteratively until convergence, where the parameters stabilize. At the end, the model assigns each data point a probability of belonging to each cluster.

Unlike hard clustering methods (like K-means), GMM allows each data point to belong to multiple clusters with different probabilities. It can model elliptical-shaped clusters, unlike K-means which assumes spherical clusters.

Why did we use Gaussian Mixture

- Speaker Modeling: GMMs are widely used in text-independent speaker recognition systems. Each speaker's voice is represented as a mixture of Gaussians in an acoustic feature space (e.g., Mel-Frequency Cepstral Coefficients - MFCCs).

- Flexibility in Data Distribution: Unlike k-means clustering, which assumes hard assignments, GMM assigns a probability to each speaker, allowing for more flexible classification.
- Capture of Variability: Speech data has variations in pitch, tone, and articulation. GMMs handle these by modeling the probability density function of features.
- Efficient Estimation: Using the Expectation-Maximization (EM) algorithm, GMMs efficiently learn the underlying distribution of voice features from training data.

Code of Gaussian Mixture

```
gmm = GaussianMixture(n_components=2)
gmm.fit(X)

labels = gmm.predict(X)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
plt.figure(figsize=(8, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=labels, cmap='viridis', marker='o')
plt.title('Gaussian Mixture Model Clustering')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.colorbar(label='Cluster Label')
plt.show()
print("Cluster Centers (means of the Gaussians):")
print(gmm.means_)
```

```
X_scaled = X
k_values = range(1, len(df_identity1))
silhouette_scores = []
for k in k_values:
      gmm = GaussianMixture(n_components=k, random_state=42)
      gmm.fit(X_scaled)
      labels = gmm.predict(X_scaled)
      if k > 1:
      score = silhouette_score(X_scaled, labels)
      silhouette_scores.append(score)
      else:
      silhouette_scores.append(np.nan)
k_values = k_values[1:]
```

```
silhouette_scores = [score for score in silhouette_scores if not
np.isnan(score)]
plt.figure(figsize=(12, 6))
plt.plot(k_values, silhouette_scores, marker='o', label='Silhouette Score')
plt.title('Silhouette Score for Different Number of Components')
plt.xlabel('Number of Components (K)')
plt.ylabel('Silhouette Score')
plt.legend()
plt.show()
best_k_silhouette = k_values[np.argmax(silhouette_scores)]
print(f"Best number of components (based on Silhouette Score):
{best_k_silhouette}")
```

```
gmm = GaussianMixture(n_components=best_k_silhouette)
gmm.fit(X)
labels = gmm.predict(X)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
plt.figure(figsize=(8, 6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=labels, cmap='viridis', marker='o')
plt.title('Gaussian Mixture Model Clustering')
plt.xlabel('PCA Component 1')
plt.ylabel('PCA Component 2')
plt.colorbar(label='Cluster Label')
plt.show()
print("Cluster Centers (means of the Gaussians):")
print(gmm.means_)
```

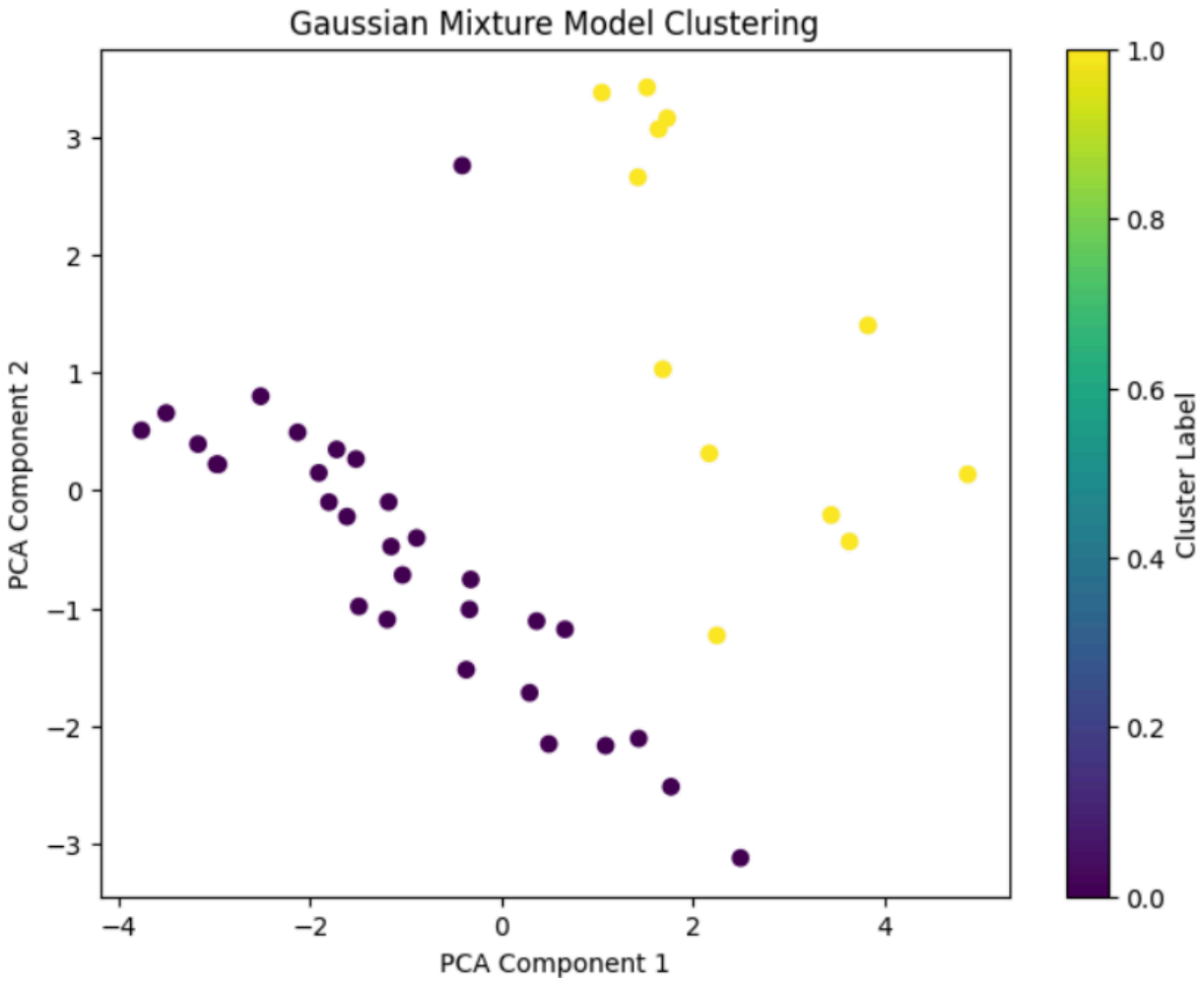
This code applies a Gaussian Mixture Model (GMM) to cluster data and evaluates the optimal number of components using the Silhouette Score. First, it initializes a GMM with two components (n_components=2), fits it to the dataset X, and assigns cluster labels to the data points. To visualize the clustering, the data is transformed using Principal Component Analysis (PCA) into two dimensions, and a scatter plot is generated where points are colored based on their assigned clusters. The mean values of the Gaussians, representing the cluster centers, are printed.

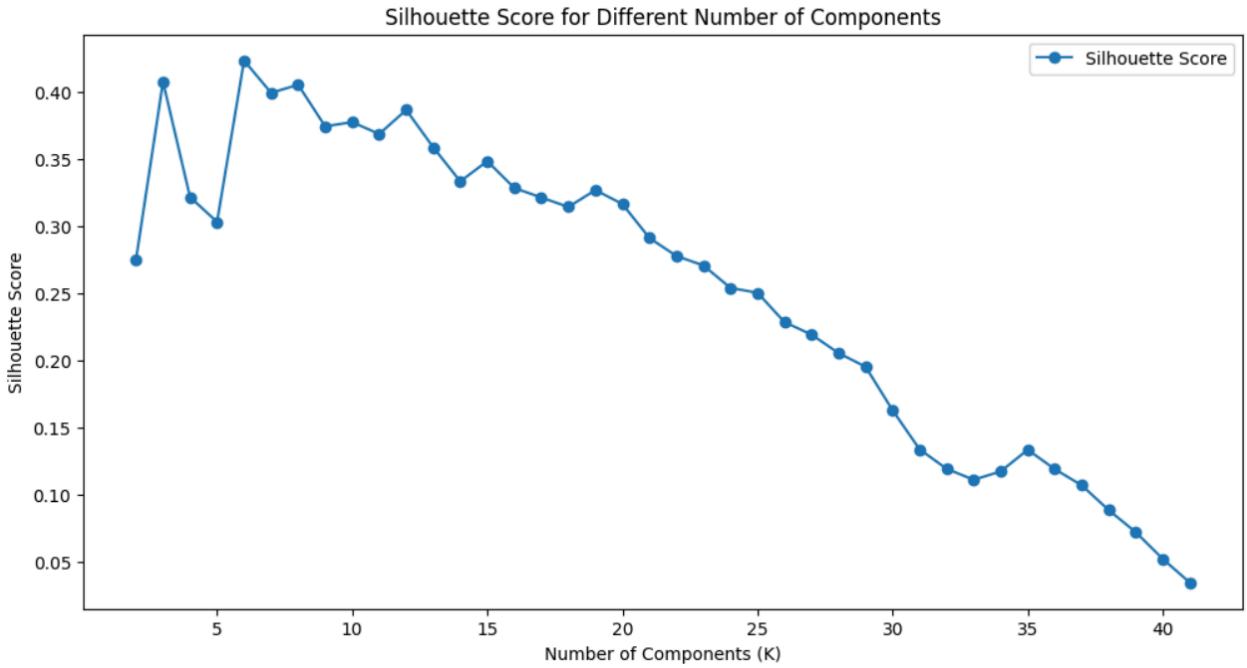
Next, the code determines the best number of Gaussian components by computing the Silhouette Score for different values of k (from 1 to the number of samples). It iterates over different values of k, fitting a GMM to the dataset and computing the Silhouette Score when k > 1, as the score is not meaningful for k = 1. The scores are plotted against different values of k,

allowing for a visual interpretation of the optimal number of components. The best k is selected as the one that maximizes the Silhouette Score, and the corresponding value is printed.

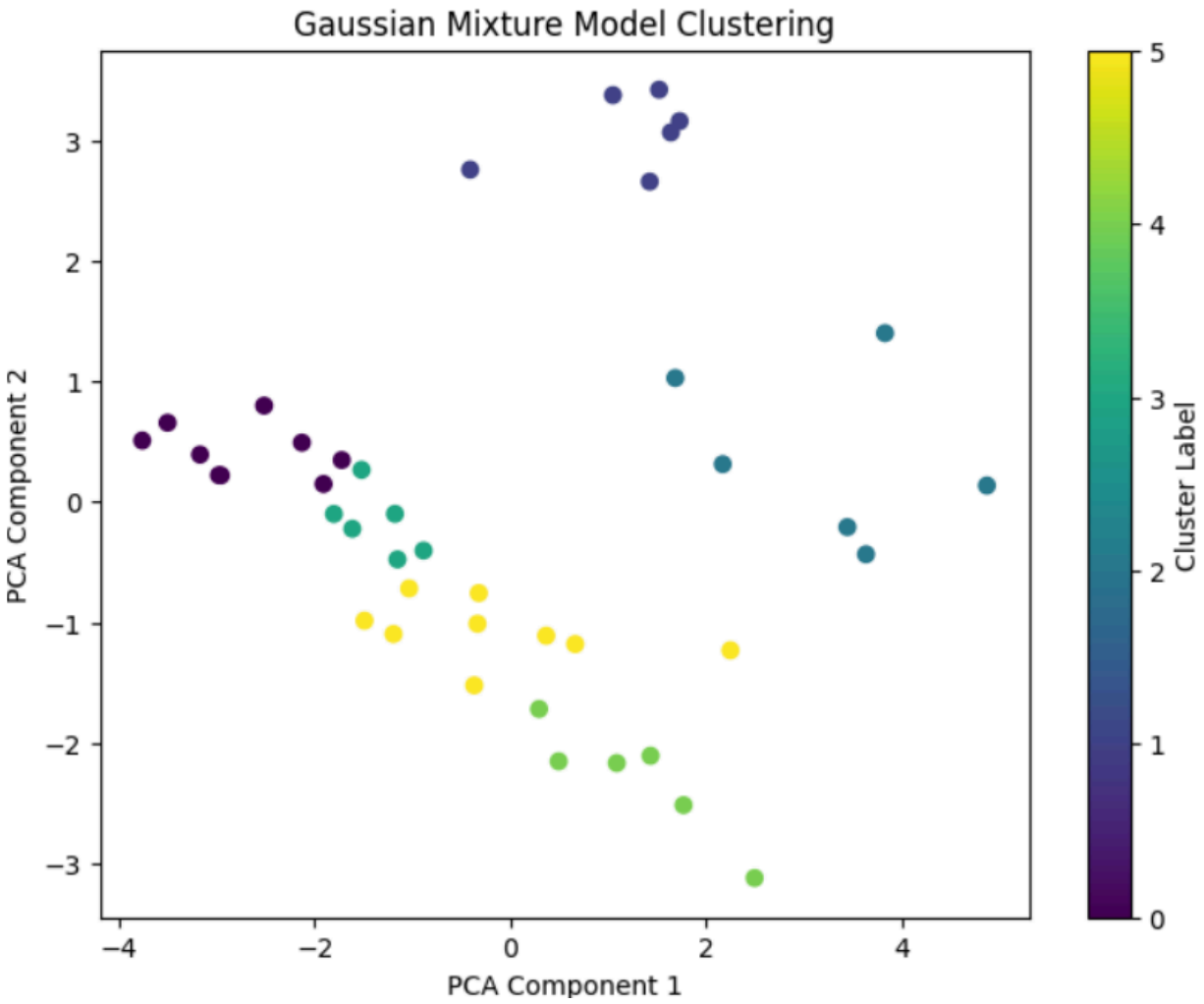
Finally, the model is re-trained using the optimal number of components (best_k_silhouette), and clustering is performed again. The PCA transformation is re-applied to visualize the new clusters, and another scatter plot is generated. The updated cluster centers (means of the Gaussians) are printed, providing insight into how the data is distributed across the learned clusters. This approach ensures a data-driven selection of the number of Gaussian components, leading to more effective clustering.

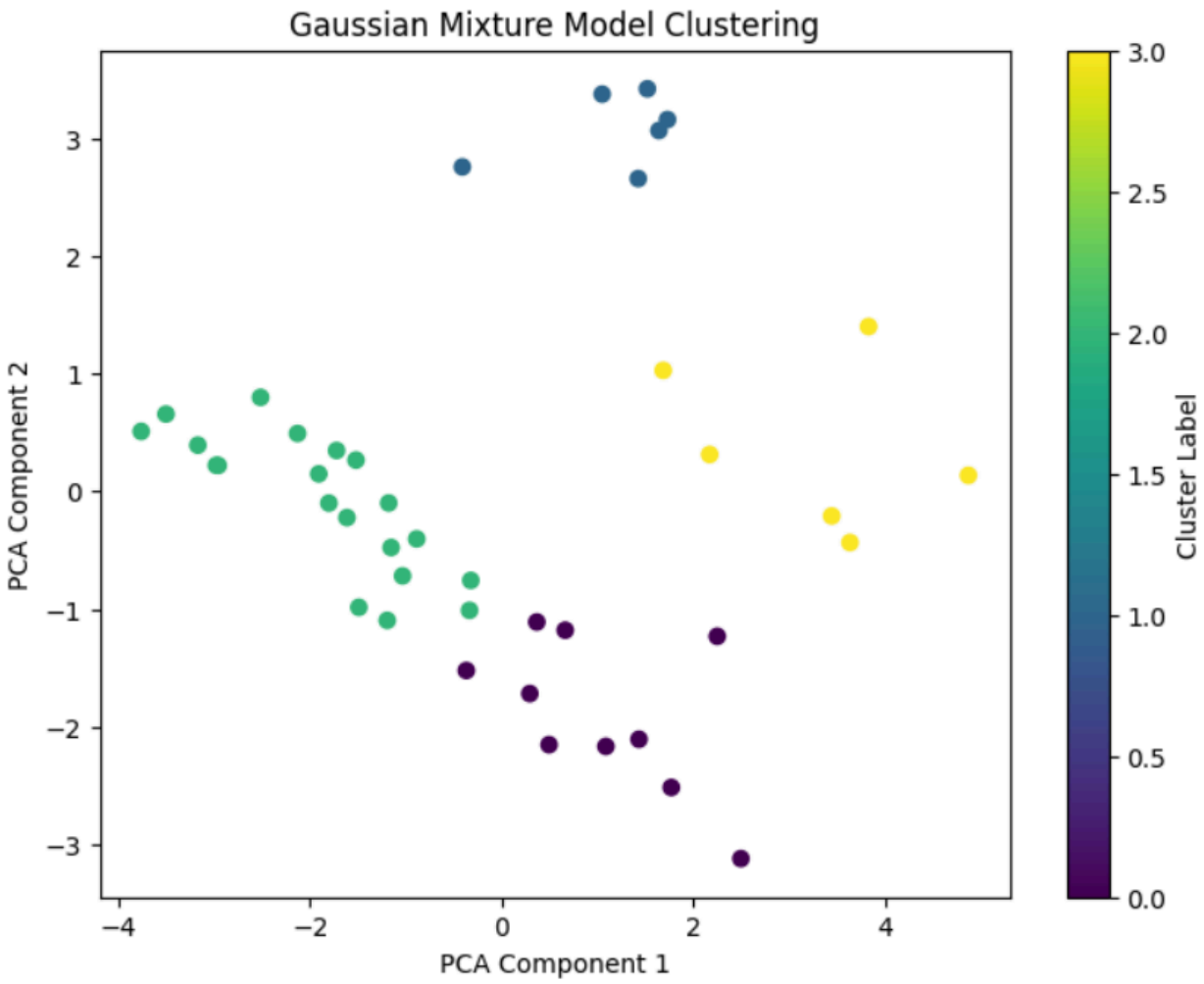
Result of Gaussian Mixture

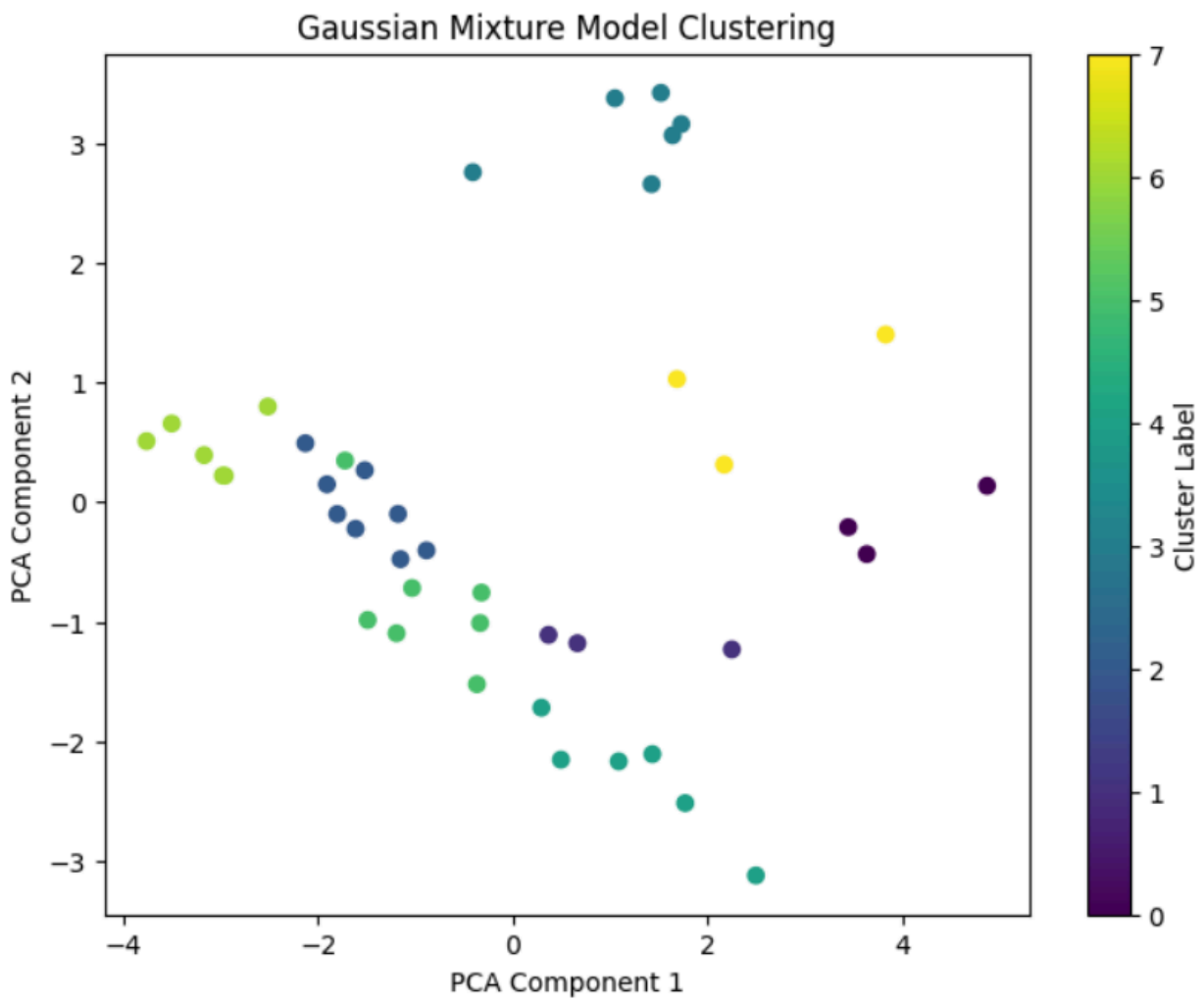




Best number of components (based on Silhouette Score): 6


Gaussian Mixture Model Clustering
PCA Component 2
PCA Component 1
Cluster Label
3
2
1
0
−1
−2
−3
−4
−2
0
2
4
5
4
3
2
1
0

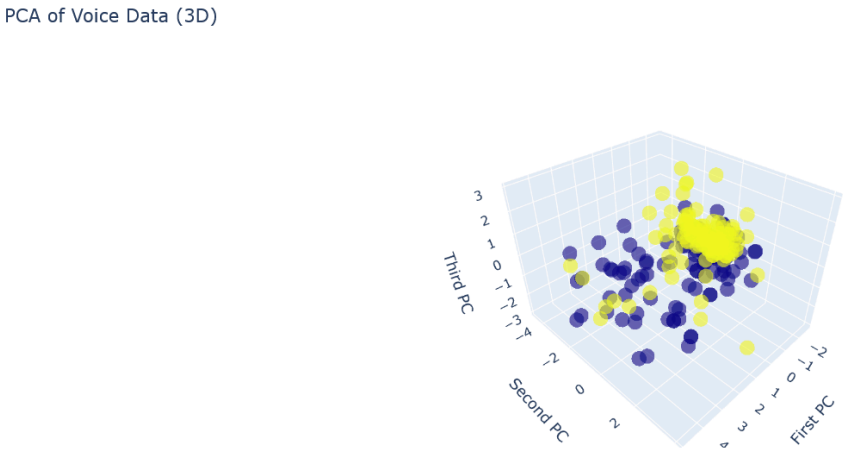

Gaussian Mixture Model Clustering
PCA Component 2
PCA Component 1
Cluster Label
3
2
1
0
−1
−2
−3
−4
−2
0
2
4
3.0
2.5
2.0
1.5
1.0
0.5
0.0



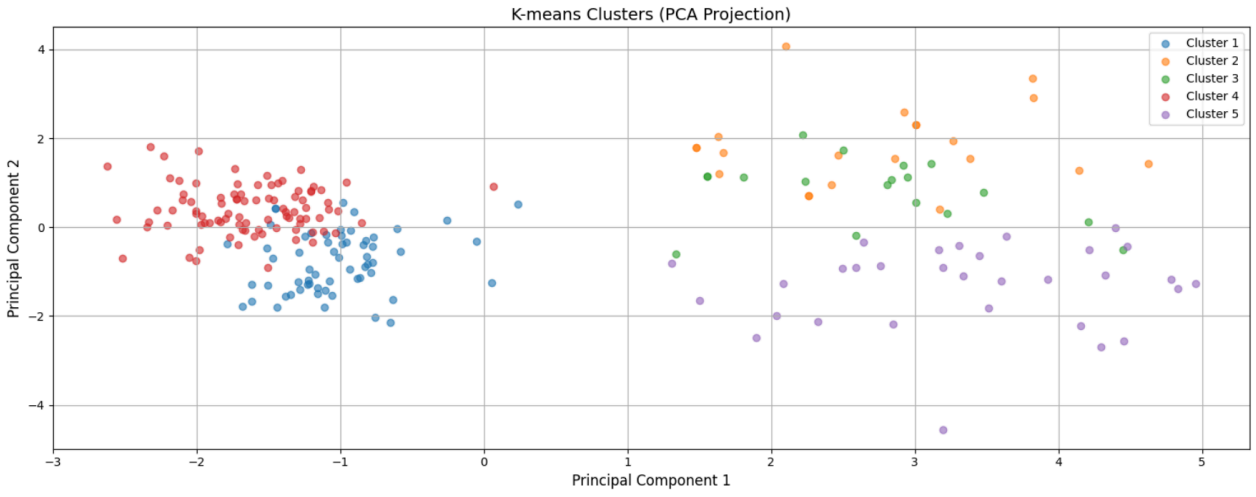
Gender Classification

We repeat the same code as we did for identity but we drop the Gender column instead of id.

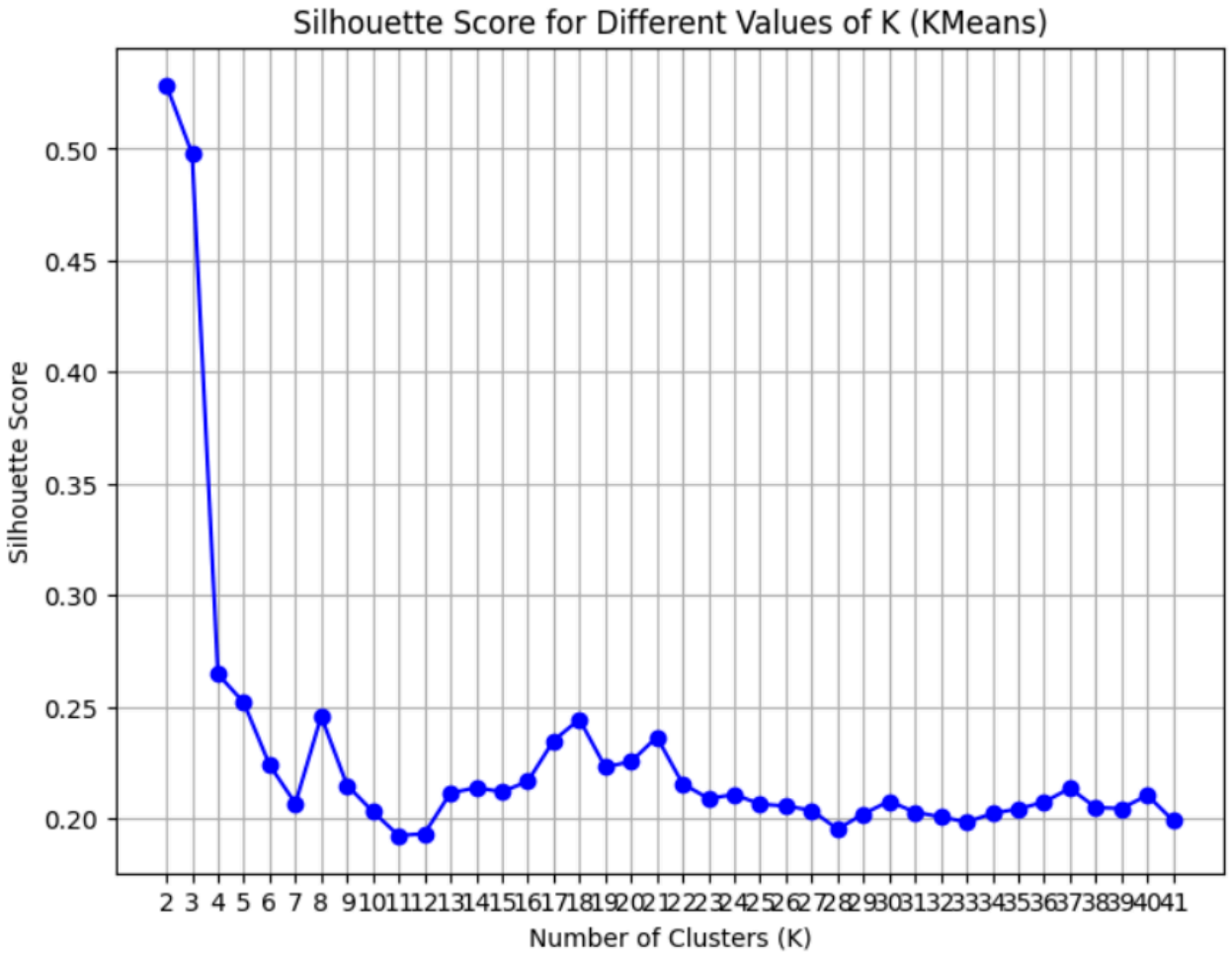
PCA



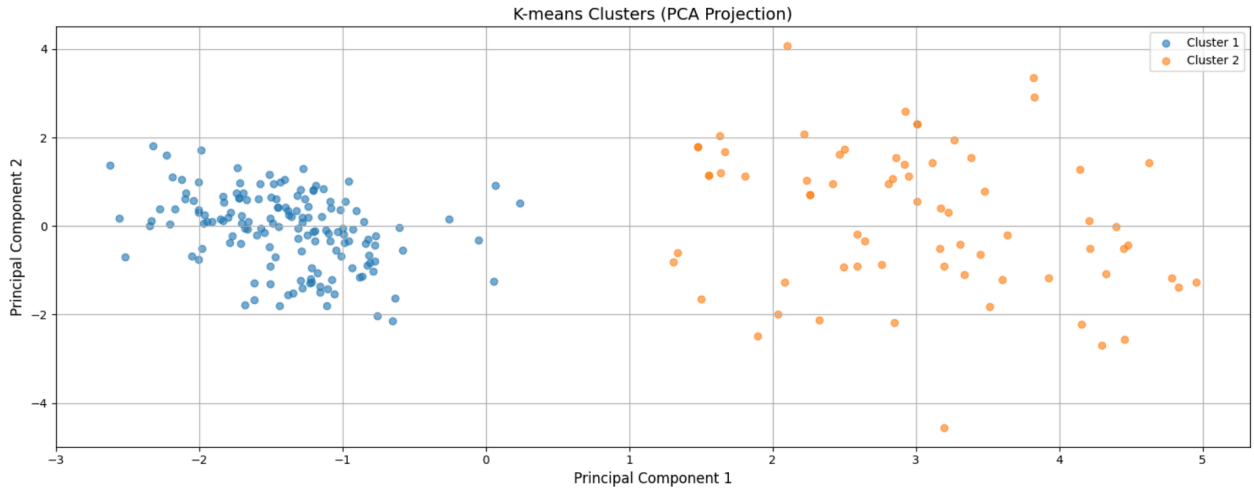
K-Means

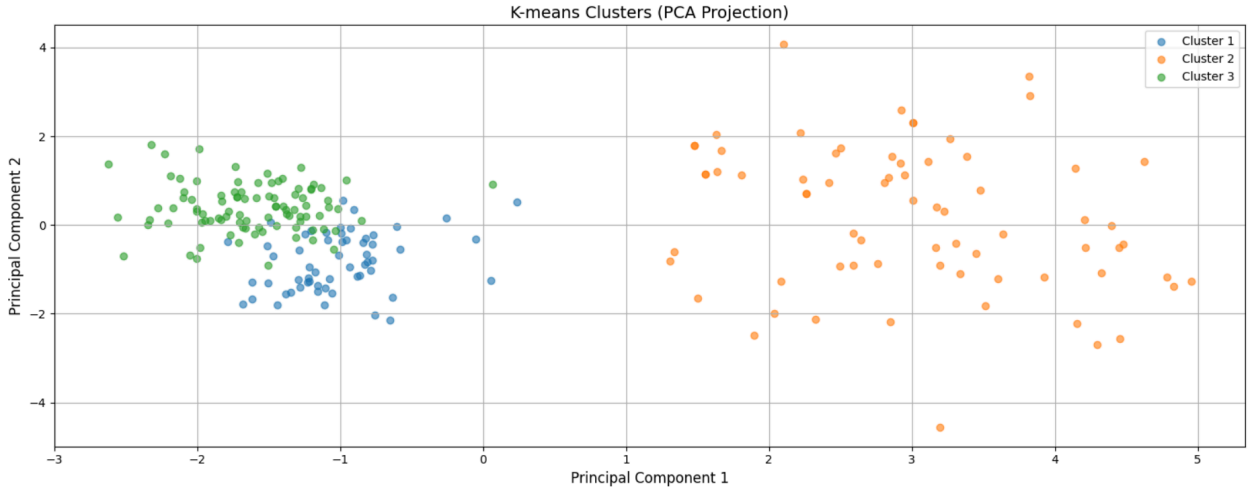


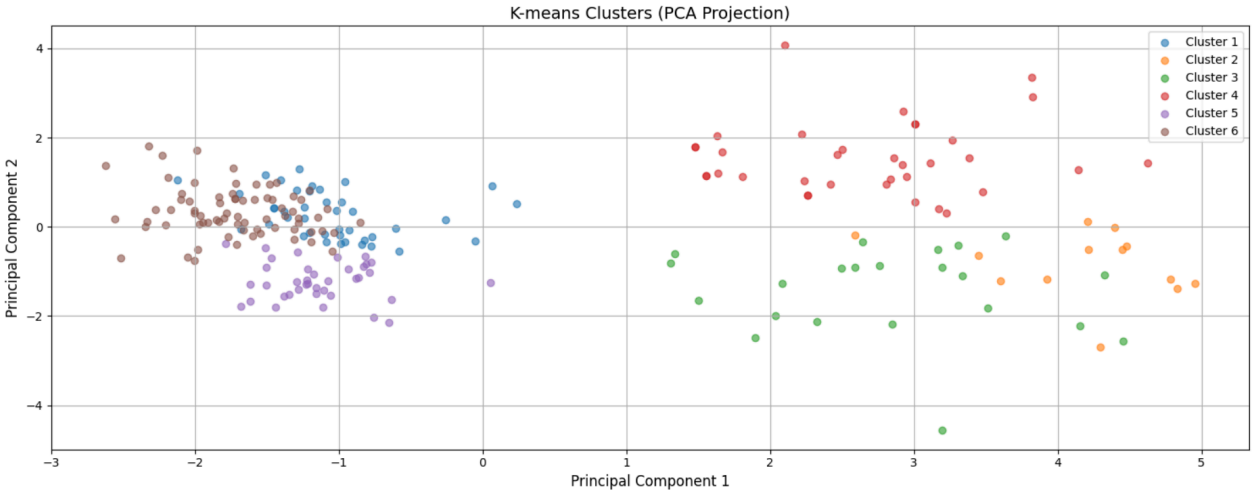
Silhouette Score for K-means clustering: 0.24878811990710686



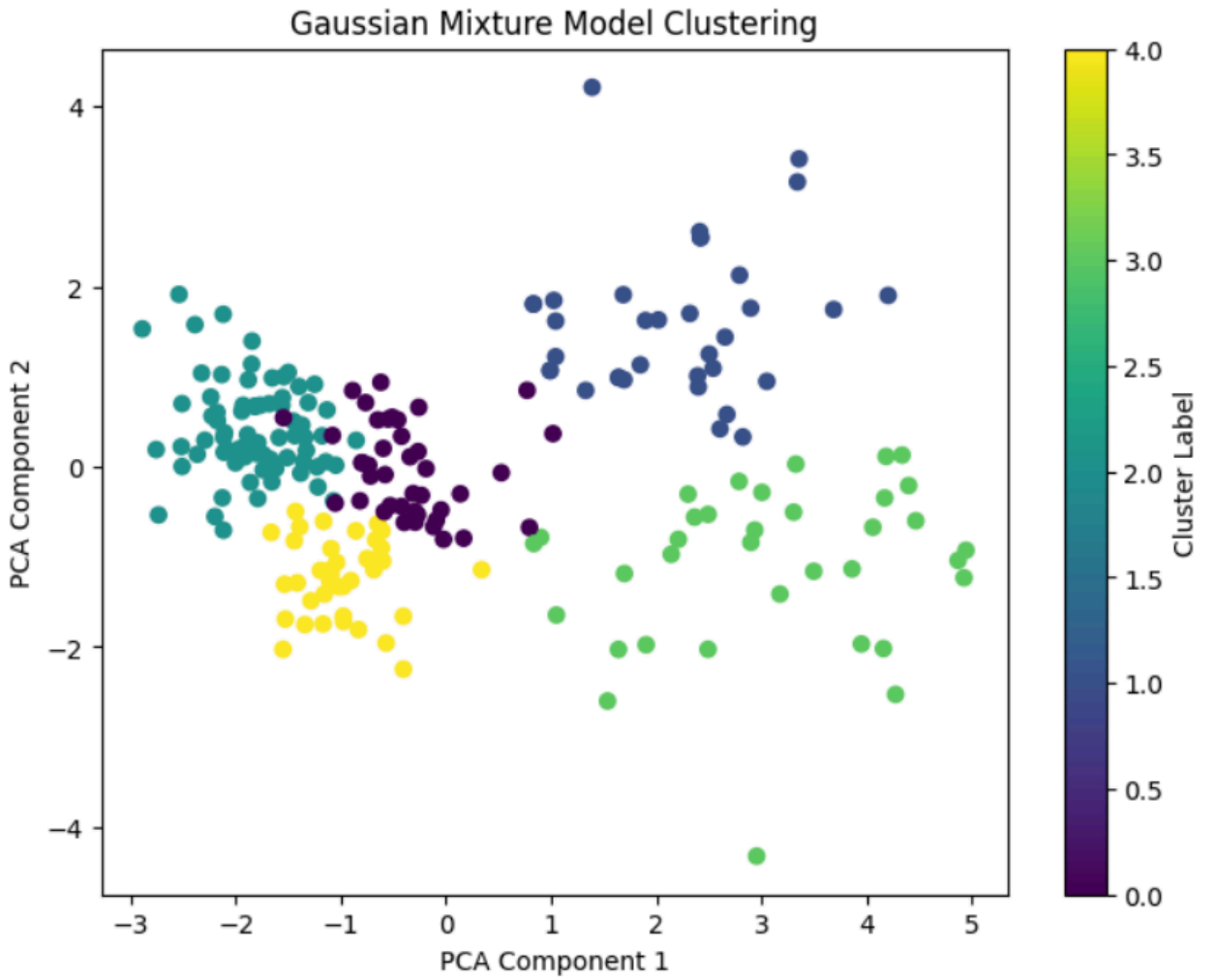
Best K (Number of Clusters) based on Silhouette Score: 2

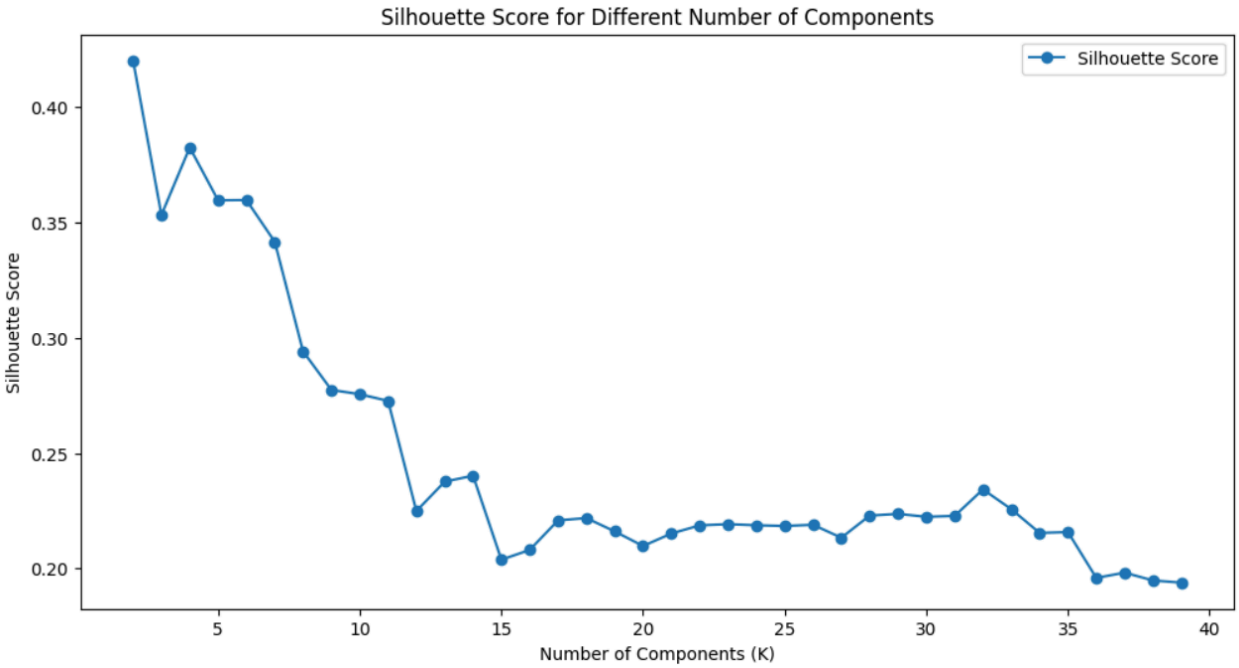



K-means Clusters (PCA Projection)
Cluster 1
Cluster 2
Cluster 3
Principal Component 2
Principal Component 1

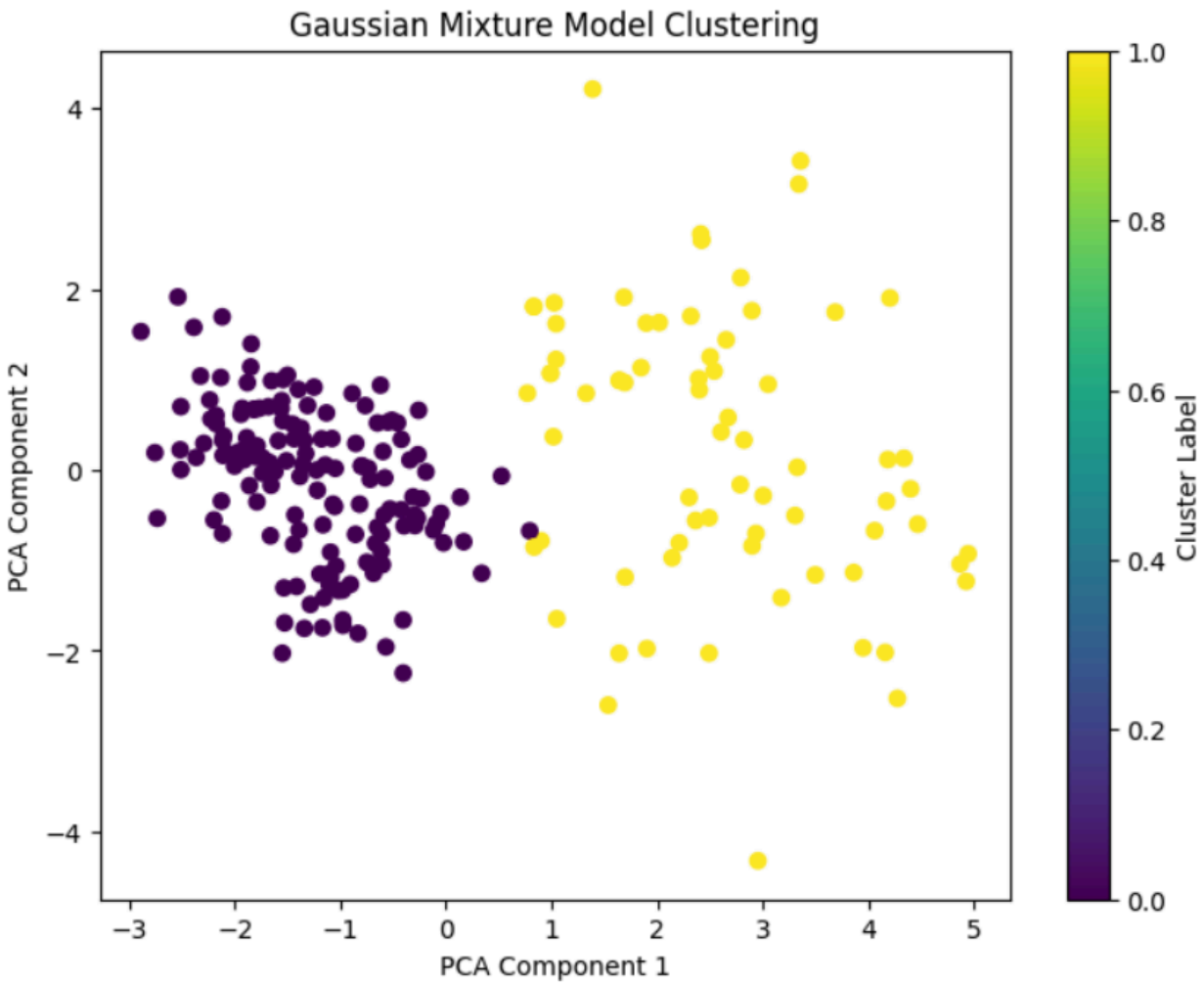

K-means Clusters (PCA Projection)
Cluster 1
Cluster 2
Cluster 3
Cluster 4
Cluster 5
Cluster 6
Principal Component 2
Principal Component 1

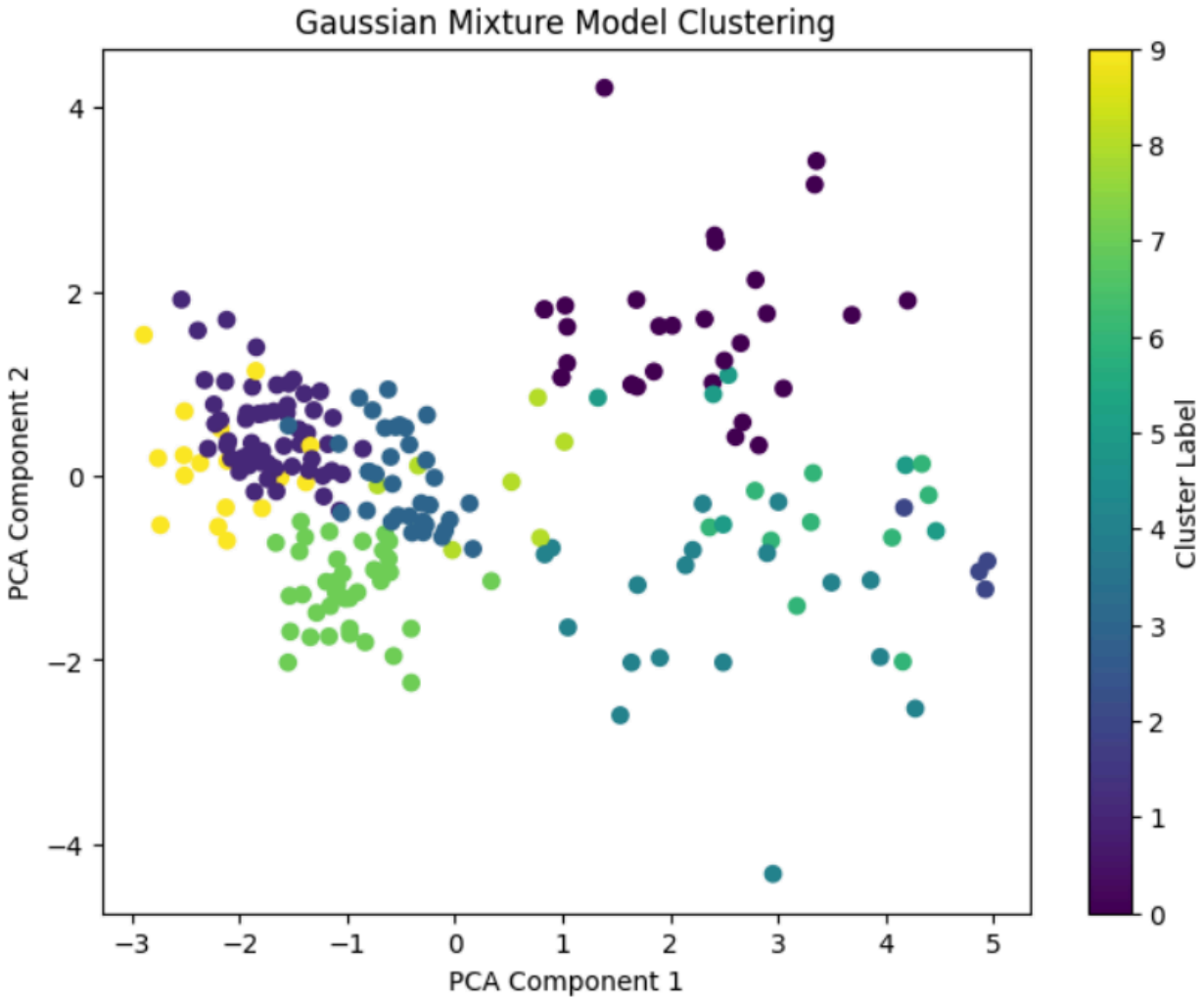
Gaussian Mixture

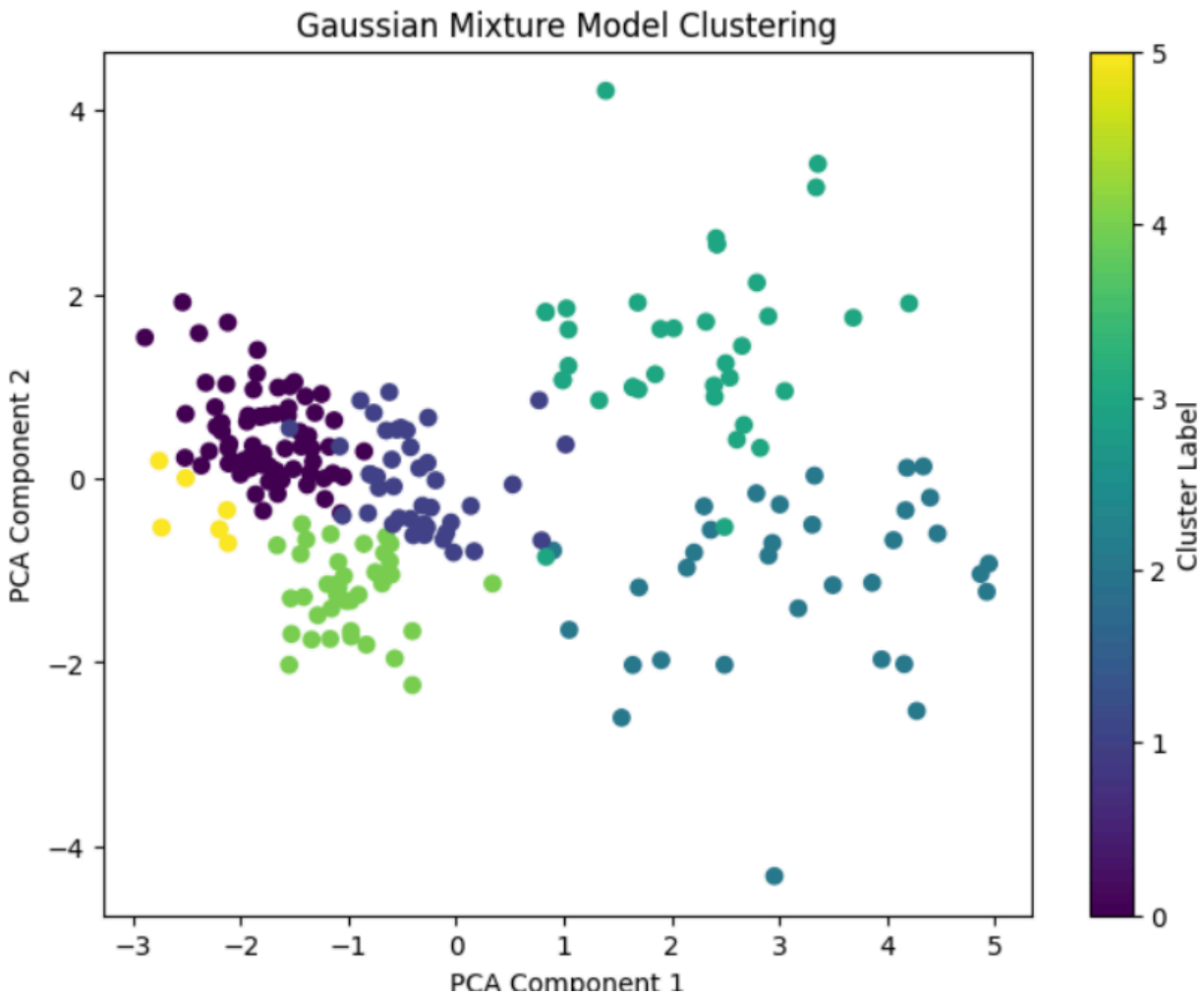




Best number of components (based on Silhouette Score): 2


Gaussian Mixture Model Clustering
PCA Component 2
PCA Component 1
Cluster Label
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4
5
1.0
0.8
0.6
0.4
0.2
0.0


Gaussian Mixture Model Clustering
PCA Component 2
PCA Component 1
Cluster Label
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4
5
9
8
7
6
5
4
3
2
1
0


Gaussian Mixture Model Clustering
PCA Component 2
PCA Component 1
Cluster Label
4
2
0
−2
−4
−3
−2
−1
0
1
2
3
4
5
0
1
2
3
4
5

Resources

- Closed-set identification
- Top 10 use cases for voice biometric systems
- What is Voice Recognition?
- Advantages and disadvantages of voice authentication
- Machine Learning based Voice Authentication and Identification
- Clustering and Classification on Gender Classification Dataset
- Top 4 Speech Recognition Challenges & Solutions In 2024
- What challenges do we meet in voice recognition?
- Voice Recognition Technology
- Top 4 Speech Recognition Challenges & Solutions
- MFCC
- DTFT
- FFT
- Mel Spectrogram
- Spectral Centroid
- Chroma features
- Spectral contrast
- Zero-crossing rate
- LPC
- What is Similarity Learning? Definition, Use Cases & Methods
- Enabling Open-Set Person Re-Identification for Real-World Scenarios
- Contrastive Loss
- Triplet loss
- Loss Functions in Machine Learning Explained
- Preprocessing the Audio Dataset
- Methods for sound noise reduction
- Speech and Language Processing Book by Daniel Jurafsky and James H. Martin
- Automatic Speech Recognition: A Deep Learning Approach (Signals and Communication Technology) 2015th Edition by Dong Yu (Author), Li Deng (Author)

- Handbook of Biometrics 2008th Edition by Anil K. Jain (Editor), Patrick Flynn (Editor), Arun A. Ross (Editor)
- "Application of speaker age and gender classification in speech recognition." IEEE Transactions on Audio, Speech, and Language Processing.
- "Speaker recognition by machines and humans: A tutorial review." IEEE Signal Processing Magazine.
- "Emotion recognition in human-computer interaction." IEEE Signal Processing Magazine.
- Self-Supervised Learning for Speech Recognition. Advances in Neural Information Processing Systems (NeurIPS).

Splitted Dataset

Dataset is available in this google drive.