Voice Authentication Project Report

Authors:
Amin Ariafar - Student ID : 610300017
Negar Sourati - Student ID : 610300070
Reza Torabi - Student ID : 610300032
Pooria Heydarian - Student ID : 610399206

Winter 2024-2025

1 Load Dataset

First of all, we need to list the main libraries we used in different sections of the project :
Numpy, Pandas -> data proccessing and data handling
Matplot, Seaborn -> plotting
Scikit-Learn, Tensorflow, xgboost -> modeling and performance measures
Regex -> distinguishing text patterns
Librosa -> preproccessing and extracting features from audio data

In this section, we use a regex pattern (label_pattern) to filter audio files with valid filenames containing a student ID and gender. For all matching audio files, we perform preprocessing and feature extraction. The extracted gender and student ID values are expanded to match the number of frames in the feature set. We just pick 500 audios out of 800 audios from the dataset because it takes so much time to preprocess and extract features from all of them. Finally, we construct a dataframe and save the processed data to a CSV file (matrix1.csv) to avoid redundant processing in future runs.

```
label_pattern = \
re.compile(r".+([0-9]{9}).(male|female|MALE|FEMALE|Male|Female).+")

filename = "matrix1.csv"
i = 0

for audio_file in audio_files[:]:

    file_path = os.path.join(audio_folder, audio_file)

    y, sr = librosa.load(file_path, sr=48000)
    matches = list(label_pattern.finditer(audio_file))

    if matches:
        match = matches[0]

        gender = match.group(2).lower()
        student_id = match.group(1)
        gender = np.array(gender)
        student_id = np.array(student_id)

        preprocessed = preprocessing(y)
```

```
audio_features = feature_extraction(preprocessed)

gender_rep  = np.repeat(gender, repeats=1954, axis= 0)\
.reshape(-1, 1)

student_id_rep  = np.repeat(student_id, repeats=1954, axis= 0)\
.reshape(-1,1)

Feature_Matrix = np.concatenate((audio_features, gender_rep, \
student_id_rep), axis= 1)
```

2 Preprocessing

Spectral subtraction is a widely used technique for speech and audio denoising, particularly effective in removing stationary background noise like fan noise or hums. The core concept involves estimating the noise spectrum and subtracting it from the noisy signal in the frequency domain. This process begins by transforming the noisy signal into the frequency domain using the Short-Time Fourier Transform (STFT). Since noise is typically estimated from silent or low-energy frames, we identify the minimum energy of the signal to approximate the noise. After this estimation, we subtract the noise spectrum from the original noisy signal. Any resulting negative values are replaced with zeros using np.maximum, ensuring no negative amplitudes remain in the spectrogram. Finally, we reconstruct both the noise and the cleaned signal using the inverse STFT.

```
def denoising_spectral_subtraction(y):

  S_full = np.abs(librosa.stft(y, n_fft=2048, hop_length=512))
  phase = np.angle(librosa.stft(y, n_fft=2048, hop_length=512))

  noise_est = np.min(S_full, axis=1, keepdims=True)

  S_clean = np.maximum(S_full - noise_est, 0)

  noise_complex = noise_est * np.exp(1j * phase)
  noise = librosa.istft(noise_complex, hop_length=512)

  S_clean_complex = S_clean * np.exp(1j * phase)
  y_clean = librosa.istft(S_clean_complex, hop_length=512)
```

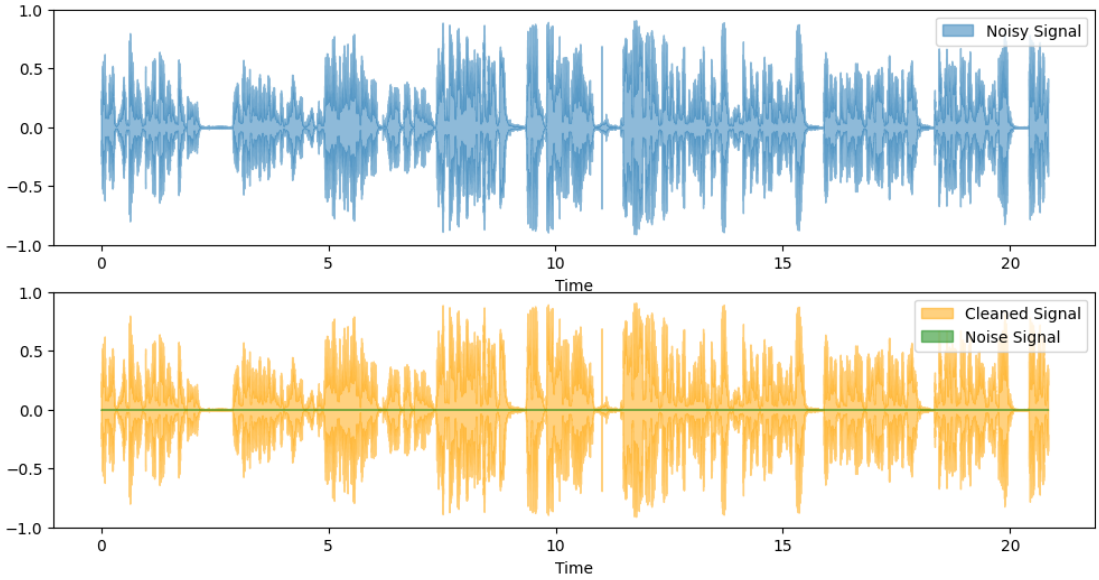

FIGURE 1 – Visualization of Spectral Subtraction for a Sample Audio

Next, we extract a central portion of the noise segment that carries the most relevant information. To ensure consistency in processing, we apply symmetric padding if the audio length is shorter than a specified target. If the audio is longer, we extract the middle section directly. Padding shorter signals guarantees uniform output size for further processing, regardless of input variations.

```
def extract_middle_audio_with_padding(audio, target_length):
    audio_length = len(audio)

    if audio_length < target_length:
        pad_width = (target_length - audio_length) // 2
        remainder = (target_length - audio_length) % 2

        padded_audio = np.pad(
            audio,
            pad_width=(pad_width, pad_width + remainder),
            mode='constant',
            constant_values=0
        )
        return padded_audio
    else:
        start_idx = (audio_length - target_length) // 2
        end_idx = start_idx + target_length
```

```
    middle_audio = audio[start_idx:end_idx]
    return middle_audio
```

3 Feature Extraction

For extracting features of our audio files, we have extracted the following features :
1 - MFCC features
2 - Spectral Bandwidth
3 - Mel Spectrogram
4 - Spectral Centroid
5 - Spectral Contrast
6 - Zero Crossing Rate

First we extract each of the features individually and then we concatenate them horizontally using np.concatenate to get the final feature matrix. Below every feature extraction, the code for its' implementation is also provided. For the sake of visualization, we attached the plot of the output of each feature (implemented on one audio file) to have a better understanding of them.

3.1 MFCC features

MFCC extraction involves converting a time-domain audio signal into a set of compact features that represent the frequency characteristics of the signal based on how humans perceive sound.
Like all the features extracted from the audio files, we used "librosa" library for getting MFCCs. The n-mfcc indicates how many of the MFCC coefficients we want to extract (13 is the mostly used for usual applications). SR parameter refers to the sampling rate at which we have gathered the audio frames. Like most of the features, first we need to transform the wave from amplitude space to frequency space. For this purpose, we need to perform Fourier transformation on our audio series. The n-fft parameter refers to the number of samples (frames) used in each Short Term Fourier Transform (STFT) and hop-length refers to the shift between every two transformation.
The Fourier Transform assumes the input signal is periodic. However, most real-world signals are not periodic over the window of analysis. Windowing ensures that the signal behaves more like a periodic function within the analysis window, improving the accuracy of the Fourier Transform.
Windowing is a process of applying a mathematical function (called a window function) to a segment of a signal. This function modifies the amplitude of the

segment to reduce edge effects, improve frequency analysis, and isolate specific parts of the signal.

Hann Window, one of the most widely used windowing types and the method we used in our project, is a window function that applies a smooth taper to the signal at the edges, reducing spectral leakage significantly. Formula (N is the total number of points in the window, n is the sample index) :
$w[n] = 0.5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right), \quad 0 \leq n < N$

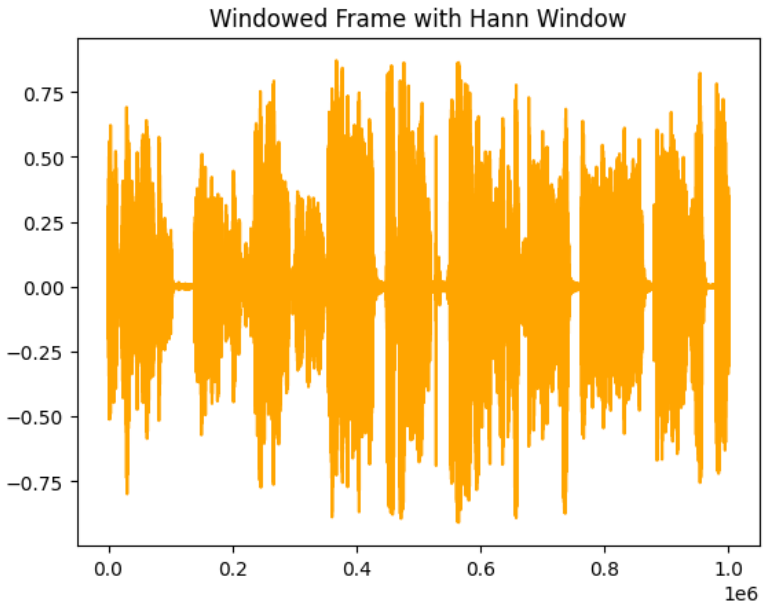

FIGURE 2 – Visualization of Hann Windowing for a Sample Audio

Finally, we normalize (by mean normalization) the output to avoid getting biased to a specific coefficient.

```
mfccs = librosa.feature.mfcc(
    y=signal_normalized,
    sr=sample_rate,
    n_mfcc=13,
    n_fft=1024,
    hop_length=512,
    window='hann'
)
mfccs_mean_normalized = mfccs - np.mean(mfccs)
```

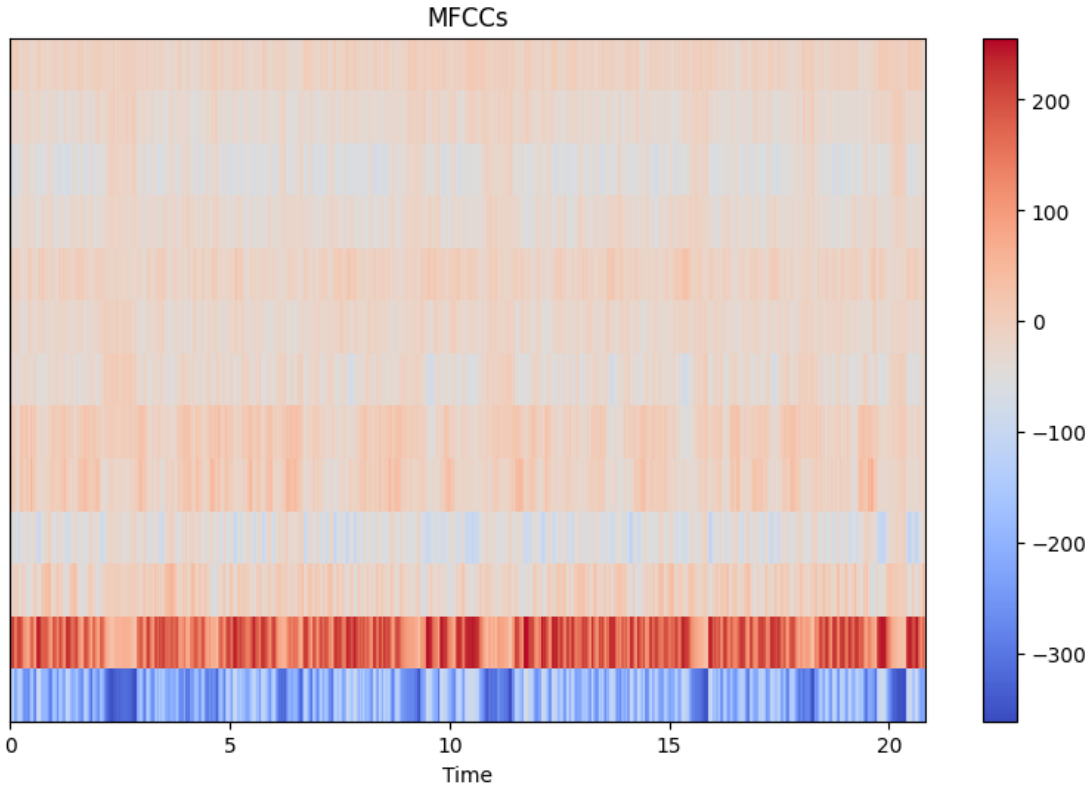

FIGURE 3 – MFCC plot

3.2 Spectral Bandwidth

Spectral Bandwidth is a feature in audio signal processing that measures the spread of frequencies in a sound signal around its spectral centroid. It essentially quantifies how much energy is spread across different frequencies and is useful for analyzing timbre, texture, and sharpness of a sound. Like MFCC, we need to perform Fourier transformation (and also windowing) for this features.

```
spectral_bandwidth = librosa.feature.spectral_bandwidth(
        y=signal_normalized,
        sr=sample_rate,
        n_fft=1024,
        hop_length=512,
        window='hann'
)
```

3.3 Spectral Centroid

The spectral centroid is a measure that identifies the center of gravity of the frequency spectrum. The S parameter if provided, should be a precomputed spectrogram (power or magnitude). The spectral centroid is usually computed from a linear-frequency spectrogram (from STFT). However, if you want to compute the spectral centroid in the Mel scale, you can pass a Mel spectrogram (S), which is already frequency-warped to match human hearing perception. We used mel spectrogram as we already computed it.

```
Spectral_Centroid = librosa.feature.spectral_centroid(
        S=MEL,
        sr=sample_rate,
        n_fft=1024,
        hop_length=512,
        window='hann'
)
```

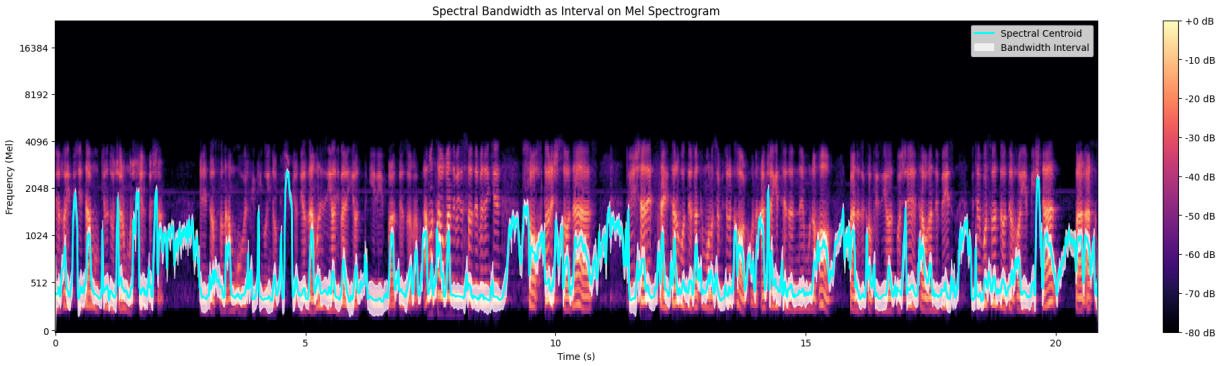

FIGURE 4 – Spectral Bandwidth and Spectral Centroid on Mel Spectrogram

3.4 Mel Spectrogram

As explained in the previous report, Mel spectrogram represents the time-frequency of an audio signal, in which the frequency axis is transformed into the Mel scale and amplitude. N-mels parameter is the number of Mel filter banks. More filters provide better frequency resolution. Fmax is the maximum frequency to analyze. The output shape of librosa.feature.melspectrogram() is a 2D NumPy array with the shape (n-mels, T), where :
n-mels = Number of Mel frequency bands (default is 128).
T = Number of time frames, which depends on the hop length and the length of the input signal.

```
MEL = librosa.feature.melspectrogram(
    y=signal_normalized,
    sr=sample_rate,
    n_fft=1024,
    n_mels=128,
    fmax= sample_rate//2,
    hop_length=512,
    window='hann'
  )
```

For visualization usually the Log Mel Spectrogram is used which is computed as :

```
log_mel_spectrogram = librosa.power_to_db(mel)
```

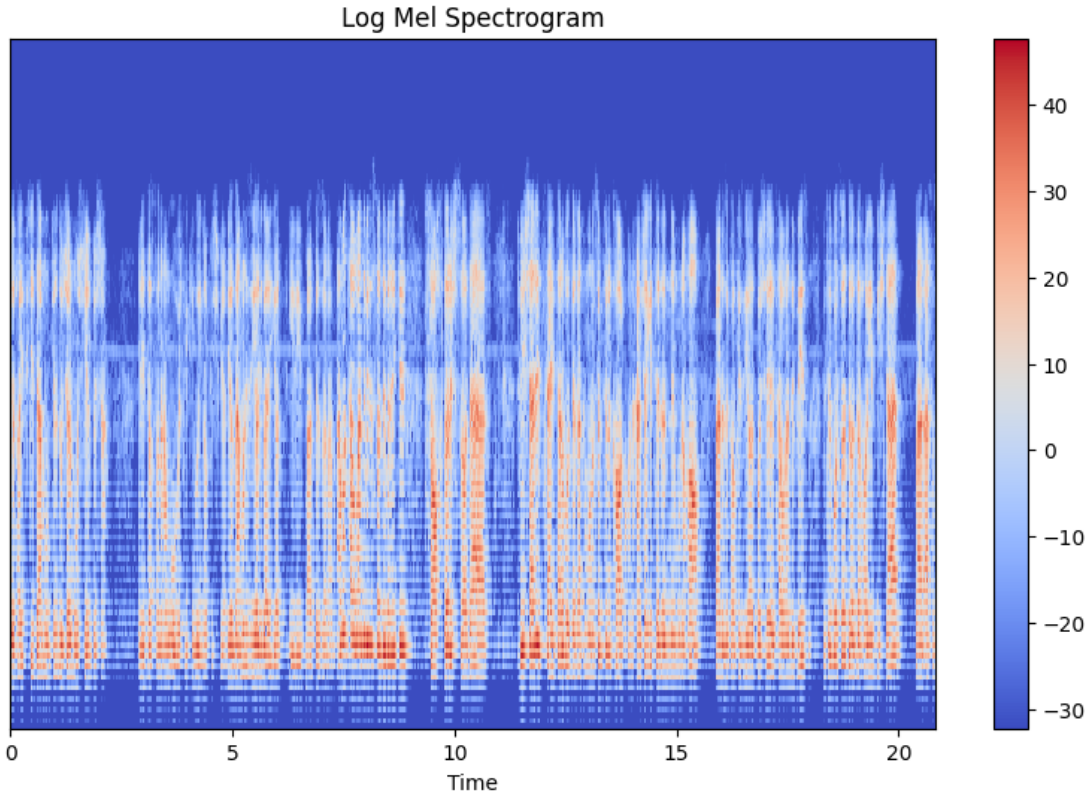

FIGURE 5 – Log Mel Spectrogram

3.5 Spectral Contrast

Spectral contrast measures the difference in amplitude between peaks and valleys within the frequency spectrum, capturing the richness of the sound. The parameters are similar to other features explained above.

```
Spectral_Contrast= librosa.feature.spectral_contrast(
    y=signal_normalized,
    sr=sample_rate,
    n_fft=1024,
    hop_length=512,
    window='hann'
)
```

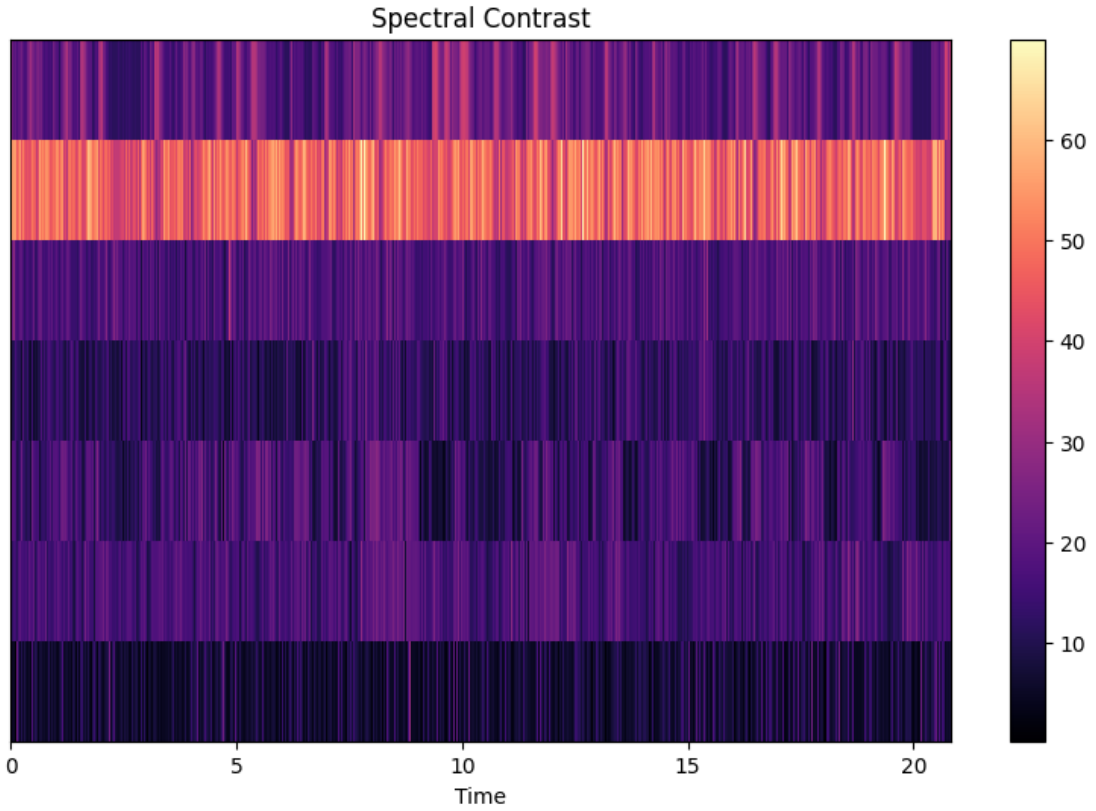

FIGURE 6 – Spectral Contrast plot

3.6 Zero Crossing Rate

The zero-crossing rate measures how frequently the signal changes its sign (crosses the zero amplitude axis) over a given time frame. The frame-length parameter is the number of samples per analysis frame (determines time resolution). Also, the hop-length parameter refers to the number of samples between successive frames (determines overlap).

```
Zcr= librosa.feature.zero_crossing_rate(
    y=signal_normalized,
    frame_length= 2048,
    hop_length=512
)
```

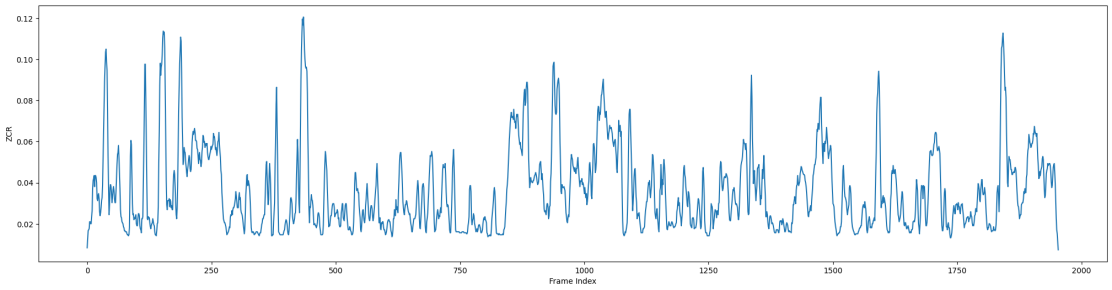

FIGURE 7 – Zero Crossing Rate plot

For the sake of visualization we reduced the feature space to two dimensions using PCA. The color of the data points are based on the two genders.

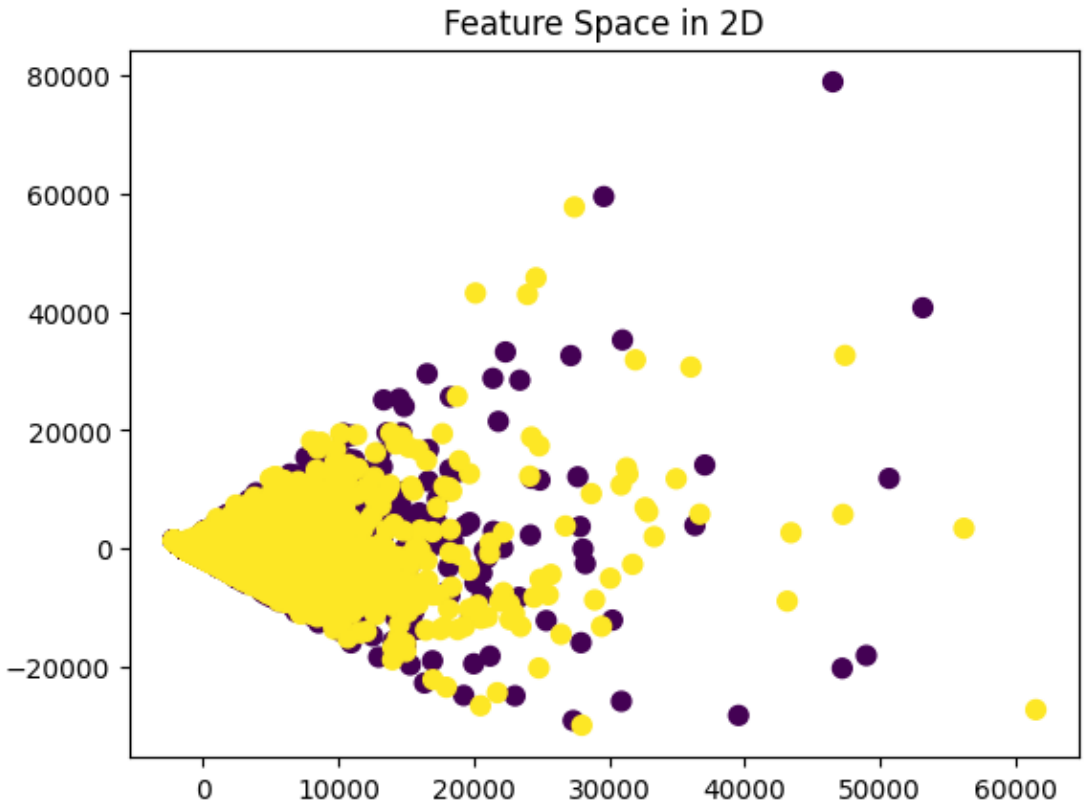

FIGURE 8 – Feature Space in 2 dimensions

4 Gender Classification

After concatenating the extracted features, we ended up with 151 features of each frame of audio files. With some analysis, we realized that we can capture 99.92% of the data's variance by keeping the first 70 components given by PCA method. So we reduced the features' dimension to 70 in the tasks of classification and clustering.

For gender classification, for having a balanced dataset, we take 10000 random samples from the records of each gender (note that our records consist of the frames of audio files). Before performing PCA, we first split the data into train and test sets (25% train, 75% test) scale our dataset using Min-Max scaling to avoid getting biased to a specific feature.

We used 4 models for gender classification :

1) Logistic Regression

2) KNN

3) XGBoost
4) MLP (MultiLayer Perceptron)
The choice of models was based on having different models with different levels of complexity to end up with the best one. From a very simple model (Logistic Regression) to a relatively complex model (MLP).
For each of these models, we reported the confusion matrix and accuracy of the classifier, recall, precision and F1 score in each of the classes and also the ROC curve and the AUC.
The code used for each model is provided. It should also be noted that "male" class is encoded as 1 and "female" class is encoded as 0.

4.1 Logistic Regression

The model ended up with a test accuracy score of 75.12% after 2000 iterations.

```
def ROC_curve(y_test, y_pred, n_classes = 2):

    fpr, tpr, _ = roc_curve(y_test, y_pred)
    roc_auc = auc(fpr, tpr)


    plt.plot(fpr, tpr, label=f'ROC curve (area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], 'k--') # Diagonal line
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic')
    plt.legend()
    plt.show()

def logistic_regression(X_train_scaled, X_test_scaled, y_train, y_test):
    logisticRegr = LogisticRegression(max_iter = 2000)

    logisticRegr.fit(X_train_scaled, y_train)

    logisticRegr.predict(X_test_scaled[0].reshape(1,-1))

    y_pred = logisticRegr.predict(X_test_scaled)

    print('Logistic Regression model accuracy score: {0:0.4f}'.\
    format(accuracy_score(y_test, y_pred)))
    return y_pred
```

```
y_pred = logistic_regression(X_train_pca, X_test_pca, y_train, y_test)
ROC_curve(y_test, y_pred)
sns.heatmap(confusion_matrix(y_test, y_pred),annot = True)
```

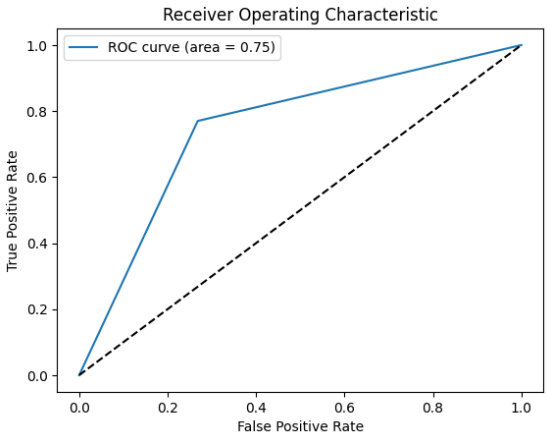

FIGURE 9 – ROC and AUC for Logistic Regression

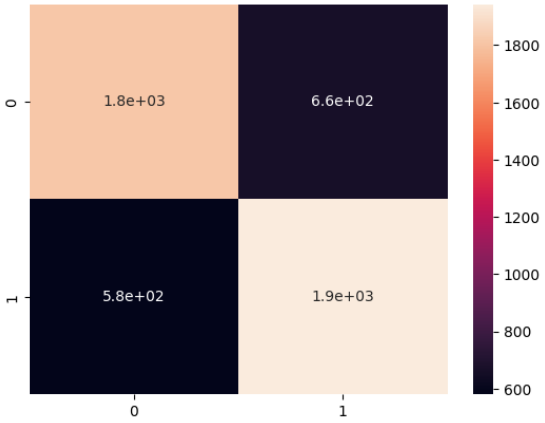

FIGURE 10 – Confusion Matrix for Logistic Regression

	precision	recall	f1-score	support
0	0.76	0.73	0.74	2477
1	0.74	0.78	0.76	2523
accuracy			0.75	5000
macro avg	0.75	0.75	0.75	5000
weighted avg	0.75	0.75	0.75	5000

FIGURE 11 – Accuracy Metrics of Logistic Regression

4.2 KNN (K Nearest Neighbors)

In order to choose the most appropriate k, we trained multiple models with a range of different values for k. Finally we chose the model which had the highest test accuracy.

```
from sklearn.metrics import classification_report
from sklearn.utils.multiclass import type_of_target

def Knn(X_train, X_test, y_train, y_test):
    accuracies = []

    for k in range(1, 11, 2):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)

        y_pred = knn.predict(X_test)
        accuracies.append(accuracy_score(y_test, y_pred))

    plt.plot(range(1, 10, 2), accuracies, marker='o')
    plt.title("Accuracies per n")
    plt.xlabel("K")
    plt.ylabel("Accuracy")
    plt.show()

    KNN = KNeighborsClassifier(n_neighbors = np.argmax(accuracies)*2+1)
    KNN.fit(X_train, y_train)

    y_pred = KNN.predict(X_test)

    print(KNN.predict_proba(X_test))
```

```
    print(classification_report(y_test, y_pred))
    print('KNN model accuracy score: {0:0.4f}'. \
    format(accuracy_score(y_test, y_pred)))
    return y_pred

y_pred = Knn(X_train_pca, X_test_pca, y_train, y_test)
ROC_curve(y_test,y_pred)
sns.heatmap(confusion_matrix(y_test, y_pred),annot = True)
```

Highest accuracy was reached with k = 9 with a test accuracy of 78.12%.

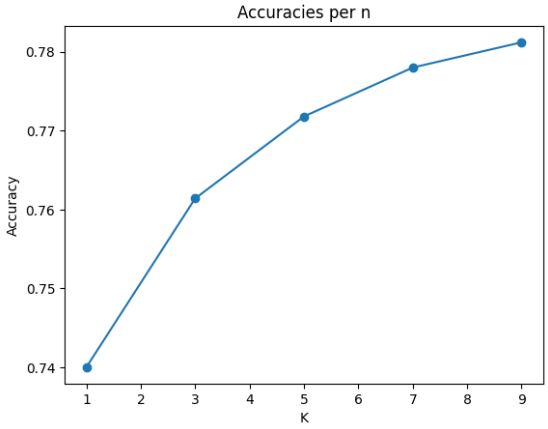

FIGURE 12 – Test Accuracy vs. K

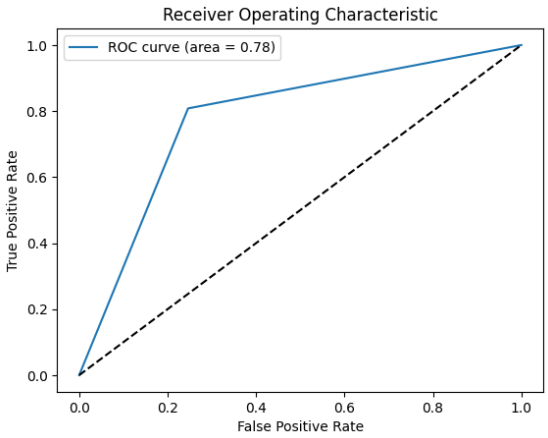

FIGURE 13 – ROC and AUC for KNN

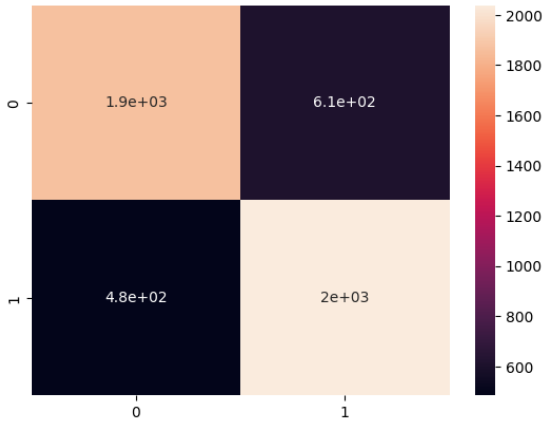

FIGURE 14 – Confusion Matrix for KNN

	precision	recall	f1-score	support
0	0.79	0.75	0.77	2477
1	0.77	0.81	0.79	2523
accuracy			0.78	5000
macro avg	0.78	0.78	0.78	5000
weighted avg	0.78	0.78	0.78	5000

FIGURE 15 – Accuracy Metrics of KNN

4.3 XGBoost

XGBoost (Extreme Gradient Boosting)) is based on the Gradient Boosting Machine (GBM) algorithm. GBM is an ensemble learning technique that builds multiple decision trees sequentially, where each new tree corrects the errors of the previous trees.
The tree-method parameter specifies the tree construction algorithm. "hist" stands for Histogram-based Gradient Boosting, which is an optimized and efficient way to build decision trees, especially for large datasets. Early-stopping-rounds enables early stopping, a regularization technique to prevent overfitting. During training, XGBoost monitors a validation metric (e.g., accuracy, AUC, log loss). If the performance on the validation set does not improve for n consecutive rounds, training stops early. This helps to avoid unnecessary computation and improves generalization.
With tree-method set to "hist" and early-stopping-rounds set to 2, XGBoost reached 79.7% accuracy on test set.

```
def XG_Boost(X_train, X_test, y_train, y_test):
    clf = xgb.XGBClassifier(tree_method="hist", early_stopping_rounds=2)

    clf.fit(X_train, y_train, eval_set=[(X_test, y_test)])

    y_pred = clf.predict(X_test)

    print(classification_report(y_test, y_pred))

    print('XGBoost model accuracy score: {0:0.4f}'. \
    format(accuracy_score(y_test, y_pred)))

    return y_pred

y_pred = XG_Boost(X_train_pca, X_test_pca, y_train, y_test)
ROC_curve(y_test,y_pred)
sns.heatmap(confusion_matrix(y_test, y_pred),annot = True)
```

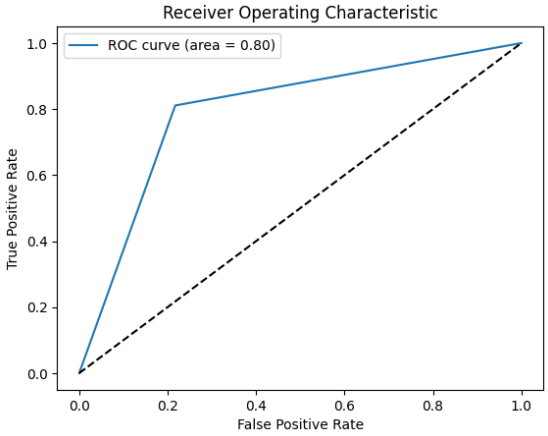

FIGURE 16 – ROC and AUC for XGBoost

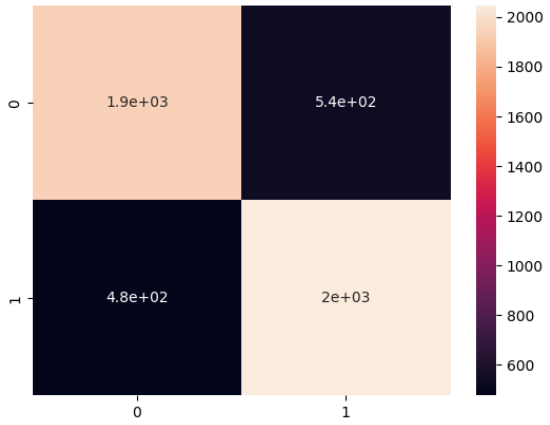

FIGURE 17 – Confusion Matrix for XGBoost

	precision	recall	f1-score	support
0	0.80	0.78	0.79	2477
1	0.79	0.81	0.80	2523
accuracy			0.80	5000
macro avg	0.80	0.80	0.80	5000
weighted avg	0.80	0.80	0.80	5000

FIGURE 18 – Accuracy Metrics of XGBoost

4.4 MLP

After some trial and error we used this architecture for gender classifier MLP which consists of two hidden layers :

```
model = Sequential([
            Dense(16, activation='relu', input_shape=(input_dim,)),
# Hidden Layer 1
            Dense(8, activation='relu'),  # Hidden Layer 2
            Dense(1, activation='sigmoid')
        ])
```

The activation function for the output layer has to be sigmoid as our task is binary classification. The loss function is set to binary cross entropy for the same reason. Also we used Adam optimizer for faster convergence and better performance. This model reached 80.5% accuracy on test set and was the best performing model among all chosen classifiers.

```
def Mlp(X_train, X_test, y_train, y_test):
    def create_mlp(input_dim):
        model = Sequential([
            Dense(16, activation='relu', input_shape=(input_dim,)),
# Hidden Layer 1
            Dense(8, activation='relu'),  # Hidden Layer 2
            Dense(1, activation='sigmoid')
        ])

        model.compile(optimizer=Adam(learning_rate=0.001),
                      loss='binary_crossentropy',
                      metrics=['accuracy'])

        return model

    input_dim = 70
    mlp_model = create_mlp(input_dim)

    history = mlp_model.fit(X_train, y_train, epochs=50, batch_size=16,
    validation_data=(X_test, y_test), verbose=1)

    y_pred = (mlp_model.predict(X_test) >= 0.5).astype(int)

    print(classification_report(y_test, y_pred))

    print('MLP model accuracy score: {0:0.4f}'. \
    format(accuracy_score(y_test, y_pred)))

    return y_pred

y_pred = Mlp(X_train_pca, X_test_pca, y_train, y_test)
ROC_curve(y_test, y_pred)
sns.heatmap(confusion_matrix(y_test, y_pred),annot = True)
```

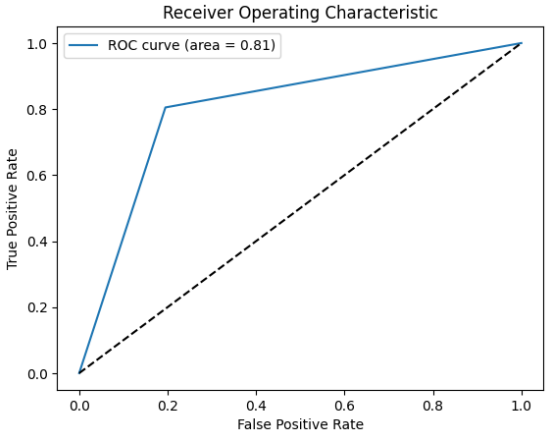

FIGURE 19 – ROC and AUC of MLP classifier

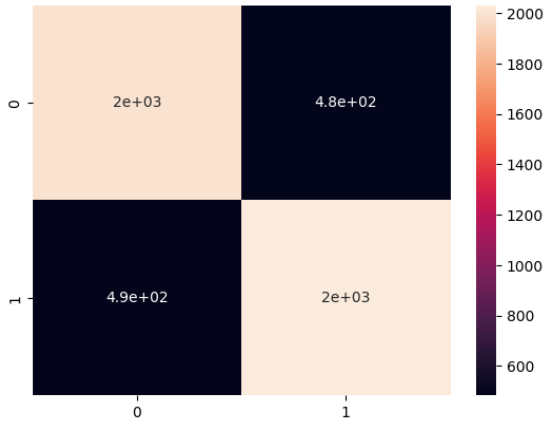

FIGURE 20 – Confusion Matrix of MLP classifier

	precision	recall	f1-score	support
0	0.80	0.81	0.80	2477
1	0.81	0.80	0.81	2523
accuracy			0.81	5000
macro avg	0.80	0.81	0.80	5000
weighted avg	0.81	0.81	0.81	5000

FIGURE 21 – Accuracy Metrics of MLP

5 Identity Classification

In this section for 3 iterations, we randomly sampled 6 students by their student IDs, encoded their IDs from 0 to 5 and filtered out the data frame the get the records with these IDs. Like gender classification we first split the data to train and test set, applied Min-Max scale on the dataset and finally performed PCA dimension reduction keeping first 70 components. These steps were done on all the 3 batches and then the models were fitted in each batch seperately. The models used for this classification were :
1) KNN
2) XGBoost
3) MLP

```
def ROC_curve_multi(y_test, y_pred_proba, n_classes):
    """
    Plots ROC curves for multi-class classification.

    Parameters:
    - y_test: True labels (one-hot encoded).
    - y_pred_proba: Predicted probabilities for each class.
    - n_classes: Number of classes.
    """
    y_test_bin = label_binarize(y_test, classes=np.arange(n_classes))
    print(y_test_bin.shape)
    fpr, tpr, roc_auc = {}, {}, {}

    plt.figure(figsize=(8, 6))
    colors = cycle(["blue", "red", "green",\
    "purple", "orange", "brown"])

    for i, color in zip(range(n_classes), colors):
        fpr[i], tpr[i], _ = roc_curve(y_test_bin[:, i],\
        y_pred_proba[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])
        plt.plot(fpr[i], tpr[i], color=color, lw=2,\
        label=f"Class {i} (AUC = {roc_auc[i]:.2f})")

    plt.plot([0, 1], [0, 1], linestyle="--", color="gray")
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
```

```
    plt.title("Multi-Class ROC Curve")
    plt.legend(loc="lower right")
    plt.grid()
    plt.show()

Unique_IDs = df.iloc[:, -1].unique()

for i in range(3):
    Unique_IDs_sample = np.random.choice(Unique_IDs, size=6,\
    replace=False)
    df_sample = df[df.iloc[:, -1].isin(Unique_IDs_sample)]
    df_sample.iloc[:, -1] = df_sample.iloc[:, -1].\
    replace(Unique_IDs_sample, list(range(6)))
    df_sample = df_sample.values

    X_train, X_test, y_train, y_test = \
    train_test_split(df_sample[:, :-2], df_sample[:, -1],\
    test_size=0.25, random_state=0)

    scaler = MinMaxScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    n_components = 70
    pca = PCA(n_components=n_components)
    X_train_pca = pca.fit_transform(X_train_scaled)
    X_test_pca = pca.transform(X_test_scaled)

    y_train = y_train.astype(int)
    y_test = y_test.astype(int)

    for model_name, model_function in [("KNN", Knn),\
    ("XGBoost", XG_Boost),("MLP", Mlp_2)]:
        y_pred_proba = \
        model_function(X_train_pca, X_test_pca, y_train, y_test)
        print(y_pred_proba.shape)
        print(f"Evaluating {model_name}:")
        ROC_curve_multi(y_test, y_pred_proba, 6)
```

The performance measures of each of the classifiers on the 3 batches are reported.

5.1 KNN

As before, we tested different values for k and chose the best one based on their performance on test set. The code of its' implementation is the same as before; except the fact that in all of our classifiers, we returned the y_proba (a matrix that indicates the probability of belonging to each class for each record) except the y_pred like the binary case.

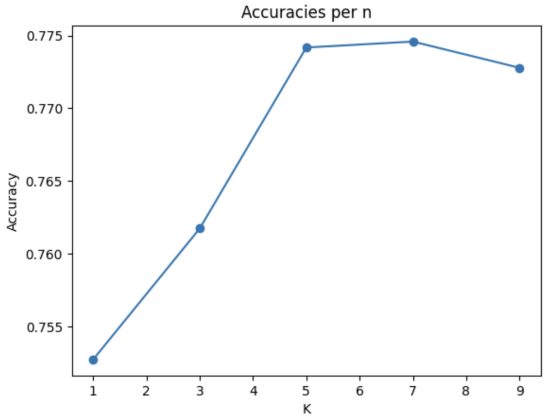

FIGURE 22 – Accuracy vs. K (batch 1)

	precision	recall	f1-score	support
0	0.78	0.80	0.79	2000
1	0.78	0.79	0.78	1955
2	0.70	0.86	0.77	1963
3	0.77	0.78	0.78	1997
4	0.83	0.70	0.76	2417
5	0.81	0.73	0.77	1881
accuracy			0.77	12213
macro avg	0.78	0.78	0.77	12213
weighted avg	0.78	0.77	0.77	12213

FIGURE 23 – Accuracy Metrics of KNN (batch 1)

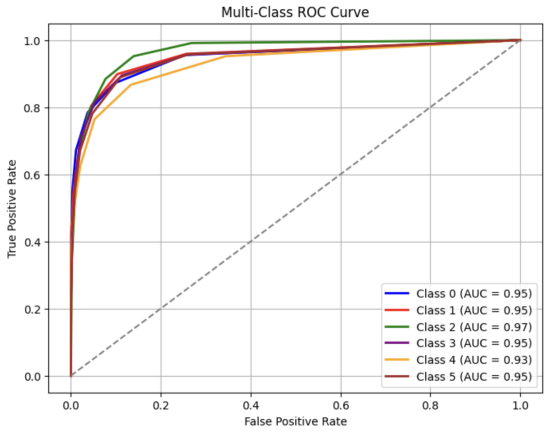

FIGURE 24 – ROC and AUC of KNN (batch 1)

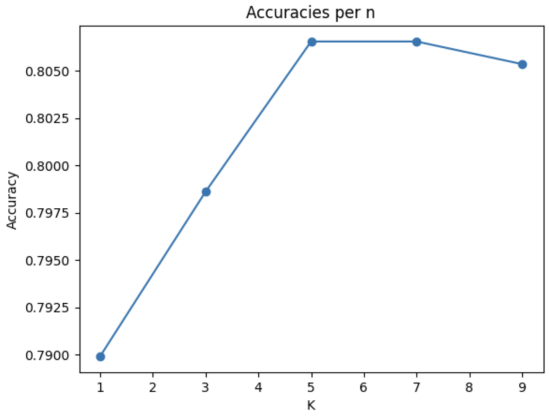

FIGURE 25 – Accuracy vs. K (batch 2)

	precision	recall	f1-score	support
0	0.70	0.85	0.77	2426
1	0.88	0.76	0.82	1992
2	0.91	0.82	0.86	499
3	0.82	0.72	0.77	1934
4	0.82	0.79	0.80	2456
5	0.84	0.88	0.86	2417
accuracy			0.81	11724
macro avg	0.83	0.80	0.81	11724
weighted avg	0.81	0.81	0.81	11724

FIGURE 26 – Accuracy Metrics of KNN (batch 2)

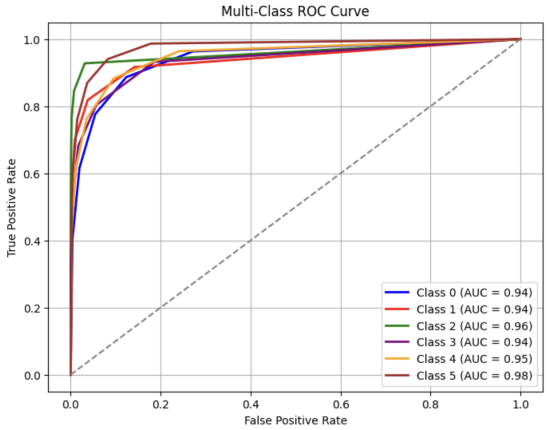

FIGURE 27 – ROC and AUC of KNN (batch 2)

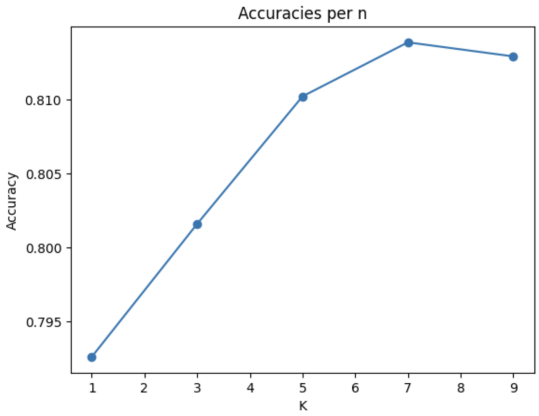

FIGURE 28 – Accuracy vs. K (batch 3)

	precision	recall	f1-score	support
0	0.67	0.71	0.69	1945
1	0.80	0.93	0.86	2513
2	0.76	0.83	0.79	2438
3	0.89	0.80	0.85	2479
4	0.87	0.82	0.84	2378
5	0.93	0.75	0.83	1925
accuracy			0.81	13678
macro avg	0.82	0.81	0.81	13678
weighted avg	0.82	0.81	0.81	13678

FIGURE 29 – Accuracy Metrics of KNN (batch 3)

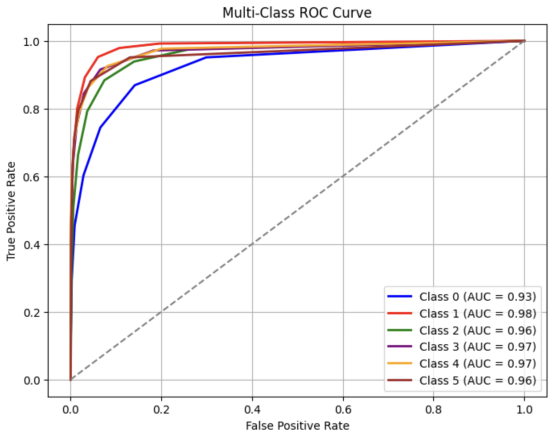

FIGURE 30 – ROC and AUC of KNN (batch 3)

5.2 XGBoost

Like gender classification, the tree_method parameter was set to "hist" and the early_stopping_rounds was set to 2. The performance measures in the 3 batches are reported below.

	precision	recall	f1-score	support
0	0.86	0.83	0.85	2000
1	0.86	0.83	0.84	1955
2	0.87	0.88	0.88	1963
3	0.84	0.86	0.85	1997
4	0.92	0.96	0.94	2417
5	0.87	0.86	0.86	1881
accuracy			0.87	12213
macro avg	0.87	0.87	0.87	12213
weighted avg	0.87	0.87	0.87	12213

FIGURE 31 – Accuracy Metrics of XGBoost (batch 1)

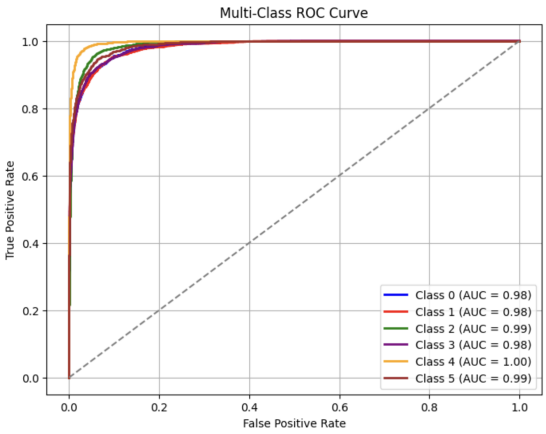

FIGURE 32 – ROC and AUC of XGBoost (batch 1)

	precision	recall	f1-score	support
0	0.87	0.90	0.88	2426
1	0.91	0.87	0.89	1992
2	0.98	0.79	0.87	499
3	0.88	0.85	0.87	1934
4	0.87	0.89	0.88	2456
5	0.91	0.94	0.92	2417
accuracy			0.89	11724
macro avg	0.90	0.87	0.89	11724
weighted avg	0.89	0.89	0.89	11724

FIGURE 33 – Accuracy Metrics of XGBoost (batch 2)

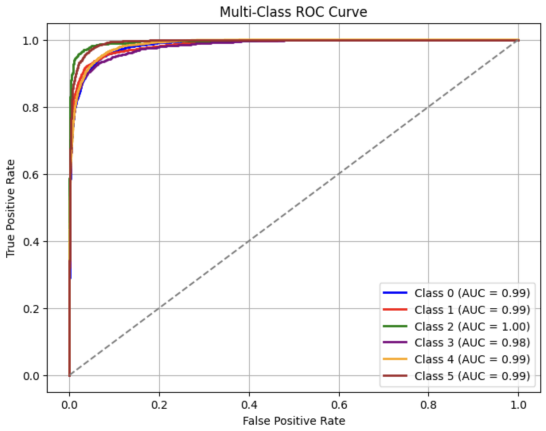

FIGURE 34 – ROC and AUC of XGBoost (batch 2)

	precision	recall	f1-score	support
0	0.85	0.84	0.85	1945
1	0.92	0.95	0.93	2513
2	0.89	0.90	0.90	2438
3	0.94	0.96	0.95	2479
4	0.91	0.90	0.91	2378
5	0.95	0.91	0.93	1925
accuracy			0.91	13678
macro avg	0.91	0.91	0.91	13678
weighted avg	0.91	0.91	0.91	13678

FIGURE 35 – Accuracy Metrics of XGBoost (batch 3)

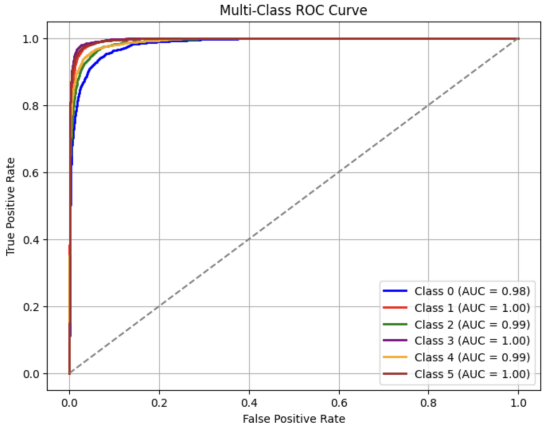

FIGURE 36 – ROC and AUC of XGBoost (batch 3)

5.3 MLP

After trial and error we chose this architecture for our MLP :

```
model = Sequential([
            Dense(32, activation='relu', input_shape=(input_dim,)),
# Hidden Layer 1
            Dropout(0.2),
            Dense(16, activation='relu'),  # Hidden Layer 2
            Dropout(0.2),
            Dense(6, activation='softmax')
        ])
```

The output layer has to have 6 units with softmax activation function as our task is a classification with 6 classes. We set dropout of 20% to avoid overfitting. The

performance measures in each batch is provided below.

	precision	recall	f1-score	support
0	0.88	0.82	0.85	2000
1	0.84	0.85	0.84	1955
2	0.84	0.92	0.88	1963
3	0.87	0.85	0.86	1997
4	0.96	0.94	0.95	2417
5	0.89	0.89	0.89	1881
accuracy			0.88	12213
macro avg	0.88	0.88	0.88	12213
weighted avg	0.88	0.88	0.88	12213

Test Accuracy: 0.8822

FIGURE 37 – Accuracy Metrics of MLP (batch 1)

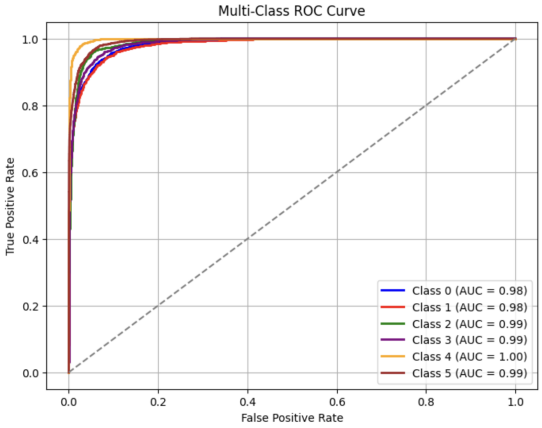

FIGURE 38 – ROC and AUC of MLP (batch 1)

	precision	recall	f1-score	support
0	0.84	0.86	0.85	2426
1	0.89	0.76	0.82	1992
2	0.93	0.74	0.83	499
3	0.88	0.79	0.83	1934
4	0.77	0.90	0.83	2456
5	0.89	0.93	0.91	2417
accuracy			0.85	11724
macro avg	0.87	0.83	0.84	11724
weighted avg	0.85	0.85	0.85	11724

Test Accuracy: 0.8493

FIGURE 39 – Accuracy Metrics of MLP (batch 2)

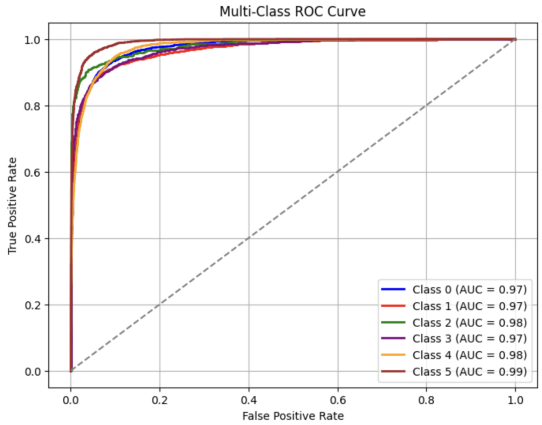

FIGURE 40 – ROC and AUC of MLP (batch 2)

	precision	recall	f1-score	support
0	0.72	0.73	0.73	1945
1	0.87	0.93	0.90	2513
2	0.86	0.80	0.83	2438
3	0.95	0.93	0.94	2479
4	0.87	0.86	0.86	2378
5	0.88	0.87	0.87	1925
accuracy			0.86	13678
macro avg	0.86	0.85	0.85	13678
weighted avg	0.86	0.86	0.86	13678

Test Accuracy: 0.8594

FIGURE 41 – Accuracy Metrics of MLP (batch 3)

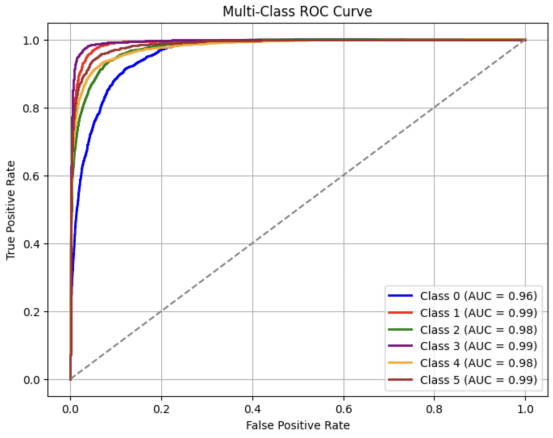

FIGURE 42 – ROC and AUC of MLP (batch 3)

Based on our results, we can say that XGBoost and MLP have performed better than KNN and they have approximately performed equally well.

6 Clustering

6.1 KMeans

In this section we first apply KMeans clustering algorithm to our data and we found the best "K"(number of clusters) using two measures(Inertia and silhouette score).

we first apply elbow method that plot the Inertia for each classifier with the specific K. Inertia measures how well a dataset was clustered by K-Means. It is calculated by measuring the distance between each data point and its centroid, squaring this distance, and summing these squares across one cluster.

A lower inertia indicates that data points are closer to their respective cluster centers, suggesting more cohesive and well-defined clusters.

```
def elbow_method(df):
    Sum_of_squared_distances = []
    K = range(1,10,2)
    for num_clusters in K :
        kmeans = KMeans(n_clusters=num_clusters)
        kmeans.fit(df)
        Sum_of_squared_distances.append(kmeans.inertia_)
    plt.plot(K, Sum_of_squared_distances,'bx-')
    plt.xlabel("Values of K")
    plt.ylabel("Sum of squared distances/Inertia")
    plt.title("Elbow Method For Optimal")
    plt.show()
    return
```

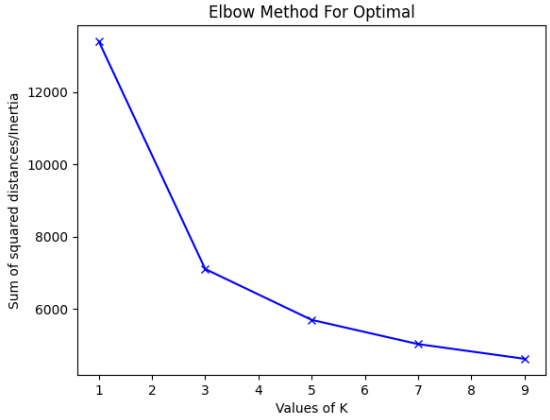

FIGURE 43 – KMeans elbow method with Inertia

As we can see the elbow occurs in K = 3 (we can also say K = 7). So the best k is 3.

Then we calculate silhouette score for a range of K(number of clusters) and plot silhouette score vs number of clusters.

The interpretation of the silhouette score measures the quality of the silhouette score of k-means by evaluating how well the data points group within their assigned groups compared to the data points in other groups.

It considers two distances for each data point :

a : Average distance between the data point and all other data points within the same cluster (intra-cluster distance).

b : Distance between the data point and the nearest cluster that the data point doesn't belong to (inter-cluster distance).

The silhouette score for a data point is then calculated as :

$$silhouette\_score = \frac{b - a}{max(a, b)} \tag{1}$$

The range of silhouette score is -1 to 1. The k that results in the highest average silhouette score. indicates a clustering where data points are well-separated within their clusters.

```
def silhouette(df):
    range_n_clusters = [2, 3, 4, 5, 6, 7, 8]
    silhouette_avg = []
    for num_clusters in range_n_clusters:

        kmeans = KMeans(n_clusters=num_clusters)
        kmeans.fit(df)
```

```
        cluster_labels = kmeans.labels_

        silhouette_avg.append(silhouette_score(df,
        cluster_labels,metric="cosine"))
    plt.plot(range_n_clusters,silhouette_avg,'bx-')
    plt.xlabel("Values of K")
    plt.ylabel("Silhouette score")
    plt.title("Silhouette analysis For Optimal k")
    plt.show()
    return
```

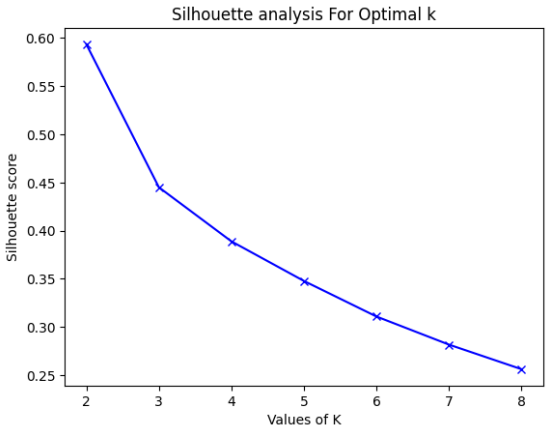

FIGURE 44 – Silhouette Score vs. K in KMeans

As we can see in the plot the best score occurs in K = 2.

6.1.1 KMeans result with different K

For comparison, we also plotted the output of KMeans some values of K. The plotting is done with the first 2 PCA components of our dataset.

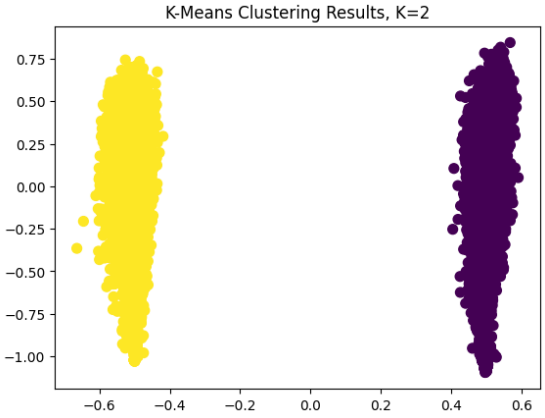

FIGURE 45 – 2D KMeans result for K=2

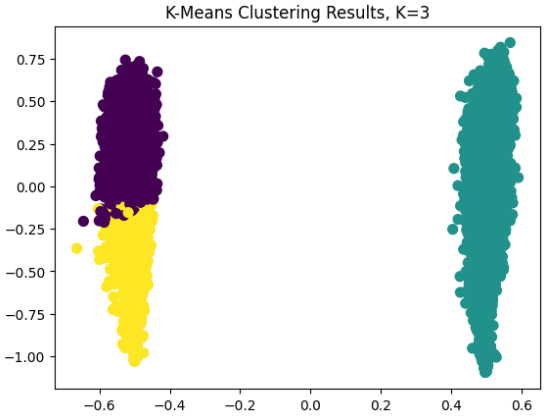

FIGURE 46 – 2D KMeans result for K=3

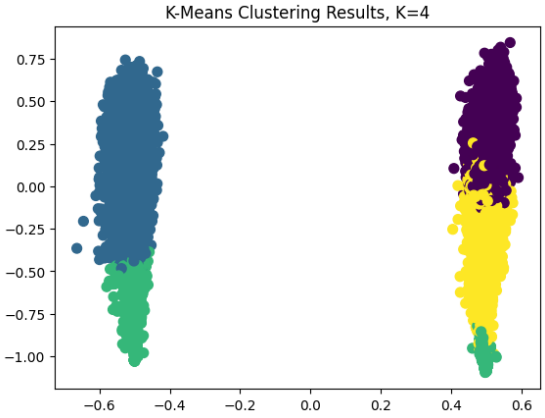

FIGURE 47 – 2D KMeans result for K=4

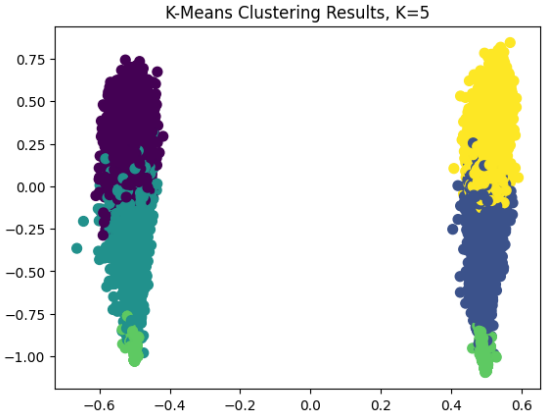

FIGURE 48 – 2D KMeans result for K=5

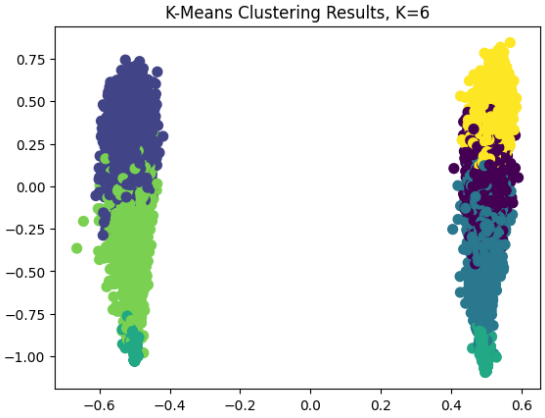

FIGURE 49 – 2D KMeans result for K=6

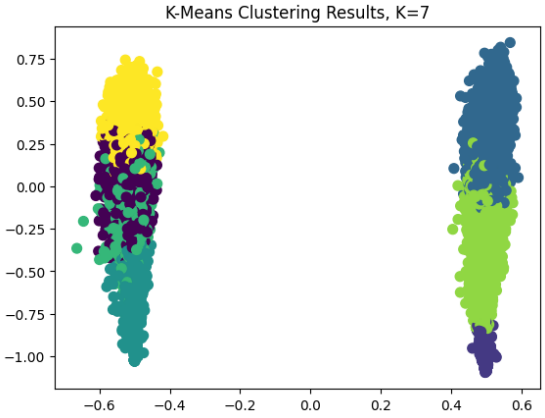

FIGURE 50 – 2D KMeans result for K=7

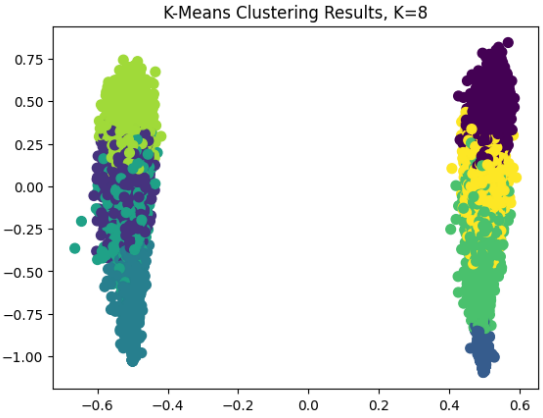

FIGURE 51 – 2D KMeans result for K=8

6.2 DBScan

We also use another clustering algorithm that is density based despite the KMeans that was distance based.

And we evaluate this algorithm by silhouette score. The best hyper-parameters were eps = 0.9 and min_samples = 5 that get silhouette score of 0.34. Then we plot the data points using the first and second pca components. For achieving the best model, we fit multiple models using different values for radius (eps). As expected, the number of clusters decrease as we increase the eps parameter.

```
for eps in [0.5,0.6,0.7,0.8,0.9]:

    dbscan = DBSCAN(eps=eps, min_samples=5)
    y_dbscan = dbscan.fit_predict(dataset_pca)

    print("silhouette_score is ",\
    silhouette_score(dataset_pca,y_dbscan), "for eps = ", eps)

    print("The number of cluster and their count's of them for eps =",\
    eps, 'is', Counter(y_dbscan))

    plt.figure()
    plt.scatter(dataset_pca[:, 0], dataset_pca[:, 1], c=y_dbscan,\
    s=50, cmap='viridis')
    plt.title("DBSCAN Clustering Results")
    plt.show()
```

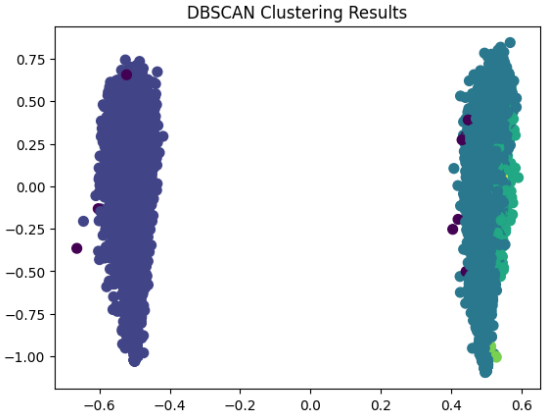

FIGURE 52 – 2D DBScan result for eps = 0.5

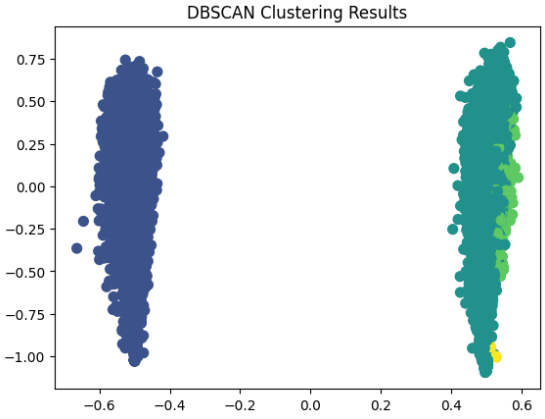

FIGURE 53 – 2D DBScan result for eps = 0.6

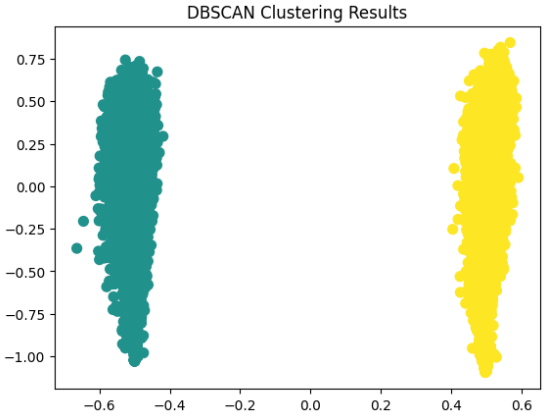

FIGURE 54 – 2D DBScan result for eps = 0.7

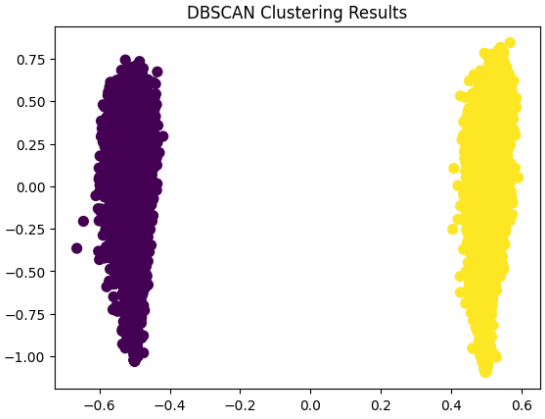

FIGURE 55 – 2D DBScan result for eps = 0.8

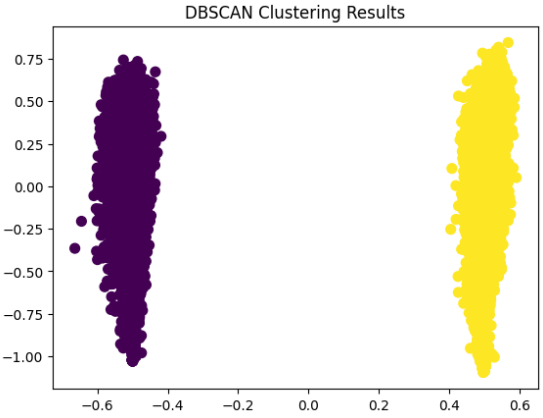

FIGURE 56 – 2D DBScan result for eps = 0.9

The clusters were identical with eps = 0.8 and eps = 0.9 (they basically partitioned the dataset based on gender judging from the results) and eps = 0.7 also partitioned similarly except the fact that there was also a cluster with size 1.