Statistical Learning Theory Final Project Report

Stuttering Analysis

[Researching Stuttering Severity in Different Situations]

Author(s): [Donia Shebrawi, Ghaidaa Zoabi]

Course Name: Statistical Learning Theory

Submission Date: 23.06.2024

Abstract

This study delves into stuttering behaviors across diverse speech contexts using data sourced from the UCLASS archives. Through meticulous audio analysis, we aimed to elucidate stuttering frequency and severity, specifically focusing on accurately classifying speech situations based on acoustic features. Utilizing statistical analysis and machine learning models, we identified significant differences in stuttering severity between conversational speech and reading, even when considering reading and monologue as a unified category. Our findings underscore the pivotal role of speech context in informing stuttering interventions. By combining reading and monologue into one category and conversation into another, we observed improved model performance, highlighting the unique nature of reading and monologue speech contexts. Despite limitations stemming from the relatively small dataset within the UCLASS archives, our study advocates for the utilization of larger and more diverse datasets for future research to ensure broader generalizability and enhanced therapeutic applications in stuttering interventions.

1 Introduction

Stuttering, a widespread communication disorder, poses significant challenges worldwide, with its underlying causes remaining complex and elusive despite extensive research. Recognizing the crucial role of speech contexts in stuttering interventions, this study investigates stuttering behaviors across various speech scenarios using the UCLASS archives. Through thorough audio analysis, we aim to elucidate stuttering frequency and severity, focusing on accurately classifying speech situations based on acoustic features and understanding their correlation with stuttering severity. By addressing these objectives, we aim to improve speech therapy interventions, providing more effective support for individuals who stutter.

2 Data description

2.1 Data Source

The study utilized data from the UCLASS archives, comprising recordings of speakers who stutter across diverse speech situations (reading, conversation, monologue) and conditions (normal speaking, altered listening). These recordings were obtained from controlled laboratory studies, offering background information about speakers and recording conditions for in-depth analysis of stuttering behaviors across contexts.

2.2 Description of Variables

In our data we have both categorical and numerical variables:

Categorical variables: Family history, Handedness, Gender, Where recorded, Recording conditions, Type of therapy, Hearing problems, Language problems, Sen, Orthographic, Phl, Phonetic, Time aligned Situation.

numeric variables: Speaker quality, Environmental noise, Time between therapy and recording, Interruption, Age at onset, Age at recording, Audio score, Pitch score, Intensity score, Disfluency duration score, Spectral feature scalar.

In addition to extracting these features from the data, we will delve into the extraction process in the modeling section.

1. pitch - representing the fundamental frequency of speech.
2. intensity - measures the strength or loudness of the speech signal.
3. disfluency duration - quantifies the duration of pauses or hesitations in speech. which helps us figure how much time someone spends hesitating or pausing whie speaking.
4. spectral feature scalar - The spectral feature scalar captures spectral characteristics of speech using Mel-frequency cepstral coefficients (MFCCs).

2.3 Exploratory Data Analysis (EDA)

2.3.1 Cleaning Plot for Audio Data

The noise reduction method reduces background noise in each audio file, enhancing recording quality. Cleaned audio data is then saved back into the original file, ensuring consistency and improved clarity for analysis.
Here is an example of audio cleanup involving comparing spectrograms before and after noise reduction.

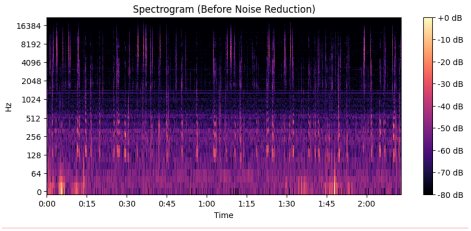


Figure 1: Original Audio Spectrogram

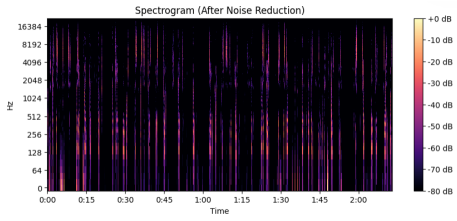


Figure 2: Cleaned Audio Spectrogram

2.3.2 Scatter plot

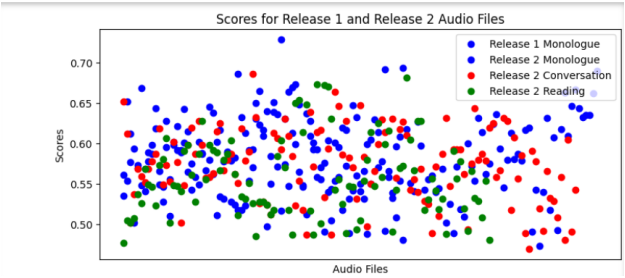


Figure 3: Stuttering severity scores in different situations

The scatter plot displays severity scores across monologue, conversation, and reading contexts, where monologue recordings predominate. Visual differentiation between groups is challenging. ANOVA testing, detailed in the next section, offers clearer insights into speech context disparities.

2.3.3 Audio severity plot

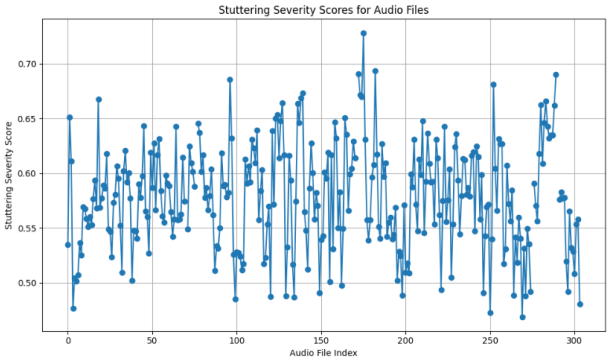


Figure 4: Stuttering severity score for the audios

The stuttering severity scores plot shows most scores clustered between 0.5 and 0.65, with some reaching 0.8, indicating more severe stuttering. (This values range is a result of the normalization process).
Note: Some recordings are missing due to file issues.

2.4 Clustering

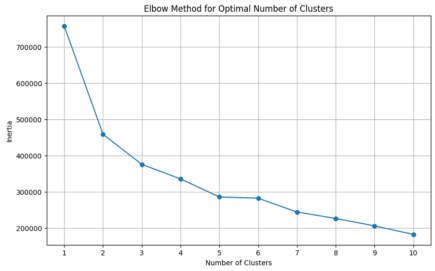
We employed clustering techniques, including k-means, to group similar data points together, aiming to partition the dataset into distinct clusters based on their similarity.
We used the original data for clustering.

2.4.1 k-means clustering with PCA

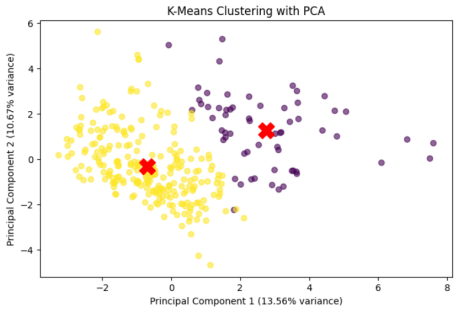
K-Means clustering was applied to identify two distinct clusters. Preprocessing includes standardizing numerical features and one-hot encoding categorical features. To determine the optimal number of clusters, the elbow method was used, which indicated that k=2 is appropriate. The results were visualized using PCA, reducing the data to two dimensions. The plot illustrates effective segmentation into two clusters, offering insights for further analysis and strategies.

2.4.2 Elbow method

We used the elbow method to find the optimal number of clusters for k-means clustering. By plotting the explained variation against the number of clusters, we identified the elbow point, which indicated that the optimal number of clusters is 2.



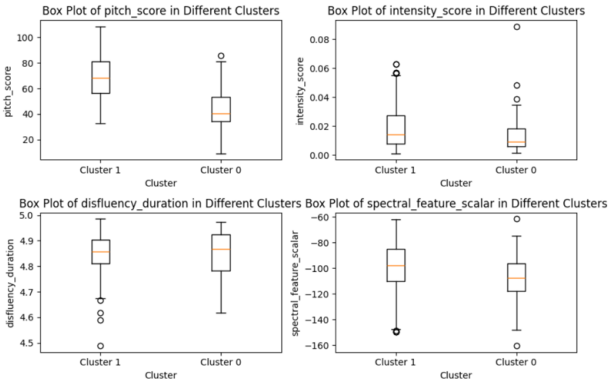
(a) Elbow method for choosing the k



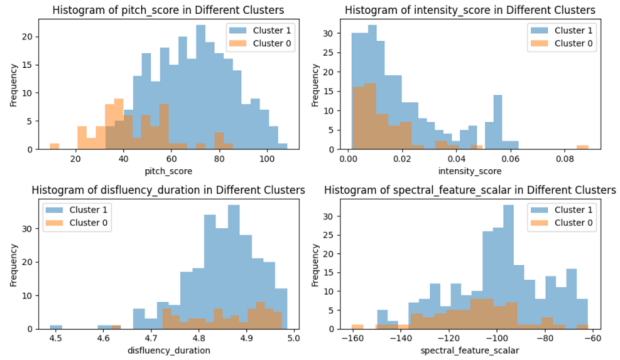
(b) PC's Plot of k-means (k=2)

Figure 5: Elbow method and PC's plot of k-means clustering

As we can see we plotted the 2 PC's and it shows separated two clusters.



(a) Box-plots across clusters



(b) Frequencies across clusters

Figure 6: Visualizing Clustering Results: Feature Distributions and Frequencies

The plots above illustrate the distribution of features across different clusters. We observe that the blue cluster contains more information compared to the second cluster. In the "pitch disfluency score" feature, cluster 1 shows a higher frequency of observations with high values. Conversely, the "intensity score" feature exhibits a small frequency of low values in both clusters 1 and 0. The box plots reveal that the mean values in cluster 0 are notably lower than those in cluster 1.

3 Modeling

3.1 Feature Engineering

We segment each audio file by its length and then extract feature values using specialized algorithms tailored to each feature type :

1. **Pitch**: Measured using Praat, we calculate the average fundamental frequency (pitch) across each audio segment.
2. **Intensity**: Calculated as the root mean square (RMS) amplitude of the segment, providing a measure of its loudness.
3. **Disfluency Duration**: Quantified by detecting pauses in speech segments using the short-time Fourier transform (STFT) energy envelope, summing the duration of identified pauses.
4. **Spectral Feature Scalar (MFCCs)**: Extracted from the segment's power spectrogram using Mel-frequency cepstral coefficients (MFCCs), averaged across coefficients to capture spectral characteristics.

For each segment, these feature values are averaged to obtain segment-level scores, and the mean of these scores across all segments within each audio file provides the final feature score used for analysis and comparison across recordings.

3.2 Audio stuttering severity score Calculation

To generate a comprehensive measure, we used weighted sums of the scores from different features. Finally, we normalized the combined score to ensure consistency across recordings, facilitating fair comparisons and interpretation.

3.2.1 Handling missing data

instances with less than 40 percent missing values were retained while others were removed. Imputation techniques addressed missing values in crucial variables to ensure data completeness. All remaining variables were included in the analysis after imputation, with categorical variables encoded as factors to facilitate modeling.

3.3 Data split

The dataset was split 70-30 for training and testing to impartially assess model performance. We applied SVM, Random Forest, KNN, and Logistic Regression, each trained and tested using this split and further evaluated through cross-validation techniques for comprehensive assessment. Additionally, we performed these methods on data that included clusters for comparative analysis. Subsequent sections will detail the results and discussions of these modeling efforts.

3.4 mathematical formulations

3.5 Support vector machine -linear

For a linear SVM, the decision function can be expressed as:

$[f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b]$

where $\mathbf{w}$ is the weight vector and b is the bias term. The optimal hyperplane maximizes the margin, which is the distance between the hyperplane and the nearest data points from both classes, known as support vectors.

3.5.1 Optimization Problem and Dual Formulation

The optimization problem for a linear Support Vector Machine (SVM) is to minimize the objective function:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$
$$\text{subject to} \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \quad \forall i$$

This problem can also be expressed in its dual form using Lagrange multipliers α_i, maximizing:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i\alpha_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j$$
$$\text{subject to} \quad \sum_{i=1}^{n} \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C$$

C regulates the trade-off between maximizing the margin and minimizing classification error.

3.5.2 Decision Boundary

The decision boundary is given by:
$[f(\mathbf{x}) = \sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b]$
A new sample $\mathbf{x}$ is classified as:
$[\text{class}(\mathbf{x}) = \text{sign}(f(\mathbf{x}))]$

3.5.3 Application to the Project

We combined 'Reading' and 'Monologue' into category 0 and 'Conversation' into category 1. After feature extraction and labeling, the dataset was split for training and testing. Using a linear SVM, the model accurately predicted speech situations' categories, facilitating effective evaluation of stuttering levels in speech contexts.

Accuracy:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \qquad (*)$$

3.6 k-Nearest Neighbors (k-NN) Method

Distance Metrics: The Euclidean distance metric is used to measure similarity between data points. The Euclidean distance between two points $\mathbf{x}_i$ and $\mathbf{x}_j$ in an n-dimensional space is given by:

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{\sum_{l=1}^{n} (x_{i,l} - x_{j,l})^2}$$

The predicted class for a test point is determined by majority vote among its k nearest neighbors.

$$\hat{y} = \text{mode}\{y_i : \mathbf{x}_i \in \mathcal{N}_k(\mathbf{x}_*)\}$$

where y_i is the class label of the i-th training sample.

Model Evaluation: Accuracy measures the proportion of correctly predicted samples out of the total number of samples(By equation *)

3.7 Random Forest Classifier

Random Forest Classifier uses multiple decision trees to predict by selecting the most common class among the trees.

Bootstrap Sampling: Given a training dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$: where x_i represents the feature, and y_i represent the target labels. Generate B bootstrap samples by randomly selecting n data points from the original dataset with replacement, denoted as D_b for the b-th sample.

Training Decision Trees: For each bootstrap sample D_b: Train a decision tree T_b on D_b. At each node of the tree, select a random subset of m features from the total d features ($m \leq d$). Split the node using the best feature j from this subset based on Gini impurity.

Gini Impurity:

Gini Impurity measures the impurity of a node. It is calculated as:
$G(t) = 1 - \sum_{k=1}^{K} p_k^2$
where p_k is the proportion of samples belonging to class k at node t.
Prediction: For a new data point $\mathbf{x}$, each of the B trained decision trees T_b generates a class prediction $\hat{y}_b(\mathbf{x})$. The final predicted class $\hat{y}$ is determined by majority voting of these predictions. We used 100 decision trees in our Random Forest classifier.

3.8 Logistic regression classifier

3.8.1 Logistic Function

The logistic function is defined as:

$$P(Y = 1|X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

where: - $P(Y = 1|X)$ is the probability that the output variable Y is equal to 1 given the input variable X, - β_0 is the intercept term, - β_1 is the coefficient for the input variable X, - e is the base of the natural logarithm.

3.8.2 Decision Boundary and Classification

In logistic regression, the decision boundary is a hyperplane that separates classes, defined by an equation for binary classification.

$$\mathbf{w} \cdot \mathbf{x} + b = 0$$

where: - $\mathbf{w}$ is the vector of weights (coefficients), - $\mathbf{x}$ is the input feature vector, - b is the intercept (bias term).

For multi-class classification, the model learns multiple such hyperplanes to separate each pair of classes. The classifier uses the predicted probability and a decision threshold to assign class labels. Typically, the threshold is 0.5. For a given test point $\mathbf{x}_*$, the predicted class $\hat{y}$ is:

$$\hat{y} = \begin{cases} 1 & \text{if } P(Y = 1|\mathbf{x}_*) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

3.8.3 Model Training

The logistic regression model is trained using maximum likelihood estimation to find coefficients β_0 and β_1 maximizing data likelihood. Our analysis utilized multinomial logistic regression with the 'saga' solver and a maximum of 2000 iterations.

4 Results

4.0.1 ANOVA test results

ANOVA test showed a significant difference in mean audio stuttering severity scores among the groups. Post hoc comparisons identified that the significant difference is specifically between the conversation and reading groups.

Source	F-statistic	p-value
Between Groups (ANOVA)	9.049	0.00014***

Table 1: ANOVA Results for Comparing Groups of Audio Stuttering Severity Scores

Comparison	p-value
Conversation - Monologue	0.8544
Conversation - Reading	0.0372**
Monologue - Reading	0.2191

Table 2: Post Hoc Comparisons for Audio Stuttering Severity Scores in all groups pairs

4.0.2 Machine learning methods results

Logistic regression and SVM methods achieved the highest accuracy of 0.53 and 0.56, surpassing other method , however they stil not reach the optimal accuracy:

Model	Accuracy	Mean CV score	std CV score
Support Vector Machine (SVM)	0.532609	0.550820	0.069242
Random Forest	0.380435	0.534426	0.087732
K-Nearest Neighbors (KNN)	0.434783	0.445902	0.097482
Logistic Regression	0.565217	0.537705	0.090030

Table 3: Machine Learning Methods Accuracy Results: Cross-validation and Test-Train Split, Across Three Categories of Situation Variable

Based on ANOVA results and machine learning performance, we opted to merge the "reading" and "monologue" categories into one group and the "conversation" category into another for further analysis.

4.0.3 Machine learning methods results-combined categories

When looking at the accuracy of each model with combined categories, the increased mean cross-validation (CV) score and decreased standard deviation suggest improved performance and consistency.

Model	Accuracy	Mean CV score	std CV score
Support Vector Machine (SVM)	0.75	0.665574	0.051424
Random Forest	0.641304	0.652459	0.052254
K-Nearest Neighbors (KNN)	0.619565	0.636066	0.054772
Logistic Regression	0.728261	0.659016	0.059016

Table 4: Machine Learning Methods Accuracy Results: Cross-validation and Test-Train Split, Across combined Categories of Situation Variable

Now , We enter the clusters to the model as a variable which migh helps capture hidden patterns and improves prediction accuracy by providing additional context about the data structure. here is the results that we get:

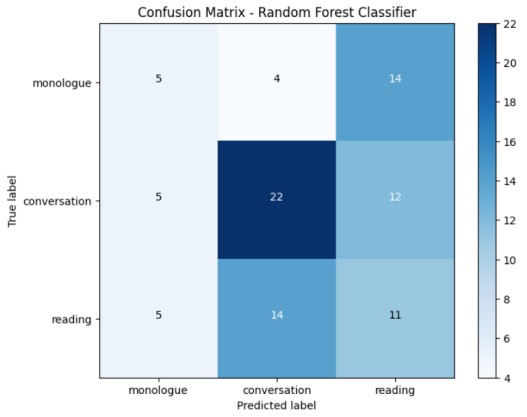
Model	Accuracy (Combined)	Mean CV score	std CV score
Support Vector Machine (SVM)	0.760870	0.672131	0.054863
Random Forest	0.641304	0.619672	0.053272
K-Nearest Neighbors (KNN)	0.6195655	0.632787	0.057354
Logistic Regression	0.728261	0.659016	0.059016

Table 5: Machine Learning Methods Accuracy Results Across combined Categories of Situation Variable with adding Cluster variable

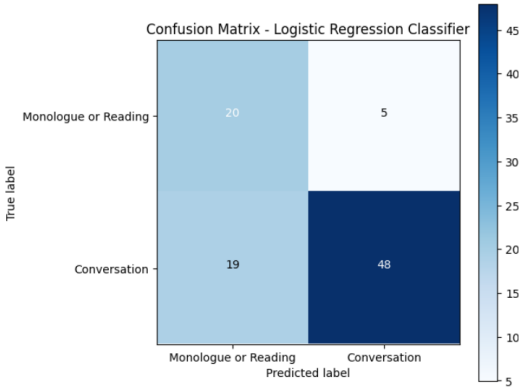
The addition of cluster variables resulted in a slight increase in accuracy, although not consistently across all methods, indicating minimal impact on the overall results.

4.0.4 Comparing between original categories and combined ones

Here's a comparison of confusion matrices for Random Forest before and after combining 'reading' and 'monologue' into a single category.



(a) Initial categories



(b) Combined categories

Figure 7: Confusion Matrices of Random Forest Method: Comparing Initial and Combined Categories in Situation Variable

The combined categories result in an improved overall accuracy of 73.91% (calculated as $\frac{20+48}{20+48+5+19}$), compared to 47.5% accuracy for the initial categories (calculated as $\frac{22+11+5}{22+11+5+4+14+5+5+14}$). This improvement indicates that the model performs better when combining reading and monologue.

5 Conclusion

This study investigated stuttering behaviors across diverse speech contexts using data sourced from the UCLASS archives. Significant differences in stuttering severity were identified through ANOVA analysis ($F=9.049$, $P=0.00014$), with post hoc tests revealing a notable distinction between conversation and reading contexts ($P=0.0372$).

Machine learning techniques including SVM, Random Forests, KNN, and logistic regression were applied to the data initially yielding modest accuracy (0.3804-0.5652). However, consolidating reading and monologue into one category and conversation into another significantly improved model performance (0.619-0.75), highlighting distinct stuttering patterns between solitary and interactive speech situations.

The inclusion of cluster variables did not substantially enhance accuracy, indicating limited additional predictive value in this context.

These findings underscore the nuanced relationship between speech context and stuttering severity, suggesting potential therapeutic implications. Addressing communication dynamics and social interaction may positively influence stuttering outcomes. Despite limitations, notably the small dataset (304 observations), this study provides valuable insights.

Future research should prioritize larger and more diverse datasets to further elucidate these findings and enhance the robustness of therapeutic interventions.

6 Appendix

```
# Installing packages
!pip install pandas
!pip install numpy
!pip install matplotlib
!pip install glob
!pip install librosa

# importing libraries
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
from glob import glob
import librosa
import librosa.display
import IPython.display as ipd
import IPython
import zipfile

# Installing csv files

R1_mon= pd.read_csv("release1-monolog-csv.csv")
R2_con = pd.read_csv("release2-Conversation-csv.csv")
R2_Read= pd.read_csv("release2-Reading-csv.csv")
R2_mon= pd.read_csv("release2-Monologe-csv.csv")

#From ZIP file installing the audios

#Release 1 monologue
with zipfile.ZipFile("AllAudioWav.R1.zip", 'r') as
    zip_ref:
     zip_ref.extractall()

#Release 2 monologue
with zipfile.ZipFile("AllMonoWavAudioOnly.zip", 'r') as
     zip_ref:
```

```
    zip_ref.extractall()
#Release 2 Conversation
with zipfile.ZipFile("AllConvWavAudioOnly.zip", 'r') as
    zip_ref:
    zip_ref.extractall()
#Release 2 Reading
with zipfile.ZipFile("AllReadWavAudioOnly.zip", 'r') as
    zip_ref:
    zip_ref.extractall()

import os
#all the aduios
extracted_files=os.listdir()
wav_files = [file for file in extracted_files if file.
   endswith('.wav')]
print(wav_files)

#extracting the file names from each csv file(each
   situation)


col1_R1_mon=  R1_mon.iloc[:,0]
col1_R2_mon= R2_mon.iloc[:,0]
col1_R2_con= R2_con.iloc[:,0]
col1_R2_red= R2_Read.iloc[:,0]
#-------------------------------------------------------

#One audio example before cleaning

!pip install setuptools
import matplotlib.pyplot as plt

# listening  to one audio example:

import matplotlib.pyplot as plt

#plot the original audio before cleaning up
audio1,sr= librosa.load("M_0104_13y3m_1.wav")
```

```
plt.figure(figsize=(10, 5))
plt.plot(audio1)
plt.xlabel('Time (samples)')
plt.ylabel('Amplitude')
plt.title('Audio Waveform')
plt.show()

# summerinzing all the audios
# Function to calculate descriptive statistics for each
    audio file
def calculate_statistics(audio_files):
    statistics = []
    for audio_file in audio_files:
        try:
            # Load audio data
            y, sr = librosa.load(audio_file, sr=None)

            # Calculate statistics
            mean = np.mean(y)
            median = np.median(y)
            std = np.std(y)

            # Append statistics to list
            statistics.append({
                'Audio File': audio_file,
                'Mean': mean,
                'Median': median,
                'Standard Deviation': std
            })
        except Exception as e:
            print(f"Error loading {audio_file}: {str(e)
                }")
    return pd.DataFrame(statistics)

# Calculate descriptive statistics
statistics = calculate_statistics(wav_files)
```

```
# Plot boxplot graph
plt.figure(figsize=(8, 4))
plt.title('Summary Statistics of Audio Files')
plt.xlabel('Statistics')
plt.ylabel('Values')

# Create boxplot
boxplot = plt.boxplot(statistics[['Mean', 'Median', '
    Standard Deviation']].values, patch_artist=True)

# Add x-axis labels
plt.xticks([1, 2, 3], ['Mean', 'Median', 'Standard
    Deviation'])

# Fill boxplot with colors
colors = ['lightblue', 'lightgreen', 'lightyellow']
for patch, color in zip(boxplot['boxes'], colors):
    patch.set_facecolor(color)

# Add legend
plt.legend(boxplot['boxes'], ['Mean', 'Median', '
    Standard Deviation'], loc='upper right')

# Show plot
plt.tight_layout()
plt.show()

# Function to calculate descriptive statistics for a
    single audio file
def calculate_statistics(audio_file):
    try:
        # Load audio data
        y, sr = librosa.load(audio_file, sr=None)

        # Calculate statistics
        mean = np.mean(y)
        median = np.median(y)
        std = np.std(y)
```

```
            return mean, median, std
        except Exception as e:
            print(f"Error processing {audio_file}: {str(e)}
                ")
            return None, None, None

# Calculate descriptive statistics for all audio files
statistics = [calculate_statistics(audio_file) for
    audio_file in wav_files]

# Separate statistics
means, medians, stds = zip(*statistics)

# Plot descriptive statistics
plt.figure(figsize=(8, 5))
plt.title('Descriptive Statistics of Audio Files')
plt.ylabel('Values')

# Plot mean, median, and standard deviation for each
    audio file
plt.plot(range(len(wav_files)), means, label='Mean',
    marker='o')
plt.plot(range(len(wav_files)), medians, label='Median'
    , marker='o')
plt.plot(range(len(wav_files)), stds, label='Standard
    Deviation', marker='o')

# Hide x-axis tick labels
plt.xticks([])

# Show legend
plt.legend()

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

```
#cleaning the audios
pip install noisereduce

# Apply for one audio
import librosa
import numpy as np
import matplotlib.pyplot as plt
from noisereduce import reduce_noise

# Load the audio file
audio_path = "M_0104_13y3m_1.wav"
y, sr = librosa.load(audio_path, sr=None)  # Load with
    the original sampling rate

# Listen to the audio
import IPython.display as ipd
ipd.Audio(audio_path)
# Plot the spectrogram to visualize the audio
plt.figure(figsize=(10, 4))
librosa.display.specshow(librosa.amplitude_to_db(
    librosa.stft(y), ref=np.max), y_axis='log', x_axis='
    time', sr=sr)
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram (Before Noise Reduction)')
plt.show()

# Perform noise reduction
y_reduced = reduce_noise(y, sr)

# Plot the spectrogram after noise reduction
plt.figure(figsize=(10, 4))
librosa.display.specshow(librosa.amplitude_to_db(
    librosa.stft(y_reduced), ref=np.max), y_axis='log',
    x_axis='time', sr=sr)
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram (After Noise Reduction)')
```

```
#Generelizing the cleaning method to all the audios

from noisereduce import reduce_noise
from scipy.io import wavfile

def clean_audio(audio_filename):
    # Load the audio file
    y, sr = librosa.load(audio_filename, sr=None)  #
        Load with the original sampling rate

    # Perform noise reduction
    y_reduced = reduce_noise(y, sr)

    # Save the cleaned audio with the same filename as
        the original
    wavfile.write(audio_filename, sr, y_reduced)
    print("Cleaned audio saved to:", audio_filename)

# Get all WAV audio files in the current directory
audio_files = [file for file in os.listdir() if file.
    endswith(".wav")]

#
```

```
# Process each audio file
for audio_filename in audio_files:
    clean_audio(audio_filename)

# summerinzing all the audios after cleaning
# Function to calculate descriptive statistics for each
    audio file
def calculate_statistics(audio_files):
    statistics = []
    for audio_file in audio_files:
        try:
```

```
            # Load audio data
            y, sr = librosa.load(audio_file, sr=None)

            # Calculate statistics
            mean = np.mean(y)
            median = np.median(y)
            std = np.std(y)

            # Append statistics to list
            statistics.append({
                'Audio File': audio_file,
                'Mean': mean,
                'Median': median,
                'Standard Deviation': std
            })
        except Exception as e:
            print(f"Error loading {audio_file}: {str(e)
                }")
    return pd.DataFrame(statistics)

# Calculate descriptive statistics
statistics = calculate_statistics(wav_files)
# Plot boxplot graph
plt.figure(figsize=(8, 4))
plt.title('Summary Statistics of Audio Files')
plt.xlabel('Statistics')
plt.ylabel('Values')
# Create boxplot
boxplot = plt.boxplot(statistics[['Mean', 'Median', '
    Standard Deviation']].values, patch_artist=True)

# Add x-axis labels
plt.xticks([1, 2, 3], ['Mean', 'Median', 'Standard
    Deviation'])

# Fill boxplot with colors
colors = ['lightblue', 'lightgreen', 'lightyellow']
for patch, color in zip(boxplot['boxes'], colors):
```

```
    patch.set_facecolor(color)

# Add legend
plt.legend(boxplot['boxes'], ['Mean', 'Median', '
    Standard Deviation'], loc='upper right')

# Show plot
plt.tight_layout()
plt.show()

# Function to calculate descriptive statistics for a
    single audio file after cleaning
def calculate_statistics(audio_file):
    try:
        # Load audio data
        y, sr = librosa.load(audio_file, sr=None)

        # Calculate statistics
        mean = np.mean(y)
        median = np.median(y)
        std = np.std(y)

        return mean, median, std
    except Exception as e:
        print(f"Error processing {audio_file}: {str(e)}
            ")
        return None, None, None

# Calculate descriptive statistics for all audio files
statistics = [calculate_statistics(audio_file) for
    audio_file in wav_files]

# Separate statistics
means, medians, stds = zip(*statistics)

# Plot descriptive statistics
plt.figure(figsize=(10, 6))
plt.title('Descriptive Statistics of Audio Files')
```

```
plt.ylabel('Values')

# Plot mean, median, and standard deviation for each
    audio file
plt.plot(range(len(wav_files)), means, label='Mean',
    marker='o')
plt.plot(range(len(wav_files)), medians, label='Median'
    , marker='o')
plt.plot(range(len(wav_files)), stds, label='Standard-
    Deviation', marker='o')

# Hide x-axis tick labels
plt.xticks([])

# Show legend
plt.legend()

# Show plot
plt.tight_layout()
plt.show()

#-----------------------------------------------------------
#In stuttering analysis, we often divide the audio
    recordings into smaller segments, which can
    correspond to individual words, phrases, or
    sentences. Each of these segments is then analyzed
    separately to assess stuttering severity or detect
    stuttering events.

pip install praat-parselmouth
import librosa
import numpy as np
import parselmouth

# Function to calculate pitch using Praat
def calculate_pitch(segment, sampling_rate):
    sound = parselmouth.Sound(segment,
        sampling_frequency=sampling_rate)
```

```
    pitch = sound.to_pitch()
    mean_pitch = pitch.selected_array['frequency'].mean
        ()
    return mean_pitch

# Function to calculate intensity
def calculate_intensity(segment):
    energy = np.square(segment)
    rms_amplitude = np.sqrt(np.mean(energy))
    std_deviation = np.std(segment)
    return rms_amplitude

# Function to calculate disfluency duration
def calculate_disfluency_duration(segment,
    sampling_rate, pause_threshold=0.05):
    energy_env = np.abs(librosa.core.stft(segment))
    energy_per_frame = np.mean(energy_env, axis=0)
    pauses = np.where(energy_per_frame <
        pause_threshold)[0]
    pause_times = librosa.frames_to_time(pauses, sr=
        sampling_rate)
    disfluency_duration = np.sum(np.diff(pause_times))
    return disfluency_duration

import librosa
import numpy as np

def calculate_spectral_feature_scalar(segment,
    sampling_rate):
    # Compute short-time Fourier transform (STFT)
    stft = librosa.stft(segment)

    # Compute power spectrogram
    power_spectrogram = np.abs(stft)**2

    # Compute Mel-frequency cepstral coefficients (
        MFCCs)
    mfccs = librosa.feature.mfcc(S=librosa.power_to_db(
```

```
        power_spectrogram),
                                            sr=sampling_rate,
                                            n_mfcc=13)

    # Calculate the mean of MFCCs across all
        coefficients
    mfccs_mean = np.mean(mfccs)

    return mfccs_mean

# Function to assess severity of stuttering based on
    extracted features
def assess_stuttering_severity(pitch, intensity,
    disfluency_duration, spectral_feature_scalar):
    # Define weights for each feature (you can adjust
        these based on your analysis)
    pitch_weight = 0.25
    intensity_weight = 0.25
    disfluency_duration_weight = 0.25
    spectral_feature_weight = 0.25

    # Combine features using weighted sum
    combined_score = (pitch * pitch_weight +
                      intensity * intensity_weight +
                      disfluency_duration *
                          disfluency_duration_weight +
                      spectral_feature_scalar *
                          spectral_feature_weight)

    # Normalize the combined score (optional)
    min_score = min(pitch, intensity,
        disfluency_duration, spectral_feature_scalar)
    max_score = max(pitch, intensity,
        disfluency_duration, spectral_feature_scalar)
    normalized_score = (combined_score - min_score) / (
        max_score - min_score)

    return normalized_score
```

```
#-----------------------------------------------------------

#Calculating audio scores

def calculate_audio_stuttering_score(audio_file,
    max_segment_duration=5):
     try:
          # Load the audio file
          y, sr = librosa.load(audio_file, sr=None)

          # Check if the audio was loaded successfully
          if y is None:
               print("Error: Unable to load audio file.")
               return None

          # Calculate the total duration of the audio
          total_duration = librosa.get_duration(y=y, sr=
              sr)

          # Calculate the number of segments based on the
               maximum segment duration
          num_segments = int(np.ceil(total_duration /
              max_segment_duration))

          # Placeholder list to store feature scores for
              each segment
          segment_scores = []

          # Iterate over each segment
          for i in range(num_segments):
               # Calculate the start and end time of the
                   segment
               segment_start = i * max_segment_duration
               segment_end = min((i + 1) *
                   max_segment_duration, total_duration)

               # Extract the segment from the audio
```

```
            segment = y[int(segment_start * sr):int(
                segment_end * sr)]

    # Calculate pitch
            pitch = calculate_pitch(segment, sr)
            # Calculate intensity
            intensity = calculate_intensity(segment)

            # Calculate disfluency duration
            disfluency_duration =
                calculate_disfluency_duration(segment,
                sr)

            # Calculate spectral feature scalar
            spectral_feature_scalar =
                calculate_spectral_feature_scalar(
                segment, sr)

            # Define weights for each feature
            weights = [0.25, 0.25, 0.25, 0.25]  # Equal
                weights for demonstration purposes

            # Aggregate the features into a segment
                score using weighted combination
            normalized_segment_score =
                assess_stuttering_severity(pitch,
                intensity, disfluency_duration,
                spectral_feature_scalar)

            # Append the segment score to the list
            segment_scores.append(
                normalized_segment_score)

        # Calculate the overall score for the audio
            file
        overall_score = np.mean(segment_scores)

        return overall_score
```

```
    except Exception as e:
        return None

#-------------*-------------
audio_scores = []
for audio_file in wav_files:
    audio_score = calculate_audio_stuttering_score(
        audio_file)
    audio_scores.append(audio_score)



# Print or use the scores as needed
print("Stuttering Severity Scores:", audio_scores)
#-------------*-------------


# Plot audio scores
plt.figure(figsize=(10, 6))
plt.title('Audio Scores for Each Audio File')
plt.xlabel('Audio Files')
plt.ylabel('Audio Score')

# Plot the audio scores
plt.plot(wav_files, audio_scores, label='Audio Score',
    marker='o')

# Set x-ticks to audio file names for clarity
plt.xticks(rotation=45)

# Show legend
plt.legend()

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

```
#-------------------------------------------------------------------
#Creating lists of each feaature score in all the
    audios

# Function to extract features from an audio file
def extract_features(audio_file, max_segment_duration
    =5):
    try:
        y, sr = librosa.load(audio_file, sr=None)
        total_duration = librosa.get_duration(y=y, sr=
            sr)
        num_segments = int(np.ceil(total_duration /
            max_segment_duration))

        pitch_values = []
        intensity_values = []
        disfluency_durations = []
        spectral_feature_scalars = []

        for i in range(num_segments):
            segment_start = i * max_segment_duration
            segment_end = min((i + 1) *
                max_segment_duration, total_duration)
            segment = y[int(segment_start * sr):int(
                segment_end * sr)]

            pitch_values.append(calculate_pitch(segment
                , sr))
            intensity_values.append(calculate_intensity
                (segment))
            disfluency_durations.append(
                calculate_disfluency_duration(segment,
                sr))
            spectral_feature_scalars.append(
                calculate_spectral_feature_scalar(
                segment, sr))

        return {
```

```
            'pitch': pitch_values,
            'intensity': intensity_values,
            'disfluency_duration': disfluency_durations
                ,
            'spectral_feature_scalar':
                spectral_feature_scalars
        }

    except Exception as e:

        return None
def calculate_mean_pitch(audio_files,
    max_segment_duration=5):
    mean_pitches = []
    for audio_file in audio_files:
        try:
            features = extract_features(audio_file,
                max_segment_duration)
            if features:
                mean_pitches.append(np.mean(features['
                    pitch']))
            else:
                mean_pitches.append(None)  # Append
                    None if features is None
        except Exception as e:
            # Do nothing here if an exception occurs
            mean_pitches.append(None)  # Append None if
                an exception occurs
    return mean_pitches

# Function to calculate mean intensity for all audio
    files
def calculate_mean_intensity(audio_files,
    max_segment_duration=5):
    mean_intensities = []
    for audio_file in audio_files:
        try:
            features = extract_features(audio_file,
```

```
                max_segment_duration)
            if features:
                mean_intensities.append(np.mean(
                    features['intensity']))
            else:
                mean_intensities.append(None)  # Append
                    None if features is None
        except Exception as e:
            # Do nothing here if an exception occurs
            mean_intensities.append(None)  # Append
                None if an exception occurs
    return mean_intensities
# Function to calculate mean disfluency duration for
    all audio files
def calculate_mean_disfluency_duration(audio_files,
    max_segment_duration=5):
    mean_disfluency_durations = []
    for audio_file in audio_files:
        try:
            features = extract_features(audio_file,
                max_segment_duration)
            if features:
                mean_disfluency_durations.append(np.
                    mean(features['disfluency_duration'
                    ]))
            else:
                mean_disfluency_durations.append(None)
                    # Append None if features is None
        except Exception as e:
            # Do nothing here if an exception occurs
            mean_disfluency_durations.append(None)  #
                Append None if an exception occurs
    return mean_disfluency_durations

# Function to calculate mean spectral feature scalar
    for all audio files
def calculate_mean_spectral_feature_scalar(audio_files,
    max_segment_duration=5):
```

```
    mean_spectral_feature_scalars = []
    for audio_file in audio_files:
        try:
            features = extract_features(audio_file,
                max_segment_duration)
            if features:
                mean_spectral_feature_scalars.append(np
                    .mean(features['
                    spectral_feature_scalar']))
            else:
                mean_spectral_feature_scalars.append(
                    None)  # Append None if features is
                    None
        except Exception as e:
            # Do nothing here if an exception occurs
            mean_spectral_feature_scalars.append(None)
                # Append None if an exception occurs
    return mean_spectral_feature_scalars
# List of WAV audio files

# Calculate feature scores for all audio files
mean_pitches = calculate_mean_pitch(wav_files)
mean_intensities = calculate_mean_intensity(wav_files)
mean_disfluency_durations =
    calculate_mean_disfluency_duration(wav_files)
mean_spectral_feature_scalars =
    calculate_mean_spectral_feature_scalar(wav_files)

# Print the scores
print("Mean Pitch Scores:", mean_pitches)
print("Mean Intensity Scores:", mean_intensities)
print("Mean Disfluency Duration Scores:",
    mean_disfluency_durations)
print("Mean Spectral Feature Scalar Scores:",
    mean_spectral_feature_scalars)

#
```

```
#Plotting the features

fig, axs = plt.subplots(2, 2, figsize=(7, 5))

# Plot mean pitch scores
axs[0, 0].plot(mean_pitches, marker='o',markersize=5,
    linewidth=1)
axs[0, 0].set_title('Mean Pitch Scores')
axs[0, 0].set_ylabel('Mean Pitch')

# Plot mean intensity scores
axs[0, 1].plot(mean_intensities, marker='o',markersize
    =4, linewidth=1)
axs[0, 1].set_title('Mean Intensity Scores')
axs[0, 1].set_ylabel('Mean Intensity')

# Plot mean disfluency duration scores
axs[1, 0].plot(mean_disfluency_durations, marker='o',
    markersize=4, linewidth=1)
axs[1, 0].set_title('Mean Disfluency Duration Scores')
axs[1, 0].set_ylabel('Mean Disfluency Duration')

# Plot mean spectral feature scalar scores
axs[1, 1].plot(mean_spectral_feature_scalars, marker='o
    ',markersize=4, linewidth=1)
axs[1, 1].set_title('Mean Spectral Feature Scalar
    Scores')
axs[1, 1].set_ylabel('Mean Spectral Feature Scalar')

# Adjust layout
plt.tight_layout()

# Show plot
plt.show()

#-----------------------------------------------------
##plot the scores
```

```
import matplotlib.pyplot as plt

# Assuming you have already calculated audio_scores

# Plotting the audio scores
plt.figure(figsize=(10, 6))
plt.plot(audio_scores, marker='o', linestyle='-')
plt.title('Stuttering Severity Scores for Audio Files')
plt.xlabel('Audio File Index')
plt.ylabel('Stuttering Severity Score')
plt.grid(True)
plt.tight_layout()
plt.show()


#-----------------------------------------
#Scatter plot of audio scores in different situations

# Plot scores for Release 2 Monologue audio files
scores_R2_mon = []
plt.figure(figsize=(8, 3))
for audio_file in col1_R2_mon:
    try:
        index = wav_files.index(audio_file + ".wav")
        scores_R2_mon.append(audio_scores[index])
    except ValueError:
        pass  # If the audio file is not found, skip it

# Plot scores for Release 2 Conversation audio files
scores_R2_con = []
for audio_file in col1_R2_con:
    try:
        index = wav_files.index(audio_file + ".wav")
        scores_R2_con.append(audio_scores[index])
    except ValueError:
        pass  # If the audio file is not found, skip it

# Plot scores for Release 2 Reading audio files
```

```
scores_R2_red = []
for audio_file in col1_R2_red:
    try:
        index = wav_files.index(audio_file + ".wav")
        scores_R2_red.append(audio_scores[index])
    except ValueError:
        pass  # If the audio file is not found, skip it
# Plot scores for Release 1 Monologue audio files
scores_R1_mon = []
for audio_file in col1_R1_mon:
    try:
        index = wav_files.index(audio_file + ".wav")
        scores_R1_mon.append(audio_scores[index])
    except ValueError:
        pass  # If the audio file is not found, skip it

# Plot scores as points for Release 1 Monologue
plt.scatter(range(len(scores_R1_mon)), scores_R1_mon,
    color='blue', marker='o', label='Release 1 Monologue
    ')

# Plot scores as points for Release 2 Monologue
plt.scatter(range(len(scores_R2_mon)), scores_R2_mon,
    color='blue', marker='o', label='Release 2 Monologue
    ')

# Plot scores as points for Release 2 Conversation
plt.scatter(range(len(scores_R2_con)), scores_R2_con,
    color='red', marker='o', label='Release 2
    Conversation')

# Plot scores as points for Release 2 Reading
plt.scatter(range(len(scores_R2_red)), scores_R2_red,
    color='green', marker='o', label='Release 2 Reading'
    )

plt.xlabel('Audio Files')
plt.ylabel('Scores')
```

```
plt.title('Scores for Release 1 and Release 2 Audio
    Files')
plt.legend()  # Show legend
plt.xticks([])  # Remove x-axis ticks
plt.show()

#
    ------------------------------------------------------------


#Anova In orde to see if there is significant
    difference between the situations
from scipy.stats import f_oneway
import numpy as np

# Filter out None values from each group
scores_R1_mon_filtered = [score for score in
    scores_R1_mon if score is not None]
scores_R2_mon_filtered = [score for score in
    scores_R2_mon if score is not None]
scores_R2_con_filtered = [score for score in
    scores_R2_con if score is not None]
scores_R2_red_filtered = [score for score in
    scores_R2_red if score is not None]

combined_scores_mon = scores_R1_mon_filtered +
    scores_R2_mon_filtered
# Perform ANOVA
f_statistic, p_value = f_oneway(combined_scores_mon,
    scores_R2_con_filtered, scores_R2_red_filtered)

# Print results
print("ANOVA Results:")
print("F-statistic:", f_statistic)
print("p-value:", p_value)

# Interpret results
alpha = 0.05
if p_value < alpha:
```

```
    print("Reject the null hypothesis. There are
        significant differences in stuttering severity
        scores between groups.")
else:
    print("Fail to reject the null hypothesis. There
        are no significant differences in stuttering
        severity scores between groups.")

!pip install statsmodels
from statsmodels.stats.multicomp import
    pairwise_tukeyhsd
import numpy as np

# Perform Tukey's HSD post hoc test
if p_value < alpha:
    data = np.concatenate([scores_R2_mon_filtered,
        scores_R2_con_filtered, scores_R2_red_filtered])
    # Remove the 'R2_' prefix from the labels
    labels = ['mon'] * len(scores_R2_mon_filtered) + ['
        con'] * len(scores_R2_con_filtered) + ['red'] *
        len(scores_R2_red_filtered)
    tukey_results = pairwise_tukeyhsd(data, labels,
        alpha=0.05)

    # Print Tukey's HSD results
    print("\nTukey's HSD Results:")
    print(tukey_results)

#-----------------------------------------------------
# Before starting with the models in machine learning
    we have to:
#1 orginize the data(all the files in one file)
#2 use methods to deal with the small number of
    observations that we have

#Orginizing the data

# Add the name of situation to each DataFrame
```

```
R1_mon['situation'] = 'monologue'
R2_mon['situation'] = 'monologue'
R2_con['situation'] = 'conversation'
R2_Read['situation'] = 'reading'

# Create a dictionary to map filenames (without .wav)
    to their scores
file_scores = {filename[:-4]: score for filename, score
    in zip(wav_files, audio_scores)}

# Function to get the score for a filename (without .
    wav), returns None if filename not found
def get_score(filename):
    return file_scores.get(filename)

# Adding the audio scores that are relevant to each
    file
R1_mon["audio_score"] = [get_score(x) for x in
    col1_R1_mon]
R2_mon["audio_score"] = [get_score(x) for x in
    col1_R2_mon]
R2_con["audio_score"] = [get_score(x) for x in
    col1_R2_con]
R2_Read["audio_score"] = [get_score(x) for x in
    col1_R2_red]

#---------*---------

# Create dictionaries for each feature
pitch_scores_dict = {}
intensity_scores_dict = {}
disfluency_duration_dict = {}
spectral_feature_scalar_dict = {}

# Populate dictionaries with scores for each
    observation
for filename, score in zip(wav_files, mean_pitches):
    filename_key = filename[:-4]  # Removing the .wav
```

```
        extension
    pitch_scores_dict[filename_key] = score

for filename, score in zip(wav_files, mean_intensities)
    :
    filename_key = filename[:-4]  # Removing the .wav
        extension
    intensity_scores_dict[filename_key] = score

for filename, score in zip(wav_files,
    mean_disfluency_durations):
    filename_key = filename[:-4]  # Removing the .wav
        extension
    disfluency_duration_dict[filename_key] = score

for filename, score in zip(wav_files,
    mean_spectral_feature_scalars):
    filename_key = filename[:-4]  # Removing the .wav
        extension
    spectral_feature_scalar_dict[filename_key] = score
# Define functions to get scores for each feature
def get_pitch_score(filename):
    return pitch_scores_dict.get(filename)

def get_intensity_score(filename):
    return intensity_scores_dict.get(filename)

def get_disfluency_duration(filename):
    return disfluency_duration_dict.get(filename)

def get_spectral_feature_scalar(filename):
    return spectral_feature_scalar_dict.get(filename)

# Add scores for each feature to DataFrames
R1_mon["pitch_score"] = [get_pitch_score(x) for x in
    col1_R1_mon]
R1_mon["intensity_score"] = [get_intensity_score(x) for
    x in col1_R1_mon]
```

```
R1_mon["disfluency_duration"] = [
    get_disfluency_duration(x) for x in col1_R1_mon]
R1_mon["spectral_feature_scalar"] = [
    get_spectral_feature_scalar(x) for x in col1_R1_mon]

R2_mon["pitch_score"] = [get_pitch_score(x) for x in
    col1_R2_mon]
R2_mon["intensity_score"] = [get_intensity_score(x) for
     x in col1_R2_mon]
R2_mon["disfluency_duration"] = [
    get_disfluency_duration(x) for x in col1_R2_mon]
R2_mon["spectral_feature_scalar"] = [
    get_spectral_feature_scalar(x) for x in col1_R2_mon]

R2_con["pitch_score"] = [get_pitch_score(x) for x in
    col1_R2_con]
R2_con["intensity_score"] = [get_intensity_score(x) for
     x in col1_R2_con]
R2_con["disfluency_duration"] = [
    get_disfluency_duration(x) for x in col1_R2_con]
R2_con["spectral_feature_scalar"] = [
    get_spectral_feature_scalar(x) for x in col1_R2_con]

R2_Read["pitch_score"] = [get_pitch_score(x) for x in
    col1_R2_red]
R2_Read["intensity_score"] = [get_intensity_score(x)
    for x in col1_R2_red]
R2_Read["disfluency_duration"] = [
    get_disfluency_duration(x) for x in col1_R2_red]
R2_Read["spectral_feature_scalar"] = [
    get_spectral_feature_scalar(x) for x in col1_R2_red]

#-------*----------
def format_words(words):
    formatted_words = []
    for word in words:
        # Convert to lowercase, remove leading and
            trailing spaces, and replace inner spaces
```

```
            with underscores
        formatted_word = word.strip().lower().replace("
            ", "_")
        formatted_words.append(formatted_word)
    return formatted_words



# Now, all DataFrames should have consistent column
    names

col_new_R1_mon = format_words(R1_mon)
col_new_R1_mon = ["age_at_onset" if x == "age_of_onset"
    else x for x in col_new_R1_mon]  #chaning
    age_of_onset to age_at_onset (:

col_new_R2_mon = format_words(R2_mon)
col_new_R2_con = format_words(R2_con)
col_new_R2_Read = format_words(R2_Read)

R1_mon.columns= col_new_R1_mon
R2_mon.columns= col_new_R2_mon
R2_con.columns= col_new_R2_con
R2_Read.columns= col_new_R2_Read
#
```

```
#Final data

# Combine the DataFrames row-wise
final_data = pd.concat([R1_mon, R2_mon, R2_con, R2_Read
    ], ignore_index=True)

# Print the head of the final DataFrame to verify the
    result
print(final_data.head())
```

```
final_data_reset= final_data # we use it to reset (
    instead of running the first code again)

all_var= final_data.columns.tolist()
print(all_var)

print(final_data['gender'].unique())
print(final_data["situation"].unique())
print(final_data['handedness'].unique())
print(final_data['family_history'].unique())
print(final_data['where_recorded'].unique())
print(final_data['recording_conditions'].unique())
print(final_data['hearing_problems'].unique())
print(final_data['language_problems'].unique())


# Ensure no leading/trailing spaces
final_data['gender'] = final_data['gender'].str.strip()

# Ensure consistent data types
final_data['gender'] = final_data['gender'].astype(str)

# Replace 'F' with 'Female' and 'M' with 'Male'
final_data['gender'] = final_data['gender'].replace({'F
    ': 'Female', 'M': 'Male'})

# Check unique values to confirm the change
print(final_data['gender'].unique())


# Ensure no leading/trailing spaces
final_data['type_of_therapy'] = final_data['
    type_of_therapy'].str.strip()

# Ensure consistent data types
final_data['type_of_therapy'] = final_data['
    type_of_therapy'].astype(str)
```

```
# Replace values in the 'type_of_therapy' column
final_data['type_of_therapy'] = final_data['
    type_of_therapy'].replace({'F': 'Family', 'H': '
    Holistic'})

# Check unique values to confirm the change
print(final_data['type_of_therapy'].unique())

print(final_data['family_history'].unique())

#-----------------*-----------------

#Model selection

#number of all the observations
num_observations = len(final_data)
print("Number of all observations:", num_observations)


# Counting NA in each variable
na_counts = final_data.isna().sum()
print("\nNA counts in each variable:")
print(na_counts) #missing values in each variable

#The 'not known' values in the family history and
    handedness is concidered categorical values because
    they mentiones in the article that it takes 3 values
     , So i will reconvert them to 'not known' and
    convert the variable to categorical .
and for age on onset , since it numerical, i will fill
    the 'unknown' values with the mean (Imputation).
#-----------------------------------------------

###fill na with unknown in the handedness and family
    history
import pandas as pd

# Fill NaN values with 'unknown' in the 'handedness'
```

```
    column
final_data['handedness'] = final_data['handedness'].
    fillna('not-known')

# Fill NaN values with 'unknown' in the 'family_history
    ' column
final_data['family_history'] = final_data['
    family_history'].fillna('not-known')

# Check unique values in the 'family_history' column
print(final_data['family_history'].unique())

#######


###Imputation for 'age_at_onset ' :
# Convert 'not known' to NaN in the 'age_at_onset'
    column
final_data['age_at_onset'] = final_data['age_at_onset'
    ].replace('not-known', pd.NA)

# Convert the column to numeric type
final_data['age_at_onset'] = pd.to_numeric(final_data['
    age_at_onset'])

# Fill NaN values with mean in the 'age_at_onset'
    column
mean_value = final_data['age_at_onset'].mean()
final_data['age_at_onset'] = final_data['age_at_onset'
    ].fillna(mean_value)

# Check unique values in the 'age_at_onset' column
print(final_data['age_at_onset'].unique())




#######
```

```
# Counting NA in each variable
na_counts = final_data.isna().sum()
print("\nNA counts in each variable:")
print(na_counts) #missing values in each variable

#-----------------------------------------------------------
#print all the categories in order to do mapping when
    converting to factors
print(final_data['family_history'].unique())
print(final_data['handedness'].unique())
print(final_data['gender'].unique())
print(final_data['where_recorded'].unique())
print(final_data[ 'recording_conditions'].unique())
print(final_data['type_of_therapy'].unique())
print(final_data[ 'hearing_problems'].unique())
print(final_data['language_problems'].unique())
print(final_data['sen'].unique())
print(final_data[ 'orthographic'].unique())
print(final_data['phonetic'].unique())
print(final_data['time_aligned'].unique())
print(final_data[ 'situation' ].unique())
print(final_data[ 'phl'].unique())


# deleting the variables that include high rate of
    missing values
#We put 0.4 missing of the observations as high rate
    because we don't have a lot of observations

threshold = 0.4  # Example threshold, adjust as needed
variables_to_keep = na_counts[na_counts /
    num_observations <= threshold].index
variables_to_delete = na_counts[na_counts /
    num_observations > threshold].index
final_data_filtered = final_data[variables_to_keep].
    copy()

print("\nVariables after filtering:")
```

```
print(final_data_filtered.head())

print("\nVariables deleted due to having too many
    missing values:")
print(variables_to_delete)

# Deleting observations with NA
final_data_filtered= final_data_filtered.dropna()
print(final_data_filtered)

num_rows, num_cols = final_data_filtered.shape

print("Number of rows:", num_rows)
print("Number of columns:", num_cols)

# Count missing values in each variable
missing_counts = final_data_filtered.isna().sum()

# Print the missing value counts
print("Missing value counts for each variable:")
print(missing_counts)

#-------------------------------------------------------
##convert categorical variables to factor:
#Converting categorical variables to factor

# List of columns to convert to factors
cols_to_factorize = ['gender', 'handedness', '
    family_history', 'where_recorded',
                     'recording_conditions', '
                        type_of_therapy', '
                        hearing_problems',
                     'language_problems', 'sen', '
                        orthographic', 'phonetic',
                     'time_aligned',
                     'situation' , 'phl']

# Convert selected columns to factors
```

```
final_data_filtered[cols_to_factorize] = final_data[
    cols_to_factorize].apply(lambda x: pd.factorize(x)
    [0])

print(final_data_filtered["situation"].unique())

# Reset index
final_data_filtered.reset_index(inplace=True)

# Drop 'file_name' and 'index' columns if they exist
if 'file_name' in final_data_filtered.columns:
    final_data_filtered.drop(columns='file_name',
        inplace=True)
if 'index' in final_data_filtered.columns:
    final_data_filtered.drop(columns='index', inplace=
        True)
    # Drop 'level_0' column
if 'level_0' in final_data_filtered.columns:
    final_data_filtered.drop(columns='level_0', inplace
        =True)

#-----------------------------------------------------
#Machine learning
#way A

#Standardizing the data

final_data_filtered_stan = final_data_filtered.copy()
# Categorical columns as specified
categorical_columns = [
    'gender', 'handedness', 'family_history', '
        where_recorded',
    'recording_conditions', 'type_of_therapy', '
        hearing_problems',
    'language_problems', 'sen', 'orthographic', '
        phonetic',
    'time_aligned', 'situation', 'phl'
]
```

```
# Identify numeric columns by excluding the categorical
    columns
numeric_columns = final_data_filtered_stan.columns.
    difference(categorical_columns).tolist()

# Standardize the numeric columns
scaler = StandardScaler()
final_data_filtered_stan[numeric_columns] = scaler.
    fit_transform(final_data_filtered_stan[
    numeric_columns])

# Combine the standardized numeric columns with the
    categorical columns
final_data_filtered_stan = pd.concat([
    final_data_filtered_stan[categorical_columns],
    final_data_filtered_stan[numeric_columns]], axis=1)

print(final_data_filtered_stan)

#Dividing the data into train and test

from sklearn.model_selection import train_test_split

# Extract features and labels
X = final_data_filtered_stan.drop(columns=['situation'
    ]) # Features
y = final_data_filtered_stan['situation'] # Labels

X_train, X_test, y_train, y_test = train_test_split(X,
    y, test_size=0.3, random_state=42)

#-----------------------------------------------------
#SVM without cross validation
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,
```

```
    mean_squared_error

# Train SVM classifier
svm_classifier = SVC(kernel='linear')
svm_classifier.fit(X_train, y_train)

# Predictions
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy_SVM_way1 = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy_SVM_way1)


#confusion matrix
from sklearn.metrics import confusion_matrix
#general function that We will use to the other methods
def plot_confusion_matrix(y_true, y_pred, classes,
    title):
     cm = confusion_matrix(y_true, y_pred)
     plt.figure(figsize=(8, 6))
     plt.imshow(cm, interpolation='nearest', cmap=plt.cm
         .Blues)
     plt.title(title)
     tick_marks = np.arange(len(classes))
     plt.xticks(tick_marks, classes)
     plt.yticks(tick_marks, classes)
     plt.xlabel('Predicted label')
     plt.ylabel('True label')

     # Add text annotations for each cell
     for i in range(len(classes)):
         for j in range(len(classes)):
             plt.text(j, i, format(cm[i, j], 'd'),
                      horizontalalignment="center",
                      color="white" if cm[i, j] > cm.max
                          () / 2 else "black")
```

```
    plt.colorbar()
    plt.show()

custom_classes = ['monologue', 'conversation', 'reading
    ']
plot_confusion_matrix(y_test, y_pred, custom_classes, '
    Confusion Matrix - SVM Classifier')

#-------------------------------------------------------
# SVM with cross validation

# we got bad results this may improve by using cross
    validation(because we have only 314 observations) -
     we used 5 folds
from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC

svm_classifier = SVC(kernel='linear')

# 5 folds
cv_scores = cross_val_score(svm_classifier, X, y, cv=5)

# the scores of cross-validation:
print("Cross-Validation Scores:", cv_scores)

#  mean and standard deviation of cross-validation
    scores
mean_cv_score = cv_scores.mean()
std_cv_score = cv_scores.std()
print("Mean Cross-Validation Score:", mean_cv_score)
print("Standard Deviation of Cross-Validation Scores:",
    std_cv_score)
# :
#-------------------------------------------------------
#Random forest classifier
##Random Forest Classifier :
from sklearn.ensemble import RandomForestClassifier
rf_classifier = RandomForestClassifier(n_estimators
```

```
    =100, random_state=42)
# Train Random Forest Classifier
rf_classifier.fit(X_train, y_train)

# Predictions
y_pred_rf = rf_classifier.predict(X_test)

# check  accuracy
accuracy_rf_way1 = accuracy_score(y_test, y_pred_rf)
print("Random Forest Accuracy:", accuracy_rf_way1)

# Plot confusion matrix for Random Forest Classifier
plot_confusion_matrix(y_test, y_pred_rf, custom_classes
    , 'Confusion Matrix - Random Forest Classifier')

#-------------------------------------------------------
#KNN
## k-Nearest Neighbors Classifier

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score,
    mean_squared_error

# Train k-NN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=5)
knn_classifier.fit(X_train, y_train)

# Predictions
y_pred_knn = knn_classifier.predict(X_test)

# Calculate accuracy
accuracy_knn_way1 = accuracy_score(y_test, y_pred_knn)
print("k-NN Classifier Accuracy:", accuracy_knn_way1)

plot_confusion_matrix(y_test, y_pred_knn,
    custom_classes, 'Confusion Matrix - k-NN Classifier'
    )
```

```
#-------------------------------------------------------------
#Logistic Regression Classifier
#Logistic Regression Classifier

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,
    mean_squared_error

# Train logistic regression classifier with adjusted
    solver and increased max_iter
logistic_classifier = LogisticRegression(multi_class='
    multinomial', solver='saga', max_iter=2000)
logistic_classifier.fit(X_train, y_train)

# Predictions
y_pred_logistic = logistic_classifier.predict(X_test)

# Calculate accuracy
accuracy_logistic_way1 = accuracy_score(y_test,
    y_pred_logistic)
print("Logistic Regression Classifier Accuracy:",
    accuracy_logistic_way1)

plot_confusion_matrix(y_test, y_pred_logistic,
    custom_classes, 'Confusion Matrix - Logistic
    Regression Classifier')

#
    ------------------------------------------------------------


#Way2

# Changing the situation variable to have 2 categories
# Combining "Reading, Monologue" in the same category
    and conversation in the other
final_data_filtered_way2 = final_data_filtered.copy()
```

```
# Map 'monologue' and 'reading' to 0, and 'conversation
    ' to 1
final_data_filtered_way2['situation'] =
    final_data_filtered_way2['situation'].replace({0: 0,
     1: 1, 2: 0})

# Confirm the changes
print(final_data_filtered_way2['situation'].unique())

# To check that final_data_filtered is unchanged
print(final_data_filtered['situation'].unique())


#Work on the standarized data
final_data_filtered_way2_stan =
    final_data_filtered_stan.copy()

# Map 'monologue' and 'reading' to 0, and 'conversation
    ' to 1
final_data_filtered_way2_stan['situation'] =
    final_data_filtered_way2_stan['situation'].replace
    ({0: 0, 1: 1, 2: 0})

# Confirm the changes
print(final_data_filtered_way2_stan['situation'].unique
    ())

# To check that final_data_filtered is unchanged
print(final_data_filtered_stan['situation'].unique())

#Dividing the data into train and test

from sklearn.model_selection import train_test_split

# Extract features and labels
X = final_data_filtered_way2_stan.drop(columns=['
    situation'])  # Features
y = final_data_filtered_way2_stan['situation']  #
```

```
    Labels

X_train_way2, X_test_way2, y_train_way2, y_test_way2 =
    train_test_split(X, y, test_size=0.3, random_state
    =42)
#-----------------------------------------------------
#SVM without cross validation
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,
    mean_squared_error

# Train SVM classifier
svm_classifier = SVC(kernel='linear')  # You can try
    different kernels like 'rbf' or 'poly'
svm_classifier.fit(X_train_way2, y_train_way2)

# Predictions
y_pred_way2 = svm_classifier.predict(X_test_way2)

# Calculate accuracy
accuracy_SVM_way2 = accuracy_score(y_test_way2,
    y_pred_way2)
print("Accuracy:", accuracy_SVM_way2 )

# Calculate mean squared error (MSE)
mse_SVM_way2 = mean_squared_error(y_test_way2,
    y_pred_way2)
print("Mean Squared Error (MSE):", mse_SVM_way2 )

from sklearn.metrics import confusion_matrix
# Function to plot simplified confusion matrix for
    binary classification with custom class labels
def plot_binary_confusion_matrix(y_true, y_pred,
    classes, title):
     cm = confusion_matrix(y_true, y_pred)
     tn, fp, fn, tp = cm[0, 0], cm[0, 1], cm[1, 0], cm
```

```
        [1, 1]  # Extract values from confusion matrix

    plt.figure(figsize=(6, 6))  # Adjust the figure
        size here
    plt.imshow([[tp, fp], [fn, tn]], interpolation='
        nearest', cmap=plt.cm.Blues)
    plt.title(title)
    tick_marks = np.arange(2)
    plt.xticks(tick_marks, classes)
    plt.yticks(tick_marks, classes)
    plt.xlabel('Predicted label')
    plt.ylabel('True label')

    # Add text annotations for each cell
    plt.text(0, 0, tp, horizontalalignment="center",
        color="white")
    plt.text(1, 0, fp, horizontalalignment="center",
        color="black")
    plt.text(0, 1, fn, horizontalalignment="center",
        color="black")
    plt.text(1, 1, tn, horizontalalignment="center",
        color="white")

    plt.colorbar()
    plt.show()


custom_classes_combined = ['Monologue or Reading', '
    Conversation']

# Replace y_pred_way2 with the predicted labels
    obtained from the classifier trained on the combined
     dataset (way2)
# Plot confusion matrix for the combined categories
plot_binary_confusion_matrix(y_test_way2, y_pred_way2,
    custom_classes_combined, 'Confusion Matrix - SVM
    classifier')
```

```
#-------------------------------------------------
#SVM with cross validation

# Cross validation

from sklearn.model_selection import cross_val_score
from sklearn.svm import SVC

svm_classifier = SVC(kernel='linear')

# 5 folds
cv_scores = cross_val_score(svm_classifier, X, y, cv=5)

# the scores of cross-validation:
print("Cross-Validation Scores:", cv_scores)

#  mean and standard deviation of cross-validation
   scores
mean_cv_score = cv_scores.mean()
std_cv_score = cv_scores.std()
print("Mean Cross-Validation Score:", mean_cv_score)
print("Standard Deviation of Cross-Validation Scores:",
    std_cv_score)
# :

#-------------------------------------------------
#Random Forest Classifier

##Random Forest Classifier :
from sklearn.ensemble import RandomForestClassifier
rf_classifier = RandomForestClassifier(n_estimators
   =100, random_state=42)
# Train Random Forest Classifier
rf_classifier.fit(X_train_way2, y_train_way2)

# Predictions
y_pred_rf_way2 = rf_classifier.predict(X_test_way2)
```

```
# check  accuracy
accuracy_rf_way2 = accuracy_score(y_test_way2,
    y_pred_rf_way2)
print("Random Forest Accuracy:", accuracy_rf_way2)

# Check MSE
mse_rf_way2 = mean_squared_error(y_test, y_pred_rf)
print("Random Forest Mean Squared Error (MSE):",
    mse_rf_way2)

# Plot confusion matrix for the combined categories -
    SVM classifier
plot_binary_confusion_matrix(y_test_way2,
    y_pred_rf_way2, custom_classes_combined, 'Confusion
    Matrix - Random Forest Classifier')

#-----------------------------------------
#KNN
## k-Nearest Neighbors Classifier

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score,
    mean_squared_error

# Train k-NN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=5)
knn_classifier.fit(X_train_way2, y_train_way2)

# Predictions
y_pred_knn = knn_classifier.predict(X_test_way2)

# Calculate accuracy
accuracy_knn_way2 = accuracy_score(y_test_way2,
    y_pred_knn)
print("k-NN Classifier Accuracy:", accuracy_knn_way2)

# Calculate mean squared error (MSE)
mse_knn_way2 = mean_squared_error(y_test_way2,
```

```
    y_pred_knn)
print("k-NN Classifier Mean Squared Error (MSE):",
    mse_knn_way2)

plot_binary_confusion_matrix(y_test_way2, y_pred_knn,
    custom_classes_combined, 'Confusion Matrix - k-
    Nearest Neighbors Classifier')

#-----------------------------------------------------
#Logistic Regression Classifier

#Logistic Regression Classifier

from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,
    mean_squared_error

# Train logistic regression classifier with adjusted
    solver
logistic_classifier = LogisticRegression(multi_class='
    multinomial', solver='lbfgs', max_iter=1000)
logistic_classifier.fit(X_train_way2, y_train_way2)

# Predictions
y_pred_logistic = logistic_classifier.predict(
    X_test_way2)

# Calculate accuracy
accuracy_logistic_way2 = accuracy_score(y_test_way2,
    y_pred_logistic)
print("Logistic Regression Classifier Accuracy:",
    accuracy_logistic_way2)

# Calculate mean squared error (MSE)
mse_logistic_way2 = mean_squared_error(y_test_way2,
    y_pred_logistic)
print("Logistic Regression Classifier Mean Squared
    Error (MSE):", mse_logistic_way2)
```

```
plot_binary_confusion_matrix(y_test_way2,
    y_pred_logistic, custom_classes_combined, 'Confusion
    Matrix - Logistic Regression Classifier')

#-----------------*-----------------
import matplotlib.pyplot as plt

# Accuracy scores for each model
accuracies = {
    'Model': ['SVM', 'Random Forest', 'k-NN', 'Logistic
        Regression'],
    'Three Categories': [accuracy_SVM_way1,
        accuracy_rf_way1, accuracy_knn_way1,
        accuracy_logistic_way1],
    'Combined Categories': [accuracy_SVM_way2 ,
        accuracy_rf_way2 ,accuracy_knn_way2,
        accuracy_logistic_way2]
}

# Create DataFrame for plotting
accuracy_df = pd.DataFrame(accuracies)

# Plotting
accuracy_df.plot(x='Model', kind='bar', figsize=(8, 5))
plt.ylabel('Accuracy')
plt.title('Model Accuracy Comparison')
plt.ylim(0, 1)
plt.xticks(rotation=45)
plt.legend(title='Category')
plt.show()

#
    ------------------------------------------------------------


#Clustering - On the original data without
    standardizing
#K-means
```

```
#choosing the optimal k by Elbow method
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

# Assuming final_data_filtered is a pandas DataFrame
# Extract the values from the DataFrame
data = final_data_filtered.values

# Define the range of potential number of clusters to
    try
clusters_range = range(1, 11)  # Try cluster numbers
    from 1 to 10

# Initialize an empty list to store the inertia (within
    -cluster sum of squared distances) for each cluster
    number
inertia = []

# Run KMeans for each number of clusters and store the
    inertia
for n_clusters in clusters_range:
    kmeans = KMeans(n_clusters=n_clusters, random_state
        =42)
    kmeans.fit(data)
    inertia.append(kmeans.inertia_)

# Plot the Elbow curve
plt.figure(figsize=(10, 6))
plt.plot(clusters_range, inertia, marker='o', linestyle
    ='-')
plt.title('Elbow Method for Optimal Number of Clusters'
    )
plt.xlabel('Number of Clusters')
plt.ylabel('Inertia')
plt.xticks(clusters_range)
plt.grid(True)
plt.show()
```

```
#-----------------------------

# Find the elbow point programmatically
def find_elbow_point(inertia):
    # Calculate the differences in inertia
    differences = np.diff(inertia)

    # Calculate the second derivative of the inertia
    second_derivative = np.diff(differences)

    # Find the index corresponding to the maximum
       second derivative
    elbow_index = np.argmax(second_derivative) + 1  #
       Add 1 to shift index by 1

    return elbow_index

# Find the index of the elbow point
elbow_index = find_elbow_point(inertia)

# Optimal number of clusters
optimal_n_clusters = elbow_index + 1  # Add 1 to shift
   index by 1

print("Optimal number of clusters:", optimal_n_clusters
   )

#plot k=2,pc=2

#*****************************************
import pandas as pd
from sklearn.preprocessing import StandardScaler,
   OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
```

```
import matplotlib.pyplot as plt

# Assuming final_data_filtered is your DataFrame
# Identify numerical and categorical columns
numerical_features = final_data_filtered.select_dtypes(
    include=['int64', 'float64']).columns
categorical_features = final_data_filtered.
    select_dtypes(include=['object', 'category']).
    columns

# Preprocessing pipeline
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(), categorical_features)
    ]
)

# K-Means Clustering pipeline
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('kmeans', KMeans(n_clusters=2, random_state=42))
])

# Fit and predict clusters
clusters = pipeline.fit_predict(final_data_filtered)
# Mapping cluster numbers to meaningful names
cluster_names = {0: 'Cluster A', 1: 'Cluster B'}
cluster_name_variable_k2 = pd.Series(clusters).map(
    cluster_names)

# Print the first few entries of the cluster name
    variable
print(cluster_name_variable_k2.head())

# Adding the cluster labels to the DataFrame
final_data_filtered['Cluster'] = clusters
```

```
final_data_filtered_way2_clusters =
    final_data_filtered_way2.copy()

final_data_filtered_way2_clusters["cluster"]=clusters


#For stan

final_data_filtered_way2_clusters_stan =
    final_data_filtered_way2_stan.copy()

final_data_filtered_way2_clusters_stan["cluster"]=
    clusters


# Applying PCA to reduce dimensionality to 2 components
pca = PCA(n_components=2)
principal_components = pca.fit_transform(pipeline.
    named_steps['preprocessor'].transform(
    final_data_filtered.drop('Cluster', axis=1)))

# Get the explained variance ratio
explained_variance_ratio = pca.
    explained_variance_ratio_

# Print the explained variance for each principal
    component
print(f"Principal Component 1 explains {
    explained_variance_ratio[0] * 100:.2f}% of the
    variance.")
print(f"Principal Component 2 explains {
    explained_variance_ratio[1] * 100:.2f}% of the
    variance.")

# Plotting the clusters in 2D
plt.figure(figsize=(8, 5))
plt.scatter(principal_components[:, 0],
    principal_components[:, 1], c=clusters, cmap='
```

```
    viridis', alpha=0.6)
plt.scatter(pca.transform(pipeline.named_steps['kmeans'
    ].cluster_centers_)[:, 0], pca.transform(pipeline.
    named_steps['kmeans'].cluster_centers_)[:, 1], s
    =300, c='red', marker='X')
plt.xlabel(f'Principal Component 1 ({
    explained_variance_ratio[0] * 100:.2f}% variance)')
plt.ylabel(f'Principal Component 2 ({
    explained_variance_ratio[1] * 100:.2f}% variance)')
plt.title('K-Means Clustering with PCA')
plt.show()

#----------*----------
#k=2,3pc's
pca_3d = PCA(n_components=3)
principal_components_3d = pca_3d.fit_transform(pipeline
    .named_steps['preprocessor'].transform(
    final_data_filtered.drop('Cluster', axis=1)))

# Get the explained variance ratio for 3 components
explained_variance_ratio_3d = pca_3d.
    explained_variance_ratio_

# Print the explained variance for each principal
    component
print(f"Principal Component 1 explains {
    explained_variance_ratio_3d[0] * 100:.2f}% of the
    variance.")
print(f"Principal Component 2 explains {
    explained_variance_ratio_3d[1] * 100:.2f}% of the
    variance.")
print(f"Principal Component 3 explains {
    explained_variance_ratio_3d[2] * 100:.2f}% of the
    variance.")

# Plotting the clusters in 3D
fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(111, projection='3d')
```

```
sc = ax.scatter(principal_components_3d[:, 0],
    principal_components_3d[:, 1],
    principal_components_3d[:, 2], c=clusters, cmap='
    viridis', alpha=0.6)
ax.scatter(pca_3d.transform(pipeline.named_steps['
    kmeans'].cluster_centers_)[:, 0],
               pca_3d.transform(pipeline.named_steps['
                   kmeans'].cluster_centers_)[:, 1],
               pca_3d.transform(pipeline.named_steps['
                   kmeans'].cluster_centers_)[:, 2],
               s=300, c='red', marker='X')
ax.set_xlabel(f'Principal Component 1 ({
    explained_variance_ratio_3d[0] * 100:.2f}% variance)
    ')
ax.set_ylabel(f'Principal Component 2 ({
    explained_variance_ratio_3d[1] * 100:.2f}% variance)
    ')
ax.set_zlabel(f'Principal Component 3 ({
    explained_variance_ratio_3d[2] * 100:.2f}% variance)
    ')
ax.set_title('3D K-Means Clustering with PCA (k=2)')
plt.show()

#-------*--------

#k=3,2pc's

import pandas as pd
from sklearn.preprocessing import StandardScaler,
    OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Assuming final_data_filtered is your DataFrame
# Identify numerical and categorical columns
```

```
numerical_features = final_data_filtered.select_dtypes(
    include=['int64', 'float64']).columns
categorical_features = final_data_filtered.
    select_dtypes(include=['object', 'category']).
    columns

# Preprocessing pipeline
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(), categorical_features)
    ]
)

# K-Means Clustering pipeline
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('kmeans', KMeans(n_clusters=3, random_state=42))
])

# Fit and predict clusters
clusters = pipeline.fit_predict(final_data_filtered)

# Mapping cluster numbers to meaningful names
cluster_names = {0: 'Cluster A', 1: 'Cluster B', 2: '
    Cluster C'}
cluster_name_variable_k3 = pd.Series(clusters).map(
    cluster_names)

# Print the first few entries of the cluster name
    variable
print(cluster_name_variable_k3.head())

# Adding the cluster labels to the DataFrame
final_data_filtered['Cluster'] = clusters

# Applying PCA to reduce dimensionality to 2 components
pca = PCA(n_components=2)
```

```
principal_components = pca.fit_transform(pipeline.
    named_steps['preprocessor'].transform(
    final_data_filtered.drop('Cluster', axis=1)))

# Get the explained variance ratio
explained_variance_ratio = pca.
    explained_variance_ratio_

# Print the explained variance for each principal
    component
print(f"Principal Component 1 explains {
    explained_variance_ratio[0] * 100:.2f}% of the
    variance.")
print(f"Principal Component 2 explains {
    explained_variance_ratio[1] * 100:.2f}% of the
    variance.")

# Plotting the clusters in 2D
plt.figure(figsize=(8, 5))
plt.scatter(principal_components[:, 0],
    principal_components[:, 1], c=clusters, cmap='
    viridis', alpha=0.6)
plt.scatter(pca.transform(pipeline.named_steps['kmeans'
    ].cluster_centers_)[:, 0], pca.transform(pipeline.
    named_steps['kmeans'].cluster_centers_)[:, 1], s
    =300, c='red', marker='X')
plt.xlabel(f'Principal Component 1 ({
    explained_variance_ratio[0] * 100:.2f}% variance)')
plt.ylabel(f'Principal Component 2 ({
    explained_variance_ratio[1] * 100:.2f}% variance)')
plt.title('K-Means Clustering with PCA')
plt.show()

#-------*-----------

#k=3,3pc's

# Applying PCA to reduce dimensionality to 3 components
```

```
pca_3d = PCA(n_components=3)
principal_components_3d = pca_3d.fit_transform(pipeline
    .named_steps['preprocessor'].transform(
    final_data_filtered.drop('Cluster', axis=1)))

# Get the explained variance ratio for 3 components
explained_variance_ratio_3d = pca_3d.
    explained_variance_ratio_

# Print the explained variance for each principal
    component
print(f"Principal Component 1 explains {
    explained_variance_ratio_3d[0] * 100:.2f}% of the
    variance.")
print(f"Principal Component 2 explains {
    explained_variance_ratio_3d[1] * 100:.2f}% of the
    variance.")
print(f"Principal Component 3 explains {
    explained_variance_ratio_3d[2] * 100:.2f}% of the
    variance.")

# Plotting the clusters in 3D
fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(111, projection='3d')
sc = ax.scatter(principal_components_3d[:, 0],
    principal_components_3d[:, 1],
    principal_components_3d[:, 2], c=clusters, cmap='
    viridis', alpha=0.6)
ax.scatter(pca_3d.transform(pipeline.named_steps['
    kmeans'].cluster_centers_)[:, 0],
               pca_3d.transform(pipeline.named_steps['
                   kmeans'].cluster_centers_)[:, 1],
               pca_3d.transform(pipeline.named_steps['
                   kmeans'].cluster_centers_)[:, 2],
               s=300, c='red', marker='X')
ax.set_xlabel(f'Principal Component 1 ({
    explained_variance_ratio_3d[0] * 100:.2f}% variance)
    ')
```

```
ax.set_ylabel(f'Principal Component 2 ({
    explained_variance_ratio_3d[1] * 100:.2f}% variance)
    ')
ax.set_zlabel(f'Principal Component 3 ({
    explained_variance_ratio_3d[2] * 100:.2f}% variance)
    ')
ax.set_title('3D K-Means Clustering with PCA')
plt.show()

#-------*----------

#Adding chi-square test

import pandas as pd
from sklearn.preprocessing import StandardScaler,
    OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from scipy.stats import chi2_contingency  # Ensure this
    import statement is present
import matplotlib.pyplot as plt
# Define the mapping dictionary for situations
mapping = {
    0: 'monologue',
    1: 'conversation',
    2: 'reading'
}

# Create a new list with the mapped names for
    situations
situations_in_dat = [mapping[value] for value in
    final_data_filtered["situation"]]

# Create a new list with the mapped names for
    situations in way 2
mapping_way2 = {
```

```
    0: 'monologue_reading',
    1: 'conversation',
    2: 'monologue_reading'
}
situations_in_dat_way2 = [mapping_way2[value] for value
    in final_data_filtered["situation"]]

# Contingency tables
contingency_table_k2 = pd.crosstab(
    cluster_name_variable_k2, situations_in_dat)
contingency_table_k3 = pd.crosstab(
    cluster_name_variable_k3, situations_in_dat_way2)

# Chi-square tests
chi2_k2, p_k2, dof_k2, ex_k2 = chi2_contingency(
    contingency_table_k2)
chi2_k3, p_k3, dof_k3, ex_k3 = chi2_contingency(
    contingency_table_k3)

# Convert expected frequencies to DataFrame for better
    visualization
ex_k2_df = pd.DataFrame(ex_k2, index=
    contingency_table_k2.index, columns=
    contingency_table_k2.columns)
ex_k3_df = pd.DataFrame(ex_k3, index=
    contingency_table_k3.index, columns=
    contingency_table_k3.columns)

# Print the contingency tables
print("Contingency Table for k=2 Clusters:")
print(contingency_table_k2)
print("\nContingency Table for k=3 Clusters:")
print(contingency_table_k3)

# Print the chi-square test results
print("\nChi-Square Test for k=2 Clusters:")
print(f"Chi2 Statistic: {chi2_k2}")
print(f"p-value: {p_k2}")
```

```
print(f"Degrees of Freedom: {dof_k2}")
print("\nExpected Frequencies:")
print(ex_k2_df)

print("\nChi-Square Test for k=3 Clusters:")
print(f"Chi2 Statistic: {chi2_k3}")
print(f"p-value: {p_k3}")
print(f"Degrees of Freedom: {dof_k3}")
print("\nExpected Frequencies:")
print(ex_k3_df)

#-------------------------------------------------------------------
#Hierarchical Clustering - dendogram
#Average method

import matplotlib.pyplot as plt
import seaborn as sns
from scipy.cluster.hierarchy import dendrogram, linkage
import pandas as pd

# Assuming final_data_filtered is your DataFrame with
    preprocessed data

# Calculate the linkage matrix using average linkage
linkage_matrix_av = linkage(final_data_filtered, method
    ='average', metric='euclidean')

# Plot the dendrogram
plt.figure(figsize=(5, 3))
dendrogram(linkage_matrix_av, leaf_rotation=90,
    leaf_font_size=10)
plt.title('Hierarchical Clustering Dendrogram (Average
    Linkage)')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.xticks(ticks=[], labels=[])
plt.tight_layout()
plt.show()
```

```
from scipy.cluster.hierarchy import fcluster

# Determine the number of clusters you want
num_clusters = 3

# Cut the dendrogram to obtain clusters
clusters = fcluster(linkage_matrix_av, num_clusters,
    criterion='maxclust')

# Print the cluster assignments
print(clusters)



# Define the mapping dictionary
mapping = {
     0: 'monologue',
     1: 'conversation',
     2: 'reading'
}

# Create a new list with the mapped names
situations_in_dat = [mapping[value] for value in
    final_data_filtered["situation"]]

print(situations_in_dat)


from scipy.stats import chi2_contingency

# Assuming 'clusters' and 'situations' are the vectors
    of cluster labels and situation labels
# Create a contingency table
contingency_table = pd.crosstab(clusters,
    situations_in_dat)

# Perform Chi-square test
```

```
chi2, p, dof, expected = chi2_contingency(
    contingency_table)

print("Chi-square statistic:", chi2)
print("p-value:", p)

#----------------*----------------
#single method

import matplotlib.pyplot as plt
import seaborn as sns
from scipy.cluster.hierarchy import dendrogram, linkage
import pandas as pd

# Assuming final_data_filtered is your DataFrame with
    preprocessed data

# Calculate the linkage matrix using average linkage
linkage_matrix_single = linkage(final_data_filtered,
    method='single', metric='euclidean')

# Plot the dendrogram
plt.figure(figsize=(8, 5))
dendrogram(linkage_matrix_single, leaf_rotation=90,
    leaf_font_size=10)
plt.title('Hierarchical Clustering Dendrogram (singel
    Linkage)')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.xticks(ticks=[], labels=[])
plt.tight_layout()
plt.show()

from scipy.cluster.hierarchy import fcluster

# Determine the number of clusters you want
num_clusters = 3
```

```
# Cut the dendrogram to obtain clusters
clusters_single = fcluster(linkage_matrix_single,
    num_clusters, criterion='maxclust')

# Print the cluster assignments
print(clusters)

 from scipy.stats import chi2_contingency

# Assuming 'clusters' and 'situations' are the vectors
    of cluster labels and situation labels
# Create a contingency table
contingency_table = pd.crosstab(clusters_single,
    situations_in_dat)

# Perform Chi-square test
chi2, p, dof, expected = chi2_contingency(
    contingency_table)

print("Chi-square statistic:", chi2)
print("p-value:", p)

#-------*---------

#complete
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.cluster.hierarchy import dendrogram, linkage
import pandas as pd

# Assuming final_data_filtered is your DataFrame with
    preprocessed data

# Calculate the linkage matrix using average linkage
linkage_matrix_comp = linkage(final_data_filtered,
    method='complete', metric='euclidean')

# Plot the dendrogram
```

```
plt.figure(figsize=(8, 5))
dendrogram(linkage_matrix_comp, leaf_rotation=90,
    leaf_font_size=10)
plt.title('Hierarchical Clustering Dendrogram (Complete
    Linkage)')
plt.xlabel('Samples')
plt.ylabel('Distance')
plt.xticks(ticks=[], labels=[])
plt.tight_layout()
plt.show()

from scipy.cluster.hierarchy import fcluster

# Determine the number of clusters you want
num_clusters = 3

# Cut the dendrogram to obtain clusters
clusters_comp = fcluster(linkage_matrix_comp,
    num_clusters, criterion='maxclust')

# Print the cluster assignments
print(clusters_comp)

from scipy.stats import chi2_contingency

# Assuming 'clusters' and 'situations' are the vectors
    of cluster labels and situation labels
# Create a contingency table
contingency_table = pd.crosstab(clusters_comp,
    situations_in_dat)

# Perform Chi-square test
chi2, p, dof, expected = chi2_contingency(
    contingency_table)

print("Chi-square statistic:", chi2)
print("p-value:", p)
```

```
#
    ------------------------------------------------------------------------


#As we did the final machine learning on the combined
    data, then we will continue with that data-set ,
    with adding the features as a variables and the
    cluster number too (added in the 2pc's k=2 section)

#1. Adding the clustering for optimal k=2 2D
#2. Adding the histograms
#3. Adding a new clusters variables to the data(
    combined) then applying machine #learning methods
    and compare with machine learning on combined data

#final_data_filtered_way2
print(final_data_filtered_way2_clusters)
final_data_filtered_way2_clusters.columns

#
    ------------------------------------------------------------------------


#Histograms for features variables, in different
    clusters

import matplotlib.pyplot as plt

# Define the variables of interest and the clusters
variables_of_interest = ['pitch_score', '
    intensity_score', 'disfluency_duration', '
    spectral_feature_scalar']
clusters = final_data_filtered_way2_clusters['cluster'
    ].unique()

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10,
    6))
```

```
# Loop over each variable and plot in the corresponding
    subplot
for i, variable in enumerate(variables_of_interest):
    row = i // 2  # Row index for subplot
    col = i % 2   # Column index for subplot
    ax = axes[row, col]  # Get the corresponding axis
    ax.set_title(f'Histogram of {variable} in Different
       Clusters')
    for cluster in clusters:
        # Filter data for the current cluster
        data_cluster =
           final_data_filtered_way2_clusters[
           final_data_filtered_way2_clusters['cluster']
            == cluster]
        # Plot histogram for the current variable in
           the current cluster
        ax.hist(data_cluster[variable], bins=20, alpha
           =0.5, label=f'Cluster {cluster}')
    ax.set_xlabel(variable)
    ax.set_ylabel('Frequency')
    ax.legend()

# Adjust layout and display the plot
plt.tight_layout()
plt.show()

#-----------------------------------------------------
#Box-plots

import matplotlib.pyplot as plt

# Define the variables of interest and the clusters
variables_of_interest = ['pitch_score', '
   intensity_score', 'disfluency_duration', '
   spectral_feature_scalar']
clusters = final_data_filtered_way2_clusters['cluster'
   ].unique()
```

```
# Create a 2x2 grid of subplots
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(9,
    6))

# Loop over each variable and plot in the corresponding
    subplot
for i, variable in enumerate(variables_of_interest):
    row = i // 2  # Row index for subplot
    col = i % 2  # Column index for subplot
    ax = axes[row, col]  # Get the corresponding axis
    ax.set_title(f'Box Plot of {variable} in Different
        Clusters')
    # Create a list to store boxplot data for each
        cluster
    boxplot_data = []
    for cluster in clusters:
        # Filter data for the current cluster
        data_cluster =
            final_data_filtered_way2_clusters[
            final_data_filtered_way2_clusters['cluster']
             == cluster]
        # Add data for the current cluster to the list
        boxplot_data.append(data_cluster[variable])
    # Plot boxplot for the current variable in the
        current subplot
    ax.boxplot(boxplot_data, labels=[f'Cluster {cluster
        }' for cluster in clusters])
    ax.set_xlabel('Cluster')
    ax.set_ylabel(variable)

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
#
```

```
import matplotlib.pyplot as plt
```

```
# Define the variables of interest and the clusters
variables_of_interest = ['pitch_score', '
    intensity_score', 'disfluency_duration', '
    spectral_feature_scalar']
situations = final_data_filtered_way2_clusters['
    situation'].unique()
situation_labels = {0: 'reading or monologue', 1: '
    conversation'}  # Define the mapping

# Define custom colors for the histograms
colors = ['skyblue', 'salmon']

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10,
    6))

# Loop over each variable and plot in the corresponding
    subplot
for i, variable in enumerate(variables_of_interest):
    row = i // 2  # Row index for subplot
    col = i % 2   # Column index for subplot
    ax = axes[row, col]  # Get the corresponding axis
    ax.set_title(f'Histogram of {variable} in Different
        Situations')
    for situation in situations:
        # Filter data for the current situation
        data_situation =
            final_data_filtered_way2_clusters[
            final_data_filtered_way2_clusters['situation
            '] == situation]
        # Plot histogram for the current variable in
            the current situation
        ax.hist(data_situation[variable], bins=20,
            alpha=0.5, label=situation_labels[situation
            ], color=colors[situation])
    ax.set_xlabel(variable)
    ax.set_ylabel('Frequency')
    ax.legend()
```

```
# Adjust layout and display the plot
plt.tight_layout()
plt.show()

#-----------------------------------------------------------
# Machine learning on the data with cluster variables (
    combined data) with cross-validation
import pandas as pd
from sklearn.model_selection import train_test_split,
    cross_val_score
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Dividing the data into train and test
X = final_data_filtered_way2_clusters_stan.drop(columns
    =['situation'])  # Features
y = final_data_filtered_way2_clusters_stan['situation']
     # Labels
X_train, X_test, y_train, y_test = train_test_split(X,
    y, test_size=0.3, random_state=42)

# Initialize dictionaries to store accuracy values
accuracy_dict = {}
mean_cv_score_dict = {}
std_cv_score_dict = {}

# SVM
svm_classifier = SVC(kernel='linear')
svm_classifier.fit(X_train, y_train)
y_pred_svm = svm_classifier.predict(X_test)
accuracy_dict['SVM'] = accuracy_score(y_test,
    y_pred_svm)

# Cross-validation for SVM
```

```
cv_scores_svm = cross_val_score(svm_classifier, X, y,
    cv=5)
mean_cv_score_dict['SVM'] = cv_scores_svm.mean()
std_cv_score_dict['SVM'] = cv_scores_svm.std()

# Random Forest
rf_classifier = RandomForestClassifier(n_estimators
    =100, random_state=42)
rf_classifier.fit(X_train, y_train)
y_pred_rf = rf_classifier.predict(X_test)
accuracy_dict['Random Forest'] = accuracy_score(y_test,
     y_pred_rf)

# Cross-validation for Random Forest
cv_scores_rf = cross_val_score(rf_classifier, X, y, cv
    =5)
mean_cv_score_dict['Random Forest'] = cv_scores_rf.mean
    ()
std_cv_score_dict['Random Forest'] = cv_scores_rf.std()

# KNN
knn_classifier = KNeighborsClassifier(n_neighbors=5)
knn_classifier.fit(X_train, y_train)
y_pred_knn = knn_classifier.predict(X_test)
accuracy_dict['KNN'] = accuracy_score(y_test,
    y_pred_knn)

# Cross-validation for KNN
cv_scores_knn = cross_val_score(knn_classifier, X, y,
    cv=5)
mean_cv_score_dict['KNN'] = cv_scores_knn.mean()
std_cv_score_dict['KNN'] = cv_scores_knn.std()

# Logistic Regression
logistic_classifier = LogisticRegression(multi_class='
    multinomial', solver='lbfgs', max_iter=1000)
logistic_classifier.fit(X_train, y_train)
y_pred_logistic = logistic_classifier.predict(X_test)
```

```
accuracy_dict['Logistic Regression'] = accuracy_score(
    y_test, y_pred_logistic)

# Cross-validation for Logistic Regression
cv_scores_logistic = cross_val_score(
    logistic_classifier, X, y, cv=5)
mean_cv_score_dict['Logistic Regression'] =
    cv_scores_logistic.mean()
std_cv_score_dict['Logistic Regression'] =
    cv_scores_logistic.std()

# Create DataFrame from accuracy dictionary
accuracy_df = pd.DataFrame(accuracy_dict.items(),
    columns=['Model', 'Accuracy (Train/Test)'])
accuracy_df['Mean CV Score'] = mean_cv_score_dict.
    values()
accuracy_df['Std CV Score'] = std_cv_score_dict.values
    ()

# Display the DataFrame
print(accuracy_df)

#
    ------------------------------------------------------------------------


#without clusters variable (2 categories) WITH CROSS
    VALIDATION

import pandas as pd
from sklearn.model_selection import train_test_split,
    cross_val_score
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Dividing the data into train and test
```

```
X = final_data_filtered_way2_stan.drop(columns=['
    situation']) # Features
y = final_data_filtered_way2_stan['situation'] #
    Labels
X_train, X_test, y_train, y_test = train_test_split(X,
    y, test_size=0.3, random_state=42)

# Initialize dictionaries to store accuracy values
accuracy_dict = {}
mean_cv_score_dict = {}
std_cv_score_dict = {}

# SVM
svm_classifier = SVC(kernel='linear')
svm_classifier.fit(X_train, y_train)
y_pred_svm = svm_classifier.predict(X_test)
accuracy_dict['SVM'] = accuracy_score(y_test,
    y_pred_svm)

# Cross-validation for SVM
cv_scores_svm = cross_val_score(svm_classifier, X, y,
    cv=5)
mean_cv_score_dict['SVM'] = cv_scores_svm.mean()
std_cv_score_dict['SVM'] = cv_scores_svm.std()

# Random Forest
rf_classifier = RandomForestClassifier(n_estimators
    =100, random_state=42)
rf_classifier.fit(X_train, y_train)
y_pred_rf = rf_classifier.predict(X_test)
accuracy_dict['Random-Forest'] = accuracy_score(y_test,
     y_pred_rf)

# Cross-validation for Random Forest
cv_scores_rf = cross_val_score(rf_classifier, X, y, cv
    =5)
mean_cv_score_dict['Random-Forest'] = cv_scores_rf.mean
    ()
```

```
std_cv_score_dict['Random Forest'] = cv_scores_rf.std()

# KNN
knn_classifier = KNeighborsClassifier(n_neighbors=5)
knn_classifier.fit(X_train, y_train)
y_pred_knn = knn_classifier.predict(X_test)
accuracy_dict['KNN'] = accuracy_score(y_test,
    y_pred_knn)

# Cross-validation for KNN
cv_scores_knn = cross_val_score(knn_classifier, X, y,
    cv=5)
mean_cv_score_dict['KNN'] = cv_scores_knn.mean()
std_cv_score_dict['KNN'] = cv_scores_knn.std()

# Logistic Regression
logistic_classifier = LogisticRegression(multi_class='
    multinomial', solver='lbfgs', max_iter=1000)
logistic_classifier.fit(X_train, y_train)
y_pred_logistic = logistic_classifier.predict(X_test)
accuracy_dict['Logistic Regression'] = accuracy_score(
    y_test, y_pred_logistic)

# Cross-validation for Logistic Regression
cv_scores_logistic = cross_val_score(
    logistic_classifier, X, y, cv=5)
mean_cv_score_dict['Logistic Regression'] =
    cv_scores_logistic.mean()
std_cv_score_dict['Logistic Regression'] =
    cv_scores_logistic.std()

# Create DataFrame from accuracy dictionary
accuracy_df = pd.DataFrame(accuracy_dict.items(),
    columns=['Model', 'Accuracy (Train/Test)'])
accuracy_df['Mean CV Score'] = mean_cv_score_dict.
    values()
accuracy_df['Std CV Score'] = std_cv_score_dict.values
    ()
```

```
# Display the DataFrame
print(accuracy_df)

#-----------------------------------------------------------------
# With 3 situations(before combining)-way A with CROSS
    VALIDATION
import pandas as pd
from sklearn.model_selection import train_test_split,
    cross_val_score
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Dividing the data into train and test
X = final_data_filtered_stan.drop(columns=['situation'
    ]) # Features
y = final_data_filtered_stan['situation'] # Labels
X_train, X_test, y_train, y_test = train_test_split(X,
    y, test_size=0.3, random_state=42)

# Initialize dictionaries to store metrics
accuracy_dict = {}
mean_cv_score_dict = {}
std_cv_score_dict = {}

# Function to perform training, prediction, and cross-
    validation
def evaluate_model(model, model_name):
    # Train the model
    model.fit(X_train, y_train)
    # Predictions
    y_pred = model.predict(X_test)
    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    # Cross-validation
```

```
    cv_scores = cross_val_score(model, X, y, cv=5)
    mean_cv_score = cv_scores.mean()
    std_cv_score = cv_scores.std()
     # Store metrics in dictionaries
    accuracy_dict[model_name] = accuracy
    mean_cv_score_dict[model_name] = mean_cv_score
    std_cv_score_dict[model_name] = std_cv_score

# SVM
svm_classifier = SVC(kernel='linear')
evaluate_model(svm_classifier, 'SVM')

# Random Forest
rf_classifier = RandomForestClassifier(n_estimators
   =100, random_state=42)
evaluate_model(rf_classifier, 'Random Forest')

# KNN
knn_classifier = KNeighborsClassifier(n_neighbors=5)
evaluate_model(knn_classifier, 'KNN')

# Logistic Regression
logistic_classifier = LogisticRegression(multi_class='
   multinomial', solver='saga', max_iter=2000)
evaluate_model(logistic_classifier, 'Logistic
   Regression')

# Create DataFrame from the dictionaries
results_df = pd.DataFrame({
    'Model': accuracy_dict.keys(),
    'Accuracy (Train/Test)': accuracy_dict.values(),
    'Mean CV Score': mean_cv_score_dict.values(),
    'Std CV Score': std_cv_score_dict.values()
})

# Display the DataFrame
print(results_df)
```