King Saud University

College of Computer and Information Sciences

Department of Information Technology

IT326:Data Mining

Project final report

Student Stress Factors

Group#	4	
Section#	56547	
Group members	**Student name**	**ID**
	Rahaf Alfantoukh	443200954
	Hadeel Almutairi	443200935
	Jana Almohsen	444204517
	Amira Aljeraisy	443200950
	Daad Alali	443200944

Supervised By: Dr. Hessah Alsaaran

Table of Contents

Problem

Student life is filled with a variety of difficulties that go beyond academic tasks, including social pressures, struggles with managing time, and the demands of personal growth. The widespread stress students face can have a profound negative effect on their mental well-being and academic achievements. However, there is still a gap in targeted, data-backed research that examines the diverse causes of this stress and their specific effects on students' everyday lives and educational success

Data Mining Task

The "Student Stress Factors" dataset aims to investigate the contributors to student stress, including demographic details, academic performance, lifestyle habits, and mental health metrics. The classification task focuses on predicting stress levels ('Low', 'Medium', 'High') to identify students at risk, enabling targeted support and guiding policy changes. Simultaneously, the clustering task aims to group students with similar traits, facilitating the creation of tailored support programs and uncovering deeper insights into the various factors influencing stress. Together, these tasks strive to improve student well-being by equipping educational institutions with actionable, data-driven strategies to enhance support systems and interventions

Data

```
num_objects = len(df)
attributes_info = pd.DataFrame({
    'Attribute Name': df.columns,
    'Data Type': df.dtypes.values
})
print("Number of attributes:" ,len(df.columns))
print()
print("Attributes and their types:")
print(attributes_info)
print()
print("Number of objects: ",num_objects)
```

Number of attributes: 21

Number of objects: 1100

Class label: stress_level

The source: https://www.kaggle.com/datasets/rxnach/student-stress-factors-a-comprehensive-analysis

our dataset have no missing values

Missing values:

```
# Check for missing values
missing_values = df.isnull().sum() # This creates a Series containing the count of missing values per column
print("Missing values per column:") # Print a message indicating that missing values will be displayed
print(missing_values) # Print missing values per column
```

```
Missing values per column:
anxiety_level                   0
self_esteem                     0
mental_health_history           0
depression                      0
headache                        0
blood_pressure                  0
sleep_quality                   0
breathing_problem               0
noise_level                     0
living_conditions               0
safety                          0
basic_needs                     0
academic_performance            0
study_load                      0
teacher_student_relationship    0
future_career_concerns          0
social_support                  0
peer_pressure                   0
extracurricular_activities      0
bullying                        0
stress_level                    0
dtype: int64
```

Attribute Name	Description	Range	Attribute type
Anxiety Level	Intensity of anxiety symptoms	0 to 21	int64
Self Esteem	Overall subjective evaluation of one's own worth	0 to 30	int64
Mental Health History	History of mental health issues (0=No, 1=Yes)	0 to 1	int64
Depression	Severity of depression symptoms	0 to 27	int64
Headache	Frequency or severity of headaches	0 to 5	int64
Blood Pressure	Levels of blood pressure (1=Low, 2=Normal, 3=High)	1 to 3	int64
Sleep Quality	Perceived quality of sleep	0 to 5	int64
Breathing Problem	Severity of breathing problems	0 to 5	int64
Noise Level	Level of noise in the environment	0 to 5	int64
Living Conditions	Quality of living conditions	0 to 5	int64
Safety	Perception of safety	0 to 5	int64
Basic Needs	Fulfillment of basic needs	0 to 5	int64
Academic Performance	Perception of academic performance	0 to 5	int64
Study Load	Perceived workload from studies	0 to 5	int64
Teacher-Student Relationship	Quality of teacher-student relationships	0 to 5	int64
Future Career Concerns	Level of concern about future career prospects	0 to 5	int64
Social Support	Level of social support received	0 to 3	int64
Peer Pressure	Level of pressure felt from peers	0 to 5	int64
Extracurricular Activities	Extent of participation in activities	0 to 5	int64
Bullying	Presence and severity of bullying experiences	0 to 5	int64
Stress Level	Overall level of stress experienced	0 to 2	int64

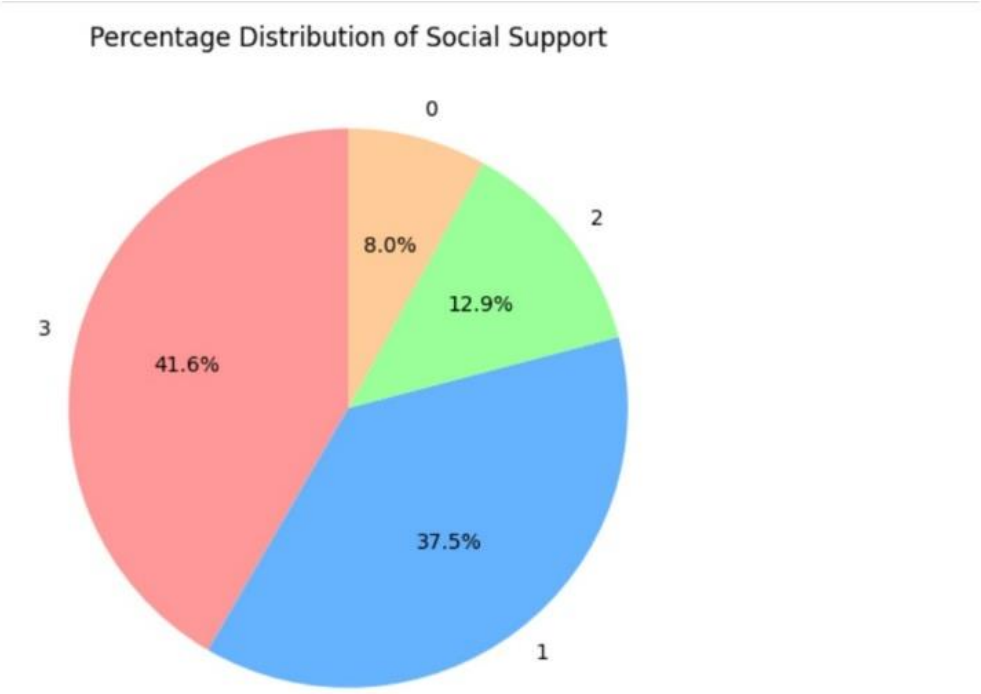
Graphs

```
import matplotlib.pyplot as plt
import seaborn as sns

# Display the class distribution (counts of each class label)
print(class_distribution)
plt.figure(figsize=(6, 4)) # Set up the figure size for the plot
sns.countplot(x='stress_level', data=df, color='lightblue') # Use Seaborn's countplot to plot the distribution of the 'stres
plt.title('Stress Level distribution') # Set the title of the plot
plt.show() # Display the plot
```

This bar chart indicates that the dataset has an equal distribution of data across all stress levels, where 0 represents low stress, 1 stands for medium stress, and 2 indicates high stress. This balance is crucial for ensuring the integrity of analysis and predictions, as it prevents class imbalance that can distort results or cause predictive models to become biased. In an imbalanced dataset, models may disproportionately favor the majority class (e.g., "low stress"), leading to inaccurate and unfair predictions for underrepresented categories (e.g., "high stress"). The balanced representation in this dataset ensures more reliable and fair predictions, allowing for better understanding and intervention across all stress levels, thereby promoting effective strategies for managing student stress.

Pie chart (Social Support):

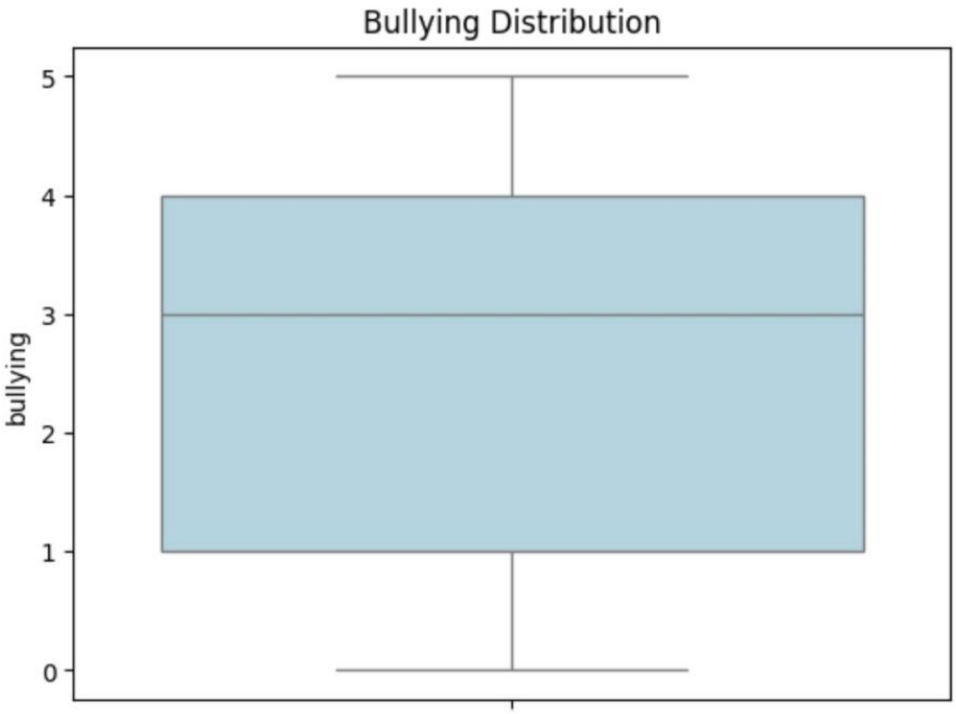
```
data2 = df['social_support'].value_counts(normalize=True) * 100 # Calculate the percentage distribution of the 'social_suppo
custom_colors = ['#ff9999', '#66b3ff', '#99ff99', '#ffcc99']
data2.plot.pie(autopct='%1.1f%%', figsize=(6, 6), startangle=90, colors=custom_colors) # Plot the percentage distribution as
plt.title('Percentage Distribution of Social Support') # Set the title of the pie chart
plt.ylabel('') # Remove the y-axis label for a cleaner pie chart presentation
plt.show() # Display the pie chart
```



The pie chart reveals that the majority of students (over 70%) experience either low or high levels of social support, with the largest percentage (41.6%) feeling well-supported. This suggests that many students have access to strong social networks, which can play a critical role in their academic success and mental health. However, around 8% of students report having no support, which is concerning. A lack of social support can significantly impact a student's ability to manage stress, maintain motivation, and succeed academically. Addressing this issue through targeted interventions, such as peer mentoring, counseling services, or group activities, could help those with little to no support build stronger connections and improve their overall well-being. Understanding these different levels of social support is key to developing strategies that support students' academic performance and mental health.

Box plot (Bullying):

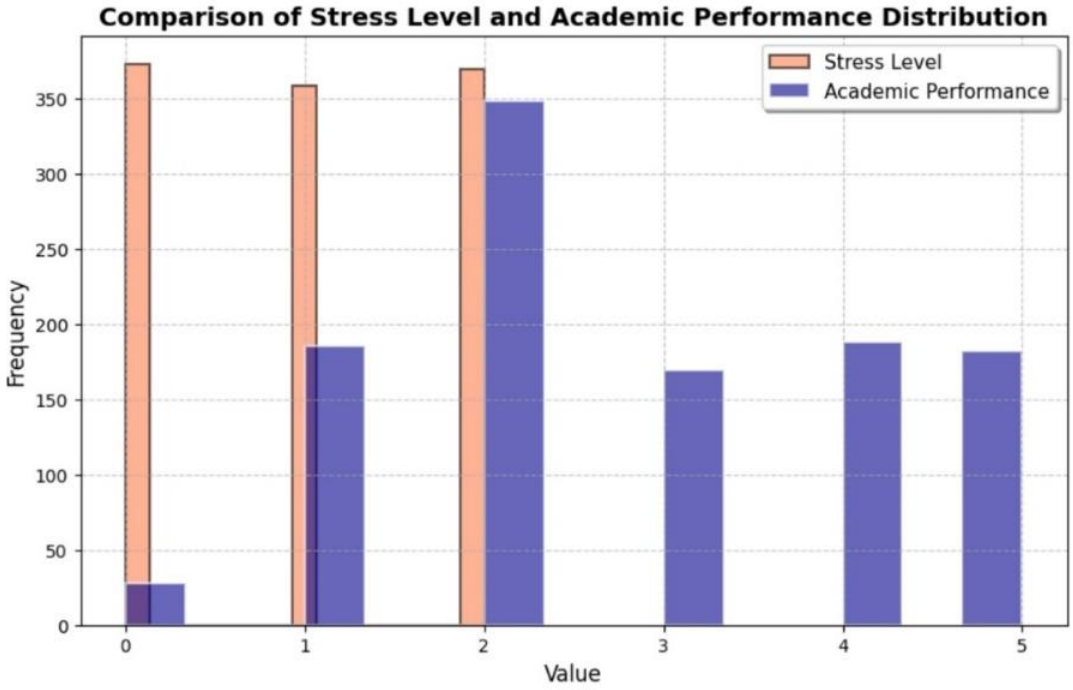
```
# Create a box plot for the 'bullying' column in the dataset
sns.boxplot(data=df['bullying'], color='lightblue')
plt.title('Bullying Distribution') # Set the title for the plot
plt.show() # Display the plot
```



The box plot shows the distribution of the "bullying" variable, with the interquartile range (IQR) captured by the box, and the 25th and 75th percentiles at the edges. The median bullying level is around 3, as indicated by the line inside the box. The whiskers extend to the minimum and maximum values, showing that most data points lie between 1 and 5, with no significant outliers. This distribution suggests that bullying incidents are relatively common in the population, with most students experiencing moderate levels of bullying. The lack of outliers indicates that extreme cases are rare, and interventions may need to focus on addressing the more typical experiences of bullying rather than isolated severe incidents. Understanding this spread is crucial for tailoring prevention and support programs.

Histogram (Stress level, Academic performance)

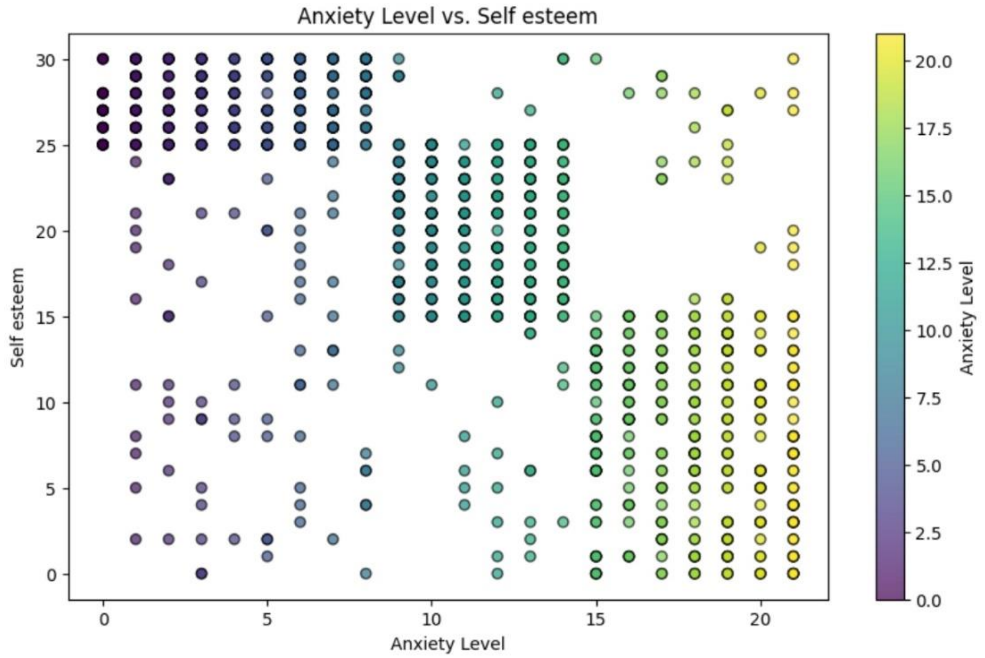
```
plt.figure(figsize=(10, 6)) # Set the figure size for the histogram
plt.hist(df['stress_level'],
         bins=15,                 # Number of bins for the histogram
         color='coral',          # Color for the Stress Level histogram
         edgecolor='black',      # Color of the bin edges
         alpha=0.6,              # Transparency level of the histogram
         label='Stress Level',    # Label for the legend
         histtype='stepfilled',   # Style of the histogram
         linewidth=1.5)          # Width of the bin edges

plt.hist(df['academic_performance'],
         bins=15,                 # Number of bins for the histogram
         color='darkblue',       # Color for the Academic Performance histogram
         edgecolor='white',      # Color of the bin edges
         alpha=0.6,              # Transparency level of the histogram
         label='Academic Performance', # Label for the legend
         histtype='stepfilled',   # Style of the histogram
         linewidth=1.5)          # Width of the bin edges

plt.title('Comparison of Stress Level and Academic Performance Distribution', fontsize=14, fontweight='bold')  # Title of th
plt.xlabel('Value', fontsize=12)        # X-axis label
plt.ylabel('Frequency', fontsize=12)    # Y-axis label
plt.grid(True, linestyle='--', alpha=0.7)  # Style of the gridlines
plt.legend(frameon=True, fancybox=True, shadow=True, loc='upper right', fontsize=11)  # Legend properties

# Display the histogram
plt.show()
```



Based on the histogram illustrating the relationship between stress level and academic performance, it is clear that students with higher academic achievement tend to exhibit elevated stress levels. This trend indicates that as students strive for better grades and higher academic standing, they often encounter increased pressure and demands associated with their studies.

Scater plot(Anxiety Level, Self esteem)

```
plt.figure(figsize=(10, 6))  # Set the figure size for the scatter plot

# Normalize the anxiety level for color mapping
norm = plt.Normalize(df['anxiety_level'].min(), df['anxiety_level'].max())

# Create a scatter plot with colors based on anxiety levels
scatter = plt.scatter(df['anxiety_level'], df['self_esteem'],
                      c=df['anxiety_level'],  # Color based on anxiety levels
                      cmap='viridis',         # Colormap to use
                      norm=norm,              # Normalize values for colormap
                      alpha=0.7,             # Set transparency of points
                      edgecolor='black')      # Outline color for points

plt.title('Anxiety Level vs. Self esteem')  # Set the title for the scatter plot to describe the data being represented
plt.xlabel('Anxiety Level')  # Label the x-axis to indicate that it represents anxiety levels
plt.ylabel('Self esteem')  # Label the y-axis to indicate that it represents depression levels
cbar = plt.colorbar(scatter) # Add a colorbar to indicate the mapping of colors to anxiety levels
cbar.set_label('Anxiety Level')  # Label for the color bar

plt.show()
```



This plot illustrates a negative relationship between anxiety level and self-esteem, suggesting that as anxiety levels increase, self-esteem tends to decrease. This finding aligns with expectations in psychological studies, where heightened anxiety is often associated with lower self-worth and confidence. The observed pattern indicates a potential link between these variables in your data, highlighting the importance of addressing anxiety to promote healthier self-esteem levels. Further analysis on the correlation between anxiety and self-esteem, as well as the potential causation, may be warranted to understand the dynamics of these relationships better and to develop effective interventions.

Statistical summaries:

```
summary_stats=df.describe()
print(summary_stats)
```

Statistical summaries:

By using the df.describe() function, we obtain these results.

Anxiety Levels

The dataset includes 1,100 entries on anxiety levels, revealing an average score of approximately 11.06, with a standard deviation of 6.12. Anxiety levels range from a minimum of 0 to a maximum of 21, with most individuals scoring between 6 and 16.

Self-Esteem

In terms of self-esteem, there are also 1,100 entries. The average self-esteem score is about 17.78, with a standard deviation of 8.94. Scores range from 0 to 30, predominantly falling between 11 and 26.

Mental Health History

Regarding mental health history, roughly 49.27% of the individuals in the dataset reported having a history of mental health issues, while 50.73% did not.

Depression Levels

The dataset includes 1,100 instances related to depression levels, with an average score of approximately 12.56 and a standard deviation of 7.73. Scores range from 0 to 27, with most values falling between 6 and 19.

Headache Reports

There are 1,100 records of reported headache levels, showing an average score of about 2.51, with a standard deviation of 1.41. The headache scores range from 0 to 5, with the majority of responses clustering between 1 and 3.

Blood Pressure Levels

The dataset contains 1,100 blood pressure readings, with an average score of approximately 2.18 and a standard deviation of 0.83. The scores range from 1 to 3, with most individuals scoring either 1 or 2.

Sleep Quality

For sleep quality, the dataset shows 1,100 instances, with an average score of about 2.66 and a standard deviation of 1.55. Sleep quality scores range from 0 to 5, and most individuals scored between 1 and 4.

Breathing Problems

The dataset includes 1,100 reports on breathing problems, revealing an average score of approximately 2.75 and a standard deviation of 1.40. Scores range from 0 to 5, with most individuals reporting scores between 2 and 4.

Noise Levels

Regarding noise levels, there are 1,100 entries in the dataset, with an average score of about 2.65 and a standard deviation of 1.33. The noise level scores range from 0 to 5, with the majority falling between 2 and 3.

Living Conditions

The dataset contains 1,100 instances of living conditions, showing an average score of approximately 2.52 with a standard deviation of 1.12. Scores range from 0 to 5, with most individuals reporting either a score of 2 or 3.

Safety Levels

There are 1,100 records of safety levels, with an average score of about 2.78 and a standard deviation of 1.51. The scores range from 0 to 5, with most responses clustered between 2 and 4.

Basic Needs Satisfaction

Finally, the dataset includes 1,100 instances regarding satisfaction with basic needs, revealing an average score of approximately 2.77 and a standard deviation of 1.43. Scores range from 0 to 5, with most individuals scoring between 2 and 4.

Data preprocessing

- Checking for missing values

```
# Check for missing values
missing_values = df.isnull().sum() # This creates a Series containing the count of missing values per column
print("Missing values per column:") # Print a message indicating that missing values will be displayed
print(missing_values) # Print missing values per column
```

```
Missing values per column:
anxiety_level                 0
self_esteem                   0
mental_health_history         0
depression                    0
headache                      0
blood_pressure                0
sleep_quality                 0
breathing_problem             0
noise_level                   0
living_conditions             0
safety                        0
basic_needs                   0
academic_performance          0
study_load                    0
teacher_student_relationship  0
future_career_concerns        0
social_support                0
peer_pressure                 0
extracurricular_activities    0
bullying                      0
stress_level                  0
dtype: int64
```

Description: Missing values in a dataset can arise due to incomplete data collection or errors in data entry. They pose challenges for analysis, as many algorithms cannot handle missing data directly. To address this, we checked each column for missing values using the isnull() function, which provided a count of missing entries in each attribute. Upon inspection, it was found that the dataset had no missing values in any column, ensuring it was complete, and no imputation or deletion of rows was necessary. This step confirms that the data is ready for further analysis without modifications for missing values.

- Handling Duplicates

```
# Handling Duplicates
num_duplicates = df.duplicated().sum()
print("Number of duplicate rows:", num_duplicates)

# Remove duplicates and save the cleaned dataset
df = df.drop_duplicates()

# Save after handling duplicates
df.to_csv('Cleaned_Dataset.csv', index=False)
```

```
Number of duplicate rows: 0
```

Description : Duplicate rows in a dataset occur when the same observation appears multiple times. These duplicates can introduce bias, inflate the dataset size unnecessarily, and lead to inaccurate analysis or misleading model results. In this step, we examined the dataset for duplicates using the duplicated() function, which checks for rows that are identical in all attributes. If duplicates were found, they were removed to ensure that each data point is unique. This process enhances the data set's quality and prevents redundancy. However, after running the check, no duplicate rows were found, so no further action was required.

- Detect Outliers

```
data = pd.read_csv('Cleaned_Dataset.csv')
import numpy as np

# Outlier handling using IQR method
outlier_threshold = 1.5

def count_outliers(column_data):
    q1 = np.percentile(column_data, 25)
    q3 = np.percentile(column_data, 75)
    iqr = q3 - q1
    upper_bound = q3 + outlier_threshold * iqr
    lower_bound = q1 - outlier_threshold * iqr
    outliers = (column_data > upper_bound) | (column_data < lower_bound)
    return sum(outliers)

# Select numeric columns
numeric_columns = data.select_dtypes(include=[np.number]).columns

# Detect outliers in each numeric column
outlier_counts = {}
total_rows_with_outliers = 0

for column in numeric_columns:
    outliers = count_outliers(data[column])
    outlier_counts[column] = outliers
    total_rows_with_outliers += outliers

# Print outlier summary
print("Outlier Counts:")
for column, count in outlier_counts.items():
    print(f"{column}: {count} rows with outliers")
```

```
Outlier Counts:
anxiety_level: 0 rows with outliers
self_esteem: 0 rows with outliers
mental_health_history: 0 rows with outliers
depression: 0 rows with outliers
headache: 0 rows with outliers
blood_pressure: 0 rows with outliers
sleep_quality: 0 rows with outliers
breathing_problem: 0 rows with outliers
noise_level: 0 rows with outliers
living_conditions: 0 rows with outliers
safety: 0 rows with outliers
basic_needs: 0 rows with outliers
academic_performance: 0 rows with outliers
study_load: 0 rows with outliers
teacher_student_relationship: 0 rows with outliers
future_career_concerns: 0 rows with outliers
social_support: 0 rows with outliers
peer_pressure: 0 rows with outliers
extracurricular_activities: 0 rows with outliers
bullying: 0 rows with outliers
stress_level: 0 rows with outliers
Total Rows with Outliers: 0
```

Description : Outliers are data points that deviate significantly from the rest of the dataset. They can distort statistical measures and influence machine learning models, leading to biased outcomes. To address outliers, we applied the Interquartile Range (IQR) method, which defines a range where most data points lie (between the first and third quartiles). Any values outside this range are flagged as potential outliers. Instead of removing outliers, we chose to cap their values to reduce their impact while preserving the dataset's overall structure. After applying this technique, we verified that no significant outliers existed in the dataset, ensuring its suitability for further analysis.

- Data Transformation
  - Normalization

```
data = pd.DataFrame(data)
# Columns to normalize
columns_to_normalize = [
    'anxiety_level', 'self_esteem', 'depression', 'blood_pressure',
    'sleep_quality', 'breathing_problem', 'noise_level', 'living_conditions',
    'study_load', 'future_career_concerns', 'social_support', 'peer_pressure',
    'extracurricular_activities', 'bullying', 'stress_level'
]

# Apply Decimal scaling normalization
for column in columns_to_normalize:
    max_abs_value = data[column].abs().max()
    data[column] = data[column] / (10 ** len(str(int(max_abs_value))))

# Output the normalized data
```

```
   anxiety_level  self_esteem  mental_health_history  depression  headache  \
0           0.14         0.20                      0        0.11         2
1           0.15         0.08                      1        0.15         5
2           0.12         0.18                      1        0.14         2
3           0.16         0.12                      1        0.15         4
4           0.16         0.28                      0        0.07         2

   blood_pressure  sleep_quality  breathing_problem  noise_level  \
0             0.1            0.2                0.4          0.2
1             0.3            0.1                0.4          0.3
2             0.1            0.2                0.2          0.2
3             0.3            0.1                0.3          0.4
4             0.3            0.5                0.1          0.3

   living_conditions  ...  basic_needs  academic_performance  study_load  \
0                0.3  ...            2                     3         0.2
1                0.1  ...            2                     1         0.4
2                0.2  ...            2                     2         0.3
3                0.2  ...            2                     2         0.4
4                0.2  ...            3                     4         0.3

   teacher_student_relationship  future_career_concerns  social_support  \
0                             3                     0.3             0.2
1                             1                     0.5             0.1
2                             3                     0.2             0.2
3                             1                     0.4             0.1
4                             1                     0.2             0.1

   peer_pressure  extracurricular_activities  bullying  stress_level
0            0.3                        0.3       0.2           0.1
1            0.4                        0.5       0.5           0.2
2            0.3                        0.2       0.2           0.1
3            0.4                        0.4       0.5           0.2
4            0.5                        0.0       0.5           0.1

[5 rows x 21 columns]
```

Description : Normalization is the process of scaling numerical features to a specific range (e.g., 0 to 1). This is crucial when datasets contain variables with varying scales, as larger magnitudes can disproportionately influence machine learning models. In this step, we normalized selected numerical attributes using decimal scaling, which divides each value by a power of 10 depending on the largest value in the column. For instance, attributes like blood_pressure, anxiety_level, and depression were normalized. This process ensures that all features contribute equally to the analysis or model, leading to more balanced and accurate results.

- Aggregation

```
# Step 5: Aggregation based on stress_level
aggregated_df = data.groupby('stress_level').agg({
    'anxiety_level': 'mean',
    'depression': 'mean',
    'self_esteem': 'mean',
    'bullying': 'sum'  # Example of sum for categorical variables
})

# Output aggregated data
print("Aggregated data:")
print(aggregated_df)
```

```
Aggregated data:
              anxiety_level  depression  self_esteem  bullying
stress_level
0.0                0.054316    0.060134     0.252520      46.8
0.1                0.114302    0.118743     0.192626      91.5
0.2                0.164011    0.198293     0.087805     149.6
```

Description : Aggregation is a summarization technique that groups data based on specific attributes and calculates metrics like averages or totals. This method provides insights into patterns within the data. For example, we grouped the dataset by stress_level and calculated the mean for numerical features like anxiety_level and depression. For categorical attributes such as bullying, we computed the total number of occurrences in each group. Aggregation helped identify trends, such as how anxiety_level or depression varied across different stress levels. These summaries provided valuable insights for understanding relationships between variables.

- Discretization

```
# Discretization of anxiety_level into categories
data['anxiety_level'] = pd.cut(data['anxiety_level'], bins=3, labels=['Low', 'Medium', 'High'])

# Save the discretized dataset
data.to_csv('Cleaned_Dataset.csv', index=False)

# Display the first few rows
print("Data after discretization:")
print(data[['anxiety_level']].head())
```

```
Data after discretization:
  anxiety_level
0          High
1          High
2        Medium
3          High
4          High
```

Description : Discretization is the process of converting continuous numerical attributes into discrete categories or bins. This helps simplify the interpretation of data and facilitates categorical analysis. For example, the anxiety_level column, which originally contained continuous numerical values, was discretized into three categori es: Low, Medium, and High. This was achieved by dividing the range of values into three equal intervals. Discretization enables easier comparison between groups and provides a more intuitive understanding of the distribution of anxiety levels within the dataset.

- Feature Selection

```
from sklearn.feature_selection import SelectKBest, f_classif

# Define the target column (class label)
class_label = 'stress_level'
# Separate features from target variable
X = df.drop(columns=[class_label])  # Features
y = df[class_label]  # The target column y should be the class label
# Check the number of features
n_features = X.shape[1] # The number of columns in X represents the number of features
print(' Number of features available :', n_features) # Print the number of features

# Specify the number of features to choose
num_features_to_select = min(5, n_features)  # Choose the least between 5 and the actual number of features
selector = SelectKBest(score_func=f_classif, k=num_features_to_select) # Set up SelectKBest with appropriate function and k
X_selected = selector.fit_transform(X, y)  # Apply feature selection to the data

# Get selected feature indicators
selected_indices = selector.get_support(indices=True) # Get indicators of selected features

# Get selected feature names
selected_features = X.columns[selected_indices] # Use pointers to get the names of the selected features

print('Selected Features:', selected_features) # Print the names of the selected features
```

```
 Number of features available : 20
Selected Features: Index(['self_esteem', 'blood_pressure', 'sleep_quality',
       'future_career_concerns', 'bullying'],
      dtype='object')
```

Description : Feature selection is a technique used to identify the most relevant attributes for a specific analysis or model. By reducing the number of features, we can enhance model performance, reduce complexity, and prevent overfitting. In this step, we applied the SelectKBest method with the f_classif scoring function to select the top five features most correlated with the target variable (stress_level). The selected features—self_esteem, blood_pressure, sleep_quality, future_career_concerns, and bullying—were identified as the most influential for predicting stress_level. This focused approach allows for more efficient and accurate analysis.

Data after processing

```
#Load data
import pandas as pd
data=pd.read_csv('Cleaned_Dataset.csv')
print(data)
```

```
     anxiety_level  self_esteem  mental_health_history  depression  headache  \
0             High         0.20                      0        0.11         2
1             High         0.08                      1        0.15         5
2           Medium         0.18                      1        0.14         2
3             High         0.12                      1        0.15         4
4             High         0.28                      0        0.07         2
...            ...          ...                    ...         ...       ...
1095        Medium         0.17                      0        0.14         3
1096        Medium         0.12                      0        0.08         0
1097           Low         0.26                      0        0.03         1
1098          High         0.00                      1        0.19         5
1099          High         0.06                      1        0.15         3

      blood_pressure  sleep_quality  breathing_problem  noise_level  \
0                0.1            0.2                0.4         0.20
1                0.3            0.1                0.4         0.30
2                0.1            0.2                0.2         0.20
3                0.3            0.1                0.3         0.40
4                0.3            0.5                0.1         0.30
...              ...            ...                ...          ...
1095             0.1            0.3                0.2         0.20
1096             0.3            0.0                0.0         0.05
1097             0.2            0.5                0.2         0.20
1098             0.3            0.1                0.4         0.30
1099             0.3            0.0                0.3         0.30

      living_conditions  ...  basic_needs  academic_performance  study_load  \
0                  0.30  ...            2                     3        0.20
1                  0.10  ...            2                     1        0.40
2                  0.20  ...            2                     2        0.30
3                  0.20  ...            2                     2        0.40
4                  0.20  ...            3                     4        0.30
...                 ...  ...          ...                   ...         ...
1095               0.20  ...            3                     2        0.20
1096               0.10  ...            4                     0        0.10
1097               0.30  ...            4                     5        0.10
1098               0.10  ...            1                     2        0.45
1099               0.05  ...            3                     3        0.40

      teacher_student_relationship  future_career_concerns  social_support  \
0                                3                     0.3             0.2
1                                1                     0.5             0.1
2                                3                     0.2             0.2
3                                1                     0.4             0.1
4                                1                     0.2             0.1
...                            ...                     ...             ...
1095                             2                     0.3             0.3
1096                             1                     0.1             0.1
1097                             4                     0.1             0.3
1098                             1                     0.4             0.1
1099                             3                     0.3             0.1

      peer_pressure  extracurricular_activities  bullying  stress_level
0               0.3                        0.3       0.2           0.1
1               0.4                        0.5       0.5           0.2
2               0.3                        0.2       0.2           0.1
3               0.4                        0.4       0.5           0.2
4               0.5                        0.0       0.5           0.1
...             ...                        ...       ...           ...
1095            0.2                        0.3       0.3           0.1
1096            0.3                        0.4       0.3           0.2
1097            0.1                        0.2       0.1           0.0
1098            0.4                        0.4       0.4           0.2
1099            0.5                        0.1       0.4           0.2

[1100 rows x 21 columns]
```

Data mining Technique:

Classification:

- **Technique:** Decision Trees
- **Why:** Decision trees are intuitive and can handle both numerical and categorical data, making them suitable for many classification tasks.
- **How:** The DecisionTreeClassifier from the Scikit-learn package in Python. This package offers efficient implementation and various parameters for tuning the model.

Clustering:

- **Technique:** K-Means Clustering
- **Why:** K-Means clustering is widely used for exploratory data analysis and identifying patterns in unlabeled datasets.
- **How:** The KMeans module from Scikit-learn will be applied, allowing the specification of the number of clusters. To determine the optimal number of clusters, the **Elbow Method** will assess the reduction in within-cluster variance, while the **Silhouette Method** will evaluate clustering quality to ensure well-separated groupings.

Evaluation and Comparison

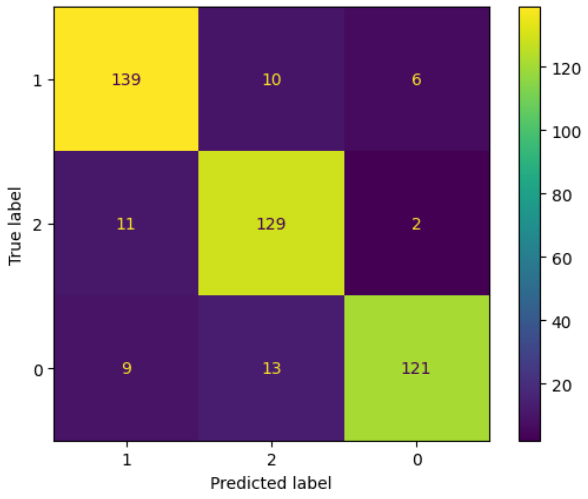
- Classification

<table>
<tr><td colspan="2">Small (60% train, 40% test)</td></tr>
<tr><td>Entropy</td><td>

```
# Encoding the 'anxiety_level' column
le = LabelEncoder()
data['anxiety_level'] = le.fit_transform(data['anxiety_level'])

# Feature Selection
fn1 = data.keys().tolist()[:-1]
X1 = data[fn1]
y1 = data['stress_level']

# Encode target variable to numerical values
le1 = LabelEncoder()
y_encoded1 = le1.fit_transform(y1)

# Splitting Data into 60% training and 40% testing subsets
X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, y_encoded1, test_size=0.4, random_state=1)

# Building decision tree model
clf1 = DecisionTreeClassifier(criterion='entropy', random_state=1)

# Train Decision Tree Classifier
clf1 = clf1.fit(X_train1, y_train1)

# Predict the response for test dataset
y_pred1 = clf1.predict(X_test1)
```

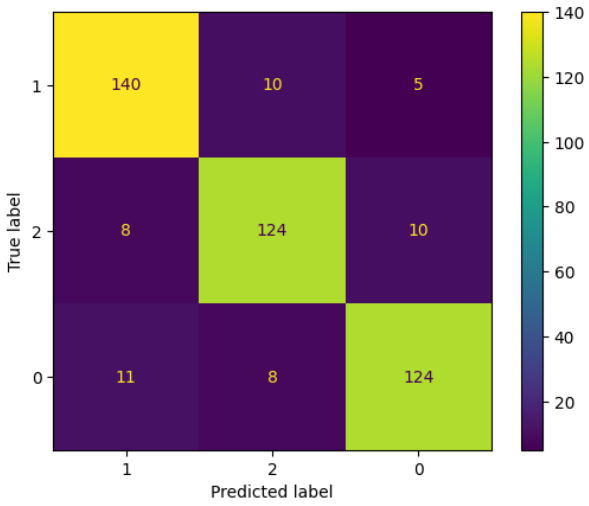
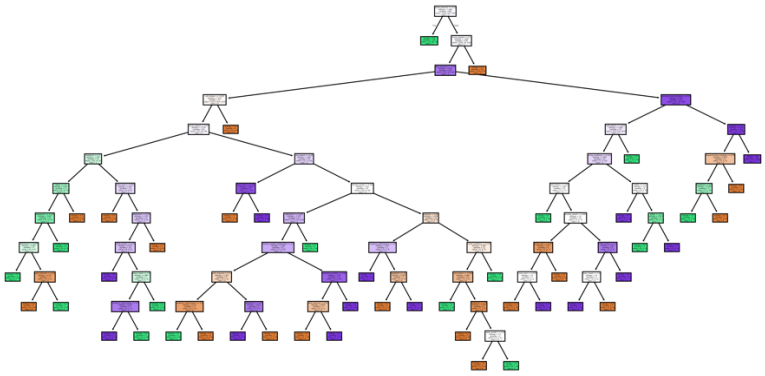
</td></tr>
<tr><td>Gini Index</td><td>

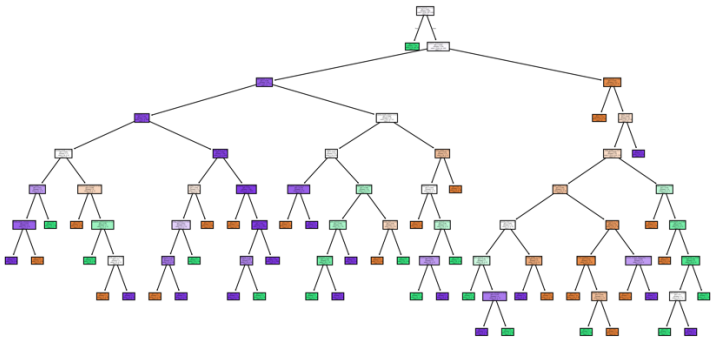
```
# Feature Selection
fn1 = data.keys().tolist()[:-1]  # Selecting columns from index 1 to the second-to-last column
X1 = data[fn1]
y1 = data['stress_level']

# Encode target variable to numerical values
le1 = LabelEncoder()
y_encoded1 = le1.fit_transform(y1)

# Splitting Data into 60% training and 40% testing subsets
X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, y_encoded1, test_size=0.4, random_state=1)

# Building decision tree model
clf1 = DecisionTreeClassifier(random_state=1)

# Train Decision Tree Classifier
clf1 = clf1.fit(X_train1, y_train1)

# Predict the response for test dataset
y_pred1 = clf1.predict(X_test1)
```

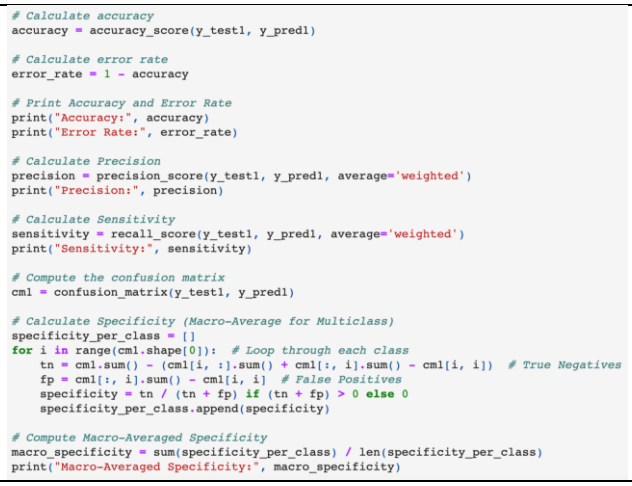
</td></tr>
<tr><td colspan="2">Confusion matrix</td></tr>
<tr><td>Entropy</td><td> 
</td></tr>
<tr><td>code</td><td>

```
# Compute and print the confusion matrix
cm1 = confusion_matrix(y_test1, y_pred1)
print(cm1)

from sklearn.metrics import ConfusionMatrixDisplay
cn1 = data['stress_level'].unique() # classes_names

class_labels1 = {0:'0', 1:'1', 2:'2'}

# create a confusion matrix display object
disp1 = ConfusionMatrixDisplay.from_estimator(clf1, X_test1, y_test1, display_labels=cn1)
```

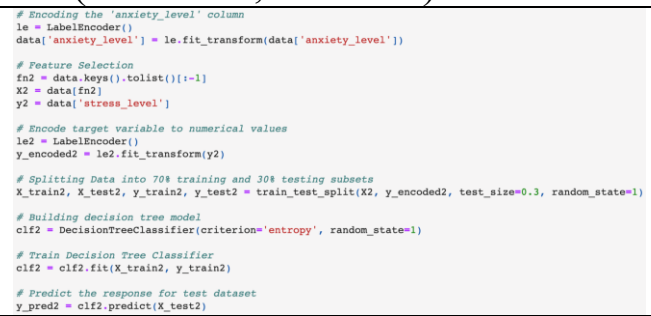
</td></tr>
</table>

Gini Index	
code	```# Compute and print the confusion matrix cm1 = confusion_matrix(y_test1, y_pred1) print(cm1) from sklearn.metrics import ConfusionMatrixDisplay cn1 = data['stress_level'].unique() # classes_names class_labels1 = {0:'0', 1:'1', 2:'2'} # create a confusion matrix display object disp1 = ConfusionMatrixDisplay.from_estimator(clf1, X_test1, y_test1, display_labels=cn1)```
Decision Tree	
Entropy	
code	```# Convert all feature columns to numeric types if applicable for col in fn1: data[col] = pd.to_numeric(data[col], errors='coerce') # Re-encode the target variable as strings for compatibility with plot_tree le1 = LabelEncoder() y1 = le1.fit_transform(data['stress_level']) class_names1 = [str(class_label) for class_label in le1.classes_] # Split dataset into features and target variable X1 = data[fn1] # Features y1 = le1.fit_transform(data['stress_level']) # Target encoded as numbers for fitting # Split data into training and test sets X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, y1, test_size=0.4, random_state=1) # Train Decision Tree Classifier clf1 = DecisionTreeClassifier(criterion="entropy") clf1.fit(X_train1, y_train1) # Set up the plot fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(16, 8), dpi=600) # Plot the decision tree tree.plot_tree(clf1, feature_names=fn1, # Ensure feature names are strings class_names=class_names1, # Use encoded class names as strings filled=True,) # Display the plot plt.show()```

Gini Index	
code	```# Convert all feature columns to numeric types if applicable for col in fn1: data[col] = pd.to_numeric(data[col], errors='coerce') # Re-encode the target variable as strings for compatibility with plot_tree le1 = LabelEncoder() y1 = le1.fit_transform(data['stress_level']) class_names1 = [str(class_label) for class_label in le1.classes_] # Split dataset into features and target variable X1 = data[fn1] # Features y1 = le1.fit_transform(data['stress_level']) # Target encoded as numbers for fitting # Split data into training and test sets X_train1, X_test1, y_train1, y_test1 = train_test_split(X1, y1, test_size=0.4, random_state=1) # Train Decision Tree Classifier clf1 = DecisionTreeClassifier() clf1.fit(X_train1, y_train1) # Set up the plot fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(16, 8), dpi=600) # Plot the decision tree tree.plot_tree(clf1, feature_names=fn1, # Ensure feature names are strings class_names=class_names1, # Use encoded class names as strings filled=True) # Display the plot plt.show()```
Entropy	Accuracy: 0.884090909090909 Error Rate: 0.11590909090909096 Precision: 0.8867001943058211 Sensitivity: 0.884090909090909 Macro-Averaged Specificity: 0.9419024421379302
code	```# Calculate accuracy accuracy = accuracy_score(y_test1, y_pred1) # Calculate error rate error_rate = 1 - accuracy # Evaluate model performance print("Accuracy:", accuracy) print("Error Rate:", error_rate) # Calculate Precision precision = precision_score(y_test1, y_pred1, average='weighted') print("Precision:", precision) # Calculate Sensitivity sensitivity = recall_score(y_test1, y_pred1, average='weighted') print("Sensitivity:", sensitivity) cm1 = confusion_matrix(y_test1, y_pred1) # Calculate Specificity (Macro-Average for Multiclass) specificity_per_class = [] for i in range(cm1.shape[0]): # Loop through each class tn = cm1.sum() - (cm1[i, :].sum() + cm1[:, i].sum() - cm1[i, i]) # True Negatives fp = cm1[:, i].sum() - cm1[i, i] # False Positives specificity = tn / (tn + fp) if (tn + fp) > 0 else 0 specificity_per_class.append(specificity) # Compute Macro-Averaged Specificity macro_specificity = sum(specificity_per_class) / len(specificity_per_class) print("Macro-Averaged Specificity:", macro_specificity)```
Gini Index	Accuracy: 0.8818181818181818 Error Rate: 0.11818181818181817 Precision: 0.8819234835116471 Sensitivity: 0.8818181818181818 Macro-Averaged Specificity: 0.9408085327548416

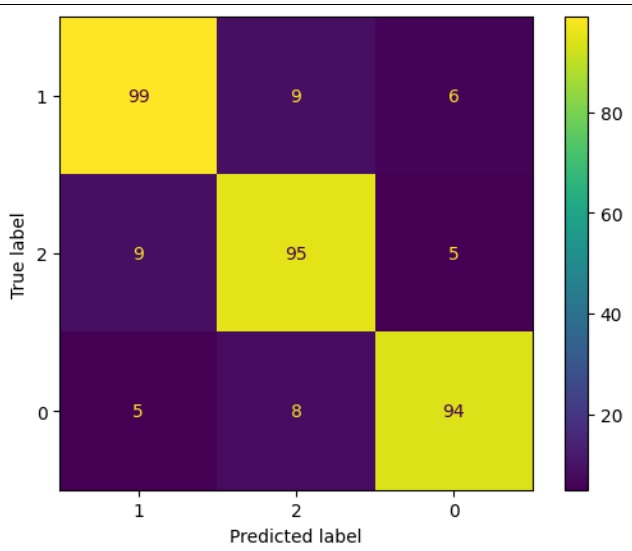
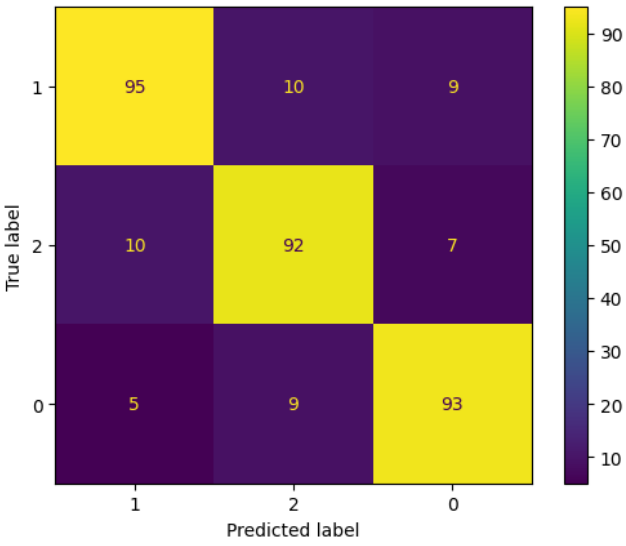
code	

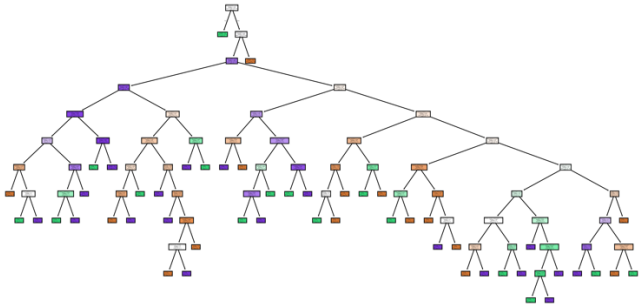
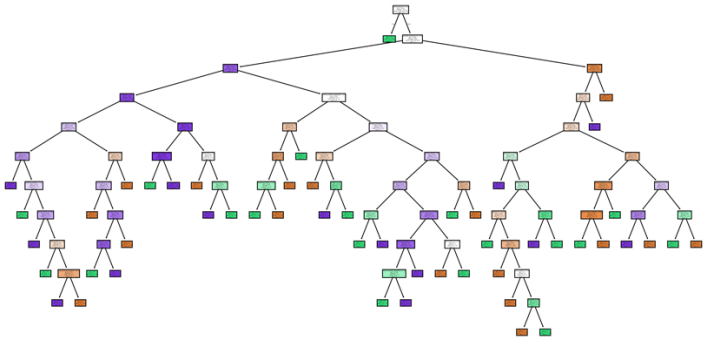
```python
# Calculate accuracy
accuracy = accuracy_score(y_test1, y_pred1)

# Calculate error rate
error_rate = 1 - accuracy

# Print Accuracy and Error Rate
print("Accuracy:", accuracy)
print("Error Rate:", error_rate)

# Calculate Precision
precision = precision_score(y_test1, y_pred1, average='weighted')
print("Precision:", precision)

# Calculate Sensitivity
sensitivity = recall_score(y_test1, y_pred1, average='weighted')
print("Sensitivity:", sensitivity)

# Compute the confusion matrix
cm1 = confusion_matrix(y_test1, y_pred1)

# Calculate Specificity (Macro-Average for Multiclass)
specificity_per_class = []
for i in range(cm1.shape[0]):  # Loop through each class
    tn = cm1.sum() - (cm1[i, :].sum() + cm1[:, i].sum() - cm1[i, i])  # True Negatives
    fp = cm1[:, i].sum() - cm1[i, i]  # False Positives
    specificity = tn / (tn + fp) if (tn + fp) > 0 else 0
    specificity_per_class.append(specificity)

# Compute Macro-Averaged Specificity
macro_specificity = sum(specificity_per_class) / len(specificity_per_class)
print("Macro-Averaged Specificity:", macro_specificity)
```

Medium (70% train, 30% test)	
Entropy	
Gini Index	
Confusion matrix	

```python
# Encoding the 'anxiety_level' column
le = LabelEncoder()
data['anxiety_level'] = le.fit_transform(data['anxiety_level'])

# Feature Selection
fn2 = data.keys().tolist()[:-1]
X2 = data[fn2]
y2 = data['stress_level']

# Encode target variable to numerical values
le2 = LabelEncoder()
y_encoded2 = le2.fit_transform(y2)

# Splitting Data into 70% training and 30% testing subsets
X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y_encoded2, test_size=0.3, random_state=1)

# Building decision tree model
clf2 = DecisionTreeClassifier(criterion='entropy', random_state=1)

# Train Decision Tree Classifier
clf2 = clf2.fit(X_train2, y_train2)

# Predict the response for test dataset
y_pred2 = clf2.predict(X_test2)
```

```python
# Feature Selection
fn2 = data.keys().tolist()[:-1]
X2 = data[fn2]
y2 = data['stress_level']

# Encode target variable to numerical values
le2 = LabelEncoder()
y_encoded2 = le2.fit_transform(y2)

# Splitting Data into 70% training and 30% testing subsets
X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y_encoded2, test_size=0.3, random_state=1)

# Building decision tree model
clf2 = DecisionTreeClassifier(random_state=1)

# Train Decision Tree Classifier
clf2 = clf2.fit(X_train2, y_train2)

# Predict the response for test dataset
y_pred2 = clf2.predict(X_test2)
```

Entropy	
code	`# Compute and print the confusion matrix` `cm2 = confusion_matrix(y_test2, y_pred2)` `print(cm2)` `from sklearn.metrics import ConfusionMatrixDisplay` `cn2 = data['stress_level'].unique() # classes_names` `class_labels2 = {0:'0', 1:'1', 2:'2'}` `# create a confusion matrix display object` `disp2 = ConfusionMatrixDisplay.from_estimator(clf2, X_test2, y_test2, display_labels=cn2)`
Gini Index	
code	`# Compute and print the confusion matrix` `cm2 = confusion_matrix(y_test2, y_pred2)` `print(cm2)` `from sklearn.metrics import ConfusionMatrixDisplay` `cn2 = data['stress_level'].unique() # classes_names` `class_labels2 = {0:'0', 1:'1', 2:'2'}` `# create a confusion matrix display object` `disp2 = ConfusionMatrixDisplay.from_estimator(clf2, X_test2, y_test2, display_labels=cn2)`
Decision Tree	

Entropy	
code	`# Convert all feature columns to numeric types if applicable` `for col in fn2:` `  data[col] = pd.to_numeric(data[col], errors='coerce')` `# Re-encode the target variable as strings for compatibility with plot_tree` `le2 = LabelEncoder()` `y2 = le2.fit_transform(data['stress_level'])` `class_names2 = [str(class_label) for class_label in le2.classes_]` `# Split dataset into features and target variable` `X2 = data[fn2] # Features` `y2 = le2.fit_transform(data['stress_level'])  # Target encoded as numbers for fitting` `# Split data into training and test sets` `X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y2, test_size=0.3, random_state=1)` `# Train Decision Tree Classifier` `clf2 = DecisionTreeClassifier(criterion="entropy", random_state=1)` `clf2.fit(X_train2, y_train2)` `# Set up the plot` `fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(16, 8), dpi=600)` `# Plot the decision tree` `tree.plot_tree(` `  clf2,` `  feature_names=fn2,  # Ensure feature names are strings` `  class_names=class_names2,  # Use encoded class names as strings` `  filled=True` `)` `# Display the plot` `plt.show()`
Gini Index	
code	`# Convert all feature columns to numeric types if applicable` `for col in fn2:` `  data[col] = pd.to_numeric(data[col], errors='coerce')` `# Re-encode the target variable as strings for compatibility with plot_tree` `le2 = LabelEncoder()` `y2 = le2.fit_transform(data['stress_level'])` `class_names2 = [str(class_label) for class_label in le2.classes_]` `# Split dataset into features and target variable` `X2 = data[fn2] # Features` `y2 = le2.fit_transform(data['stress_level'])  # Target encoded as numbers for fitting` `# Split data into training and test sets` `X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y2, test_size=0.3, random_state=1)` `# Train Decision Tree Classifier` `clf2 = DecisionTreeClassifier()` `clf2.fit(X_train2, y_train2)` `# Set up the plot` `fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(16, 8), dpi=600)` `# Plot the decision tree` `tree.plot_tree(` `  clf2,` `  feature_names=fn2,  # Ensure feature names are strings` `  class_names=class_names2,  # Use encoded class names as strings` `  filled=True` `)` `# Display the plot` `plt.show()`
	Evaluation metrics
Entropy	Accuracy: 0.8727272727272727 Error Rate: 0.12727272727272732 Precision: 0.8730967864485565 Sensitivity: 0.8727272727272727 Macro-Averaged Specificity: 0.9363115846673397

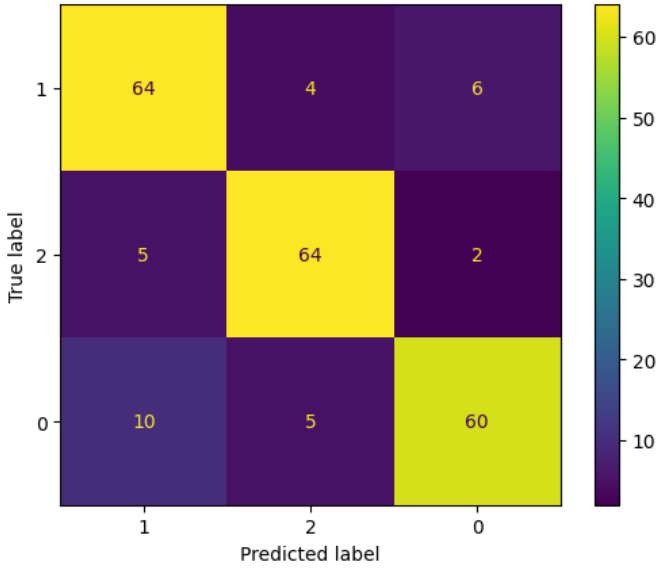
code	```# Calculate accuracy accuracy = accuracy_score(y_test2, y_pred2) # Calculate error rate error_rate = 1 - accuracy # Evaluate model performance print("Accuracy:", accuracy) print("Error Rate:", error_rate) # Calculate Precision precision = precision_score(y_test2, y_pred2, average='weighted') print("Precision:", precision) # Calculate Sensitivity sensitivity = recall_score(y_test2, y_pred2, average='weighted') print("Sensitivity:", sensitivity) # Compute the confusion matrix cm2 = confusion_matrix(y_test2, y_pred2) # Calculate Specificity (Macro-Average for Multiclass) specificity_per_class = [] for i in range(cm2.shape[0]): # Loop through each class tn = cm2.sum() - (cm2[i, :].sum() + cm2[:, i].sum() - cm2[i, i]) # True Negatives fp = cm2[:, i].sum() - cm2[i, i] # False Positives specificity = tn / (tn + fp) if (tn + fp) > 0 else 0 specificity_per_class.append(specificity) # Compute Macro-Averaged Specificity macro_specificity = sum(specificity_per_class) / len(specificity_per_class) print("Macro-Averaged Specificity:", macro_specificity)```
Gini Index	Accuracy: 0.8484848484848485 Error Rate: 0.1515151515151515 Precision: 0.8487589872076945 Sensitivity: 0.8484848484848485 Macro-Averaged Specificity: 0.9242779419843519
code	```# Calculate accuracy accuracy = metrics.accuracy_score(y_test2, y_pred2) # Calculate error rate error_rate = 1 - accuracy # Evaluate model performance print("Accuracy:", accuracy) print("Error Rate:", error_rate) # Calculate Precision precision = metrics.precision_score(y_test2, y_pred2, average='weighted') print("Precision:", precision) # Calculate Sensitivity sensitivity = metrics.recall_score(y_test2, y_pred2, average='weighted') print("Sensitivity:", sensitivity) # Compute the confusion matrix cm = metrics.confusion_matrix(y_test2, y_pred2) # Calculate Specificity (Macro-Average for Multiclass) specificity_per_class = [] for i in range(cm.shape[0]): # Loop through each class tn = cm.sum() - (cm[i, :].sum() + cm[:, i].sum() - cm[i, i]) # True Negatives fp = cm[:, i].sum() - cm[i, i] # False Positives specificity = tn / (tn + fp) if (tn + fp) > 0 else 0 specificity_per_class.append(specificity) # Compute Macro-Averaged Specificity macro_specificity = sum(specificity_per_class) / len(specificity_per_class) print("Macro-Averaged Specificity:", macro_specificity)```

Large (80% train, 20% test)	
Entropy	```# Encoding the 'anxiety_level' column le = LabelEncoder() data['anxiety_level'] = le.fit_transform(data['anxiety_level']) # Feature Selection fn3 = data.keys().tolist()[:-1] # Selecting columns from index 1 to the second-to-last column X3 = data[fn3] y3 = data['stress_level'] # Encode target variable to numerical values le3 = LabelEncoder() y_encoded3 = le3.fit_transform(y3) # Splitting Data into 80% training and 20% testing subsets X_train3, X_test3, y_train3, y_test3 = train_test_split(X3, y_encoded3, test_size=0.2, random_state=1) # Building decision tree model clf3 = DecisionTreeClassifier(criterion='entropy', random_state=1) # Train Decision Tree Classifier clf3 = clf3.fit(X_train3, y_train3) # Predict the response for test dataset y_pred3 = clf3.predict(X_test3)```

Gini Index	

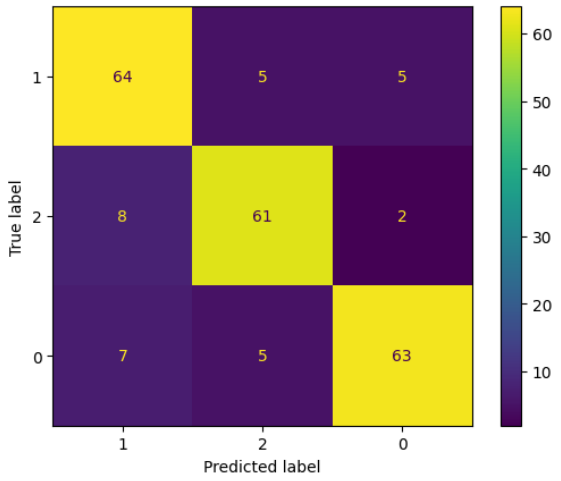
```python
# Feature Selection
fn3 = data.keys().tolist()[:-1]
X3 = data[fn3]
y3 = data['stress_level']

# Encode target variable to numerical values
le3 = LabelEncoder()
y_encoded3 = le3.fit_transform(y3)

# Splitting Data into 80% training and 20% testing subsets
X_train3, X_test3, y_train3, y_test3 = train_test_split(X3, y_encoded3, test_size=0.2, random_state=1)

# Building decision tree model
clf3 = DecisionTreeClassifier(random_state=1)

# Train Decision Tree Classifier
clf3 = clf3.fit(X_train3, y_train3)

# Predict the response for test dataset
y_pred3 = clf3.predict(X_test3)
```

Confusion matrix

Entropy	

code	

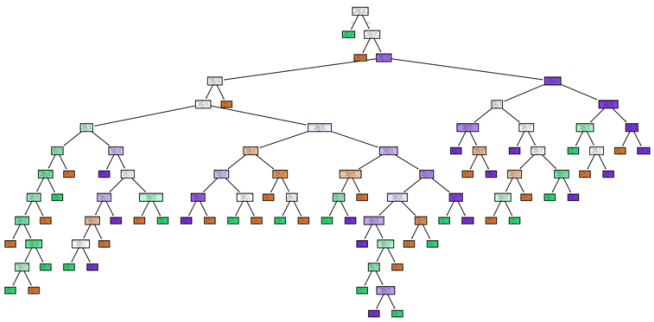
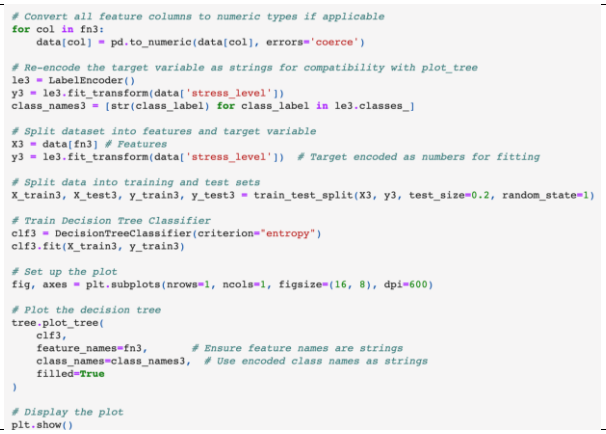
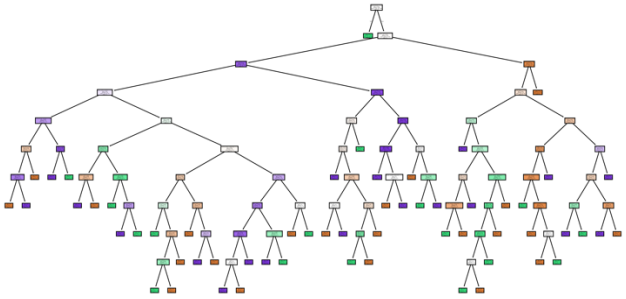
```python
# Compute and print the confusion matrix
cm3 = confusion_matrix(y_test3, y_pred3)
print(cm3)

from sklearn.metrics import ConfusionMatrixDisplay
cn3 = data['stress_level'].unique() # classes_names

class_labels3 = {0:'0', 1:'1', 2:'2'}

# create a confusion matrix display object
disp = ConfusionMatrixDisplay.from_estimator(clf3, X_test3, y_test3, display_labels=cn3)
```

Gini Index	

code	

```python
# Compute and print the confusion matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)

from sklearn.metrics import ConfusionMatrixDisplay
cn = data['stress_level'].unique() # classes_names

class_labels = {0:'0', 1:'1', 2:'2'}

# create a confusion matrix display object
disp = ConfusionMatrixDisplay.from_estimator(clf, X_test, y_test, display_labels=cn)
```

Decision Tree

Entropy	
code	 ```# Convert all feature columns to numeric types if applicable for col in fn3: data[col] = pd.to_numeric(data[col], errors='coerce') # Re-encode the target variable as strings for compatibility with plot_tree le3 = LabelEncoder() y3 = le3.fit_transform(data['stress_level']) class_names3 = [str(class_label) for class_label in le3.classes_] # Split dataset into features and target variable X3 = data[fn3] # Features y3 = le3.fit_transform(data['stress_level']) # Target encoded as numbers for fitting # Split data into training and test sets X_train3, X_test3, y_train3, y_test3 = train_test_split(X3, y3, test_size=0.2, random_state=1) # Train Decision Tree Classifier clf3 = DecisionTreeClassifier(criterion="entropy") clf3.fit(X_train3, y_train3) # Set up the plot fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(16, 8), dpi=600) # Plot the decision tree tree.plot_tree(clf3, feature_names=fn3, # Ensure feature names are strings class_names=class_names3, # Use encoded class names as strings filled=True) # Display the plot plt.show()```
Gini Index	
code	```# Convert all feature columns to numeric types if applicable for col in fn3: data[col] = pd.to_numeric(data[col], errors='coerce') # Re-encode the target variable as strings for compatibility with plot_tree le3 = LabelEncoder() y3 = le3.fit_transform(data['stress_level']) class_names3 = [str(class_label) for class_label in le3.classes_] # Split dataset into features and target variable X3 = data[fn3] # Features y3 = le3.fit_transform(data['stress_level']) # Target encoded as numbers for fitting # Split data into training and test sets X_train3, X_test3, y_train3, y_test3 = train_test_split(X3, y3, test_size=0.2, random_state=1) # Train Decision Tree Classifier clf3 = DecisionTreeClassifier() clf3.fit(X_train3, y_train3) # Set up the plot fig, axes = plt.subplots(nrows=1, ncols=1, figsize=(16, 8), dpi=600) # Plot the decision tree tree.plot_tree(clf3, feature_names=fn3, # Ensure feature names are strings class_names=class_names3, # Use encoded class names as strings filled=True) # Display the plot plt.show()```
Evaluation metrics	
Entropy	Accuracy: 0.8545454545454545 Error Rate: 0.1454545454545455 Precision: 0.8562382409968574 Sensitivity: 0.8545454545454545 Macro-Averaged Specificity: 0.9272283918719136

code		```# Calculate accuracy accuracy = accuracy_score(y_test3, y_pred3) # Calculate error rate error_rate = 1 - accuracy # Evaluate model performance print("Accuracy:", accuracy) print("Error Rate:", error_rate) # Calculate Precision precision = precision_score(y_test3, y_pred3, average='weighted') print("Precision:", precision) # Calculate Sensitivity sensitivity = recall_score(y_test3, y_pred3, average='weighted') print("Sensitivity:", sensitivity) # Compute the confusion matrix cm3 = confusion_matrix(y_test3, y_pred3) # Calculate Specificity (Macro-Average for Multiclass) specificity_per_class = [] for i in range(cm3.shape[0]): # Loop through each class tn = cm3.sum() - (cm3[i, :].sum() + cm3[:, i].sum() - cm3[i, i]) # True Negatives fp = cm3[:, i].sum() - cm3[i, i] # False Positives specificity = tn / (tn + fp) if (tn + fp) > 0 else 0 specificity_per_class.append(specificity) # Compute Macro-Averaged Specificity macro_specificity = sum(specificity_per_class) / len(specificity_per_class) print("Macro-Averaged Specificity:", macro_specificity)```
Gini Index		Accuracy: 0.8545454545454545 Error Rate: 0.1454545454545455 Precision: 0.8565880322209436 Sensitivity: 0.8545454545454545 Macro-Averaged Specificity: 0.927290105981302
code		```# Calculate accuracy accuracy = metrics.accuracy_score(y_test3, y_pred3) # Calculate error rate error_rate = 1 - accuracy # Evaluate model performance print("Accuracy:", accuracy) print("Error Rate:", error_rate) # Calculate Precision precision = metrics.precision_score(y_test3, y_pred3, average='weighted') print("Precision:", precision) # Calculate Sensitivity sensitivity = metrics.recall_score(y_test3, y_pred3, average='weighted') print("Sensitivity:", sensitivity) # Compute the confusion matrix cm3 = metrics.confusion_matrix(y_test3, y_pred3) # Calculate Specificity (Macro-Average for Multiclass) specificity_per_class = [] for i in range(cm3.shape[0]): # Loop through each class tn = cm3.sum() - (cm3[i, :].sum() + cm3[:, i].sum() - cm3[i, i]) # True Negatives fp = cm3[:, i].sum() - cm3[i, i] # False Positives specificity = tn / (tn + fp) if (tn + fp) > 0 else 0 specificity_per_class.append(specificity) # Compute Macro-Averaged Specificity macro_specificity = sum(specificity_per_class) / len(specificity_per_class) print("Macro-Averaged Specificity:", macro_specificity)```

	60% train, 40% test		70% train, 30% test		80% train, 20% test	
	Entropy	Gini Index	Entropy	Gini Index	Entropy	Gini Index
Accuracy	0.8840	0.8818	0.8727	0.8484	0.8545	0.8545
Error Rate	0.1159	0.1181	0.1272	0.1515	0.1454	0.1454
Precision	0.8867	0.8819	0.8730	0.8487	0.8562	0.8565
Sensitivity	0.8840	0.8818	0.8727	0.8484	0.8545	0.8545
Specificity	0.9419	0.9408	0.9363	0.9242	0.9272	0.9272

Partition: 60% Train, 40% Test

- **Accuracy**: **Entropy** (0.8840) outperforms **Gini Index** (0.8818).
- **Error Rate**: **Entropy** has a lower error rate (0.1159) compared to **Gini Index** (0.1181).
- **Precision**: **Entropy** performs slightly worse (0.8867) than **Gini Index** (0.8819).
- **Sensitivity**: **Entropy** (0.8840) performs better than **Gini Index** (0.8818).
- **Specificity**: **Entropy** (0.9419) slightly outperforms **Gini Index** (0.9408).

Best Algorithm: **Entropy**, as it performs better in Accuracy, Error Rate, Sensitivity, and Specificity.

Partition: 70% Train, 30% Test

- **Accuracy**: **Entropy** (0.8727) outperforms **Gini Index** (0.8484).
- **Error Rate**: **Entropy** has a lower error rate (0.1272) compared to **Gini Index** (0.1515).
- **Precision**: **Entropy** (0.8730) outperforms **Gini Index** (0.8487).
- **Sensitivity**: **Entropy** (0.8727) performs better than **Gini Index** (0.8484).
- **Specificity**: **Gini Index** (0.9242) performs better than **Entropy** (0.9363).

Best Algorithm: **Entropy**, as it consistently outperforms **Gini Index** in Accuracy, Error Rate, Precision, and Sensitivity.

Partition: 80% Train, 20% Test

- **Accuracy**: Both **Entropy** and **Gini Index** perform equally well (0.8545).
- **Error Rate**: Both algorithms have the same error rate (0.1454).
- **Precision**: **Gini Index** (0.8565) slightly outperforms **Entropy** (0.8562).
- **Sensitivity**: Both **Entropy** and **Gini Index** perform equally (0.8545).
- **Specificity**: Both **Entropy** and **Gini Index** perform equally (0.9272).

Best Algorithm: **Gini Index**, because it has a marginally higher Precision.

Considering the results across all partitions:

1. **Entropy** consistently outperforms **Gini Index** in **Accuracy**, **Error Rate**, **Precision**, and **Sensitivity** in the first two partitions (60%-40% and 70%-30%).
2. In the third partition (80%-20%), **Gini Index** has a slight edge in Precision, but the difference is negligible.

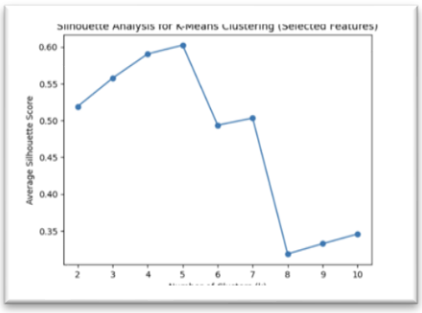
The best among all:

- **Entropy** is the best overall algorithm because it performs better in most partitions and metrics.

- Clustering
  - Silhouette method

code

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler
import pandas as pd

# Load and preprocess the dataset
df = pd.read_csv('../Dataset/Processed_dataset.csv')

# Define the selected features
features = ['self_esteem', 'blood_pressure', 'sleep_quality', 'future_career_concerns', 'bullying']

# Scale the selected features
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df[features])  # Scaling only the selected features

# Perform k-means clustering with different values of k
k_values = range(2, 11)
silhouette_avg_values = []  # List to store silhouette scores for each k

# Perform K-means clustering and calculate the average Silhouette score for each k
for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init='auto')  # Fit the K-means model with the current k
    kmeans_result = kmeans.fit_predict(df_scaled)  # Predict the clusters
    silhouette_avg = silhouette_score(df_scaled, kmeans_result)  # Calculate the average silhouette score
    silhouette_avg_values.append(silhouette_avg)  # Append the score to the list

# Find the best number of clusters based on the highest average Silhouette score
best_k = k_values[np.argmax(silhouette_avg_values)]  # k corresponding to the highest score
best_score = max(silhouette_avg_values)  # Highest silhouette score

# Find the second highest average Silhouette score and its corresponding number of clusters
silhouette_avg_values_sorted = sorted(silhouette_avg_values, reverse=True)
second_best_score = silhouette_avg_values_sorted[1]
second_best_k = k_values[silhouette_avg_values.index(second_best_score)]

# Plot the silhouette scores for different values of k
plt.plot(k_values, silhouette_avg_values, marker='o')
plt.title('Silhouette Analysis for K-Means Clustering (Selected Features)')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Average Silhouette Score')
plt.show()

# Print the highest and second-highest average silhouette scores and their corresponding k values
print(f"The highest average Silhouette score is {best_score} with k={best_k}.")
print(f"The second highest average Silhouette score is {second_best_score} with k={second_best_k}.")
```



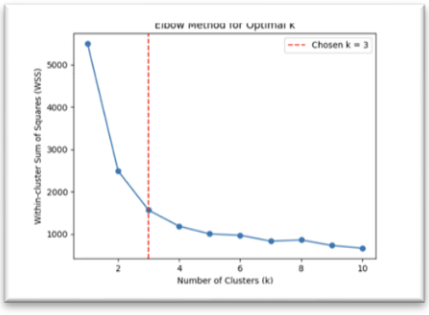
Description

This code performs K-Means clustering to determine the optimal number of clusters for a dataset using the Silhouette Score as an evaluation metric. The dataset is first loaded, and five specific features (self_esteem, blood_pressure, sleep_quality, future_career_concerns, and bullying) are selected for clustering. These features are scaled using StandardScaler to standardize their values, ensuring fair contribution to the clustering process. The algorithm is applied for a range of cluster numbers (k) from 2 to 10, and the average Silhouette Score is computed for each k. The Silhouette Score measures how well each point fits within its cluster compared to other clusters, with higher values indicating better-defined clusters. The code identifies the optimal number of clusters by selecting the k with the highest Silhouette Score, which is found to be k=5 with a score of 0.6023. Additionally, the second-highest score is observed for k=4 with a score of 0.5905, providing comparative insights. Finally, a line plot visualizes the Silhouette Scores across all k values, reinforcing the conclusion that five clusters best represent the dataset's structure.

- Elbow method

code

```
# Install the kneed library for determining the optimal number of clusters using the elbow method
!pip install kneed

# Import necessary libraries for plotting and clustering
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from kneed import KneeLocator


# Assuming df_scaled is the scaled data
X = df_scaled

# Perform k-means clustering for different values of k
wss_values = []
k_values = range(1, 11)

# Fit K-means for each k value and calculate the WSS (Within-cluster Sum of Squares)
for k in k_values:
    kmeans = KMeans(n_clusters=k, random_state=42, n_init='auto')  # Initialize KMeans for each k
    kmeans.fit(X) # Fit the model to the data
    wss_values.append(kmeans.inertia_)  # Store the WSS for the current k

# Plot the elbow method: WSS vs number of clusters (k)
plt.plot(k_values, wss_values, marker='o')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Within-cluster Sum of Squares (WSS)')
plt.title('Elbow Method for Optimal k')

# Use KneeLocator to detect the 'elbow' or turning point in the curve
knee = KneeLocator(k_values, wss_values, curve='convex', direction='decreasing')
turning_point = knee.elbow  # The point where the WSS starts to decrease at a slower rate

# Highlight the turning point (optimal k) in the plot
plt.axvline(x=turning_point, linestyle='--', color='red', label=f'Chosen k = {turning_point}')
plt.legend()  # Add a legend to the plot
plt.show()
```



Description

. This code uses the Elbow Method to determine the optimal number of clusters (k) for a dataset. The method evaluates the Within-Cluster Sum of Squares (WSS), which measures how compact the clusters are. Lower WSS values indicate tighter, more cohesive clusters. The code first computes the WSS for a range of cluster numbers (k from 1 to 10) by fitting a K-Means model to the data for each k. The results .are visualized on a plot with WSS on the y-axis and the number of clusters on the x-axis
To identify the "elbow point" (the point where the WSS starts decreasing at a slower rate), the KneeLocator library is used. This elbow point indicates the optimal number of clusters, balancing cluster compactness and complexity. In this analysis, the elbow point was determined to be k=3, suggesting that three clusters best represent the data. Additionally, a second turning point related to the silhouette coefficient was identified, which could be used for further exploration to capture distinct cluster
The method provides a systematic approach to selecting the best number of clusters, ensuring .structures meaningful and interpretable clustering results.

- K-Means Algorithm
    - Clustering [K=3]:

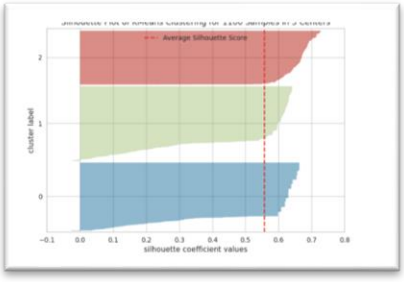
code

```
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(45)

# Perform K-means clustering for K=3
kmeans = KMeans(n_clusters=3, random_state=45, n_init='auto')  # Set n_init explicitly a
kmeans_result = kmeans.fit(df_scaled)

# Print the cluster centers (centroids) of the 3 clusters
print("Cluster Centers:")
print(kmeans_result.cluster_centers_)
# Print the labels for each data point, indicating which cluster each data point belongs
print("\nCluster Labels:")
print(kmeans_result.labels_)
```

```
Cluster Centers:
[[ 0.87160896  0.03195539  1.07891605 -0.971755   -0.9775455 ]
 [-0.96549447  0.98198051 -0.8756586   0.96937318  0.90670043]
 [ 0.24972719 -1.41044155 -0.13665306 -0.12904182 -0.03430045]]

Cluster Labels:
[2 1 2 ... 0 1 1]
```

```
# Install necessary libraries for clustering visualizations
!pip install kneed yellowbrick
# Import necessary libraries
from yellowbrick.cluster import SilhouetteVisualizer
from sklearn.cluster import KMeans

# Perform K-means clustering for K=3
kmeans = KMeans(n_clusters=3, n_init='auto', random_state=45)  # Random state for reprod
visualizer = SilhouetteVisualizer(kmeans, colors='yellowbrick') # Initialize the Silhoue
# Fit the visualizer to the scaled data (df_scaled) and display the silhouette plot
visualizer.fit(df_scaled)
visualizer.show()
```

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# Assuming df_scaled is your scaled data (from your previous steps)
X = df_scaled

# Perform k-means clustering with k=3 (based on your analysis)
kmeans = KMeans(n_clusters=3, random_state=45, n_init='auto')
kmeans.fit(X)
# Get the cluster labels for each data point after fitting the model
labels = kmeans.labels_

# Compute the Within-Cluster Sum of Squares (WSS), which measures how compact the cluste
# This is calculated as the sum of squared distances between data points and their respe
wss = kmeans.inertia_

# Compute the Average Silhouette Score, which assesses the quality of clustering
# Higher values (closer to 1) indicate well-separated clusters, while negative values su
silhouette_avg = silhouette_score(X, labels)

# Print the evaluation metrics
print("WSS:", wss)
print("Average Silhouette Score:", silhouette_avg)
```



Description

. This code applies the K-Means clustering algorithm with three clusters (k=3) to group data points and evaluates the quality of clustering using two metrics: Within-Cluster Sum of Squares (WSS) and Silhouette Score. The dataset (df_scaled) is clustered, and each cluster is characterized by its centroid, with all data points assigned a corresponding cluster label. The WSS, which measures cluster compactness by calculating the sum of squared distances between data points and their centroids, is 1563.33, indicating moderately cohesive clusters. The Silhouette Score, which assesses the separation and cohesion of clusters (ranging from -1 for poor clustering to 1 for well-separated clusters), is 0.5569, suggesting that clusters are reasonably separated with some overlap. A Silhouette Plot is generated to visually inspect the clustering quality, revealing the distribution of points across clusters. While the results indicate moderate cluster compactness and separation, further refinement, such as increasing the .number of clusters, may improve the clustering performance

- Clustering [K=4]:

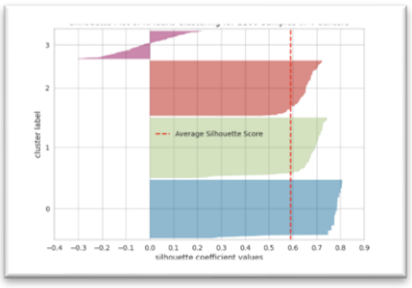
code

```
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(45)

# Perform K-means clustering for K=4
kmeans = KMeans(n_clusters=4, random_state=45, n_init='auto')  # Set n_init explicitly a
kmeans_result = kmeans.fit(df_scaled)

# Print the cluster centers (centroids) of the 4 clusters
print("Cluster Centers:")
print(kmeans_result.cluster_centers_)
# Print the labels for each data point, indicating which cluster each data point belongs
print("\nCluster Labels:")
print(kmeans_result.labels_)
```

```
Cluster Centers:
[[ 1.06884693 -0.15028213  1.16246324 -1.06848231 -1.05480507]
 [-1.07155845  0.98198051 -1.0432046   1.17510763  1.19072503]
 [ 0.2463754  -1.41841629 -0.13137956 -0.12369554 -0.03742675]
```

```
# Install the required packages for visualizing the silhouette scores and clustering
!pip install kneed yellowbrick
# Import necessary libraries
from yellowbrick.cluster import SilhouetteVisualizer
from sklearn.cluster import KMeans

# Perform K-means clustering for K=4
kmeans = KMeans(n_clusters=4, n_init='auto', random_state=45)  # Random state for reprod
# Initialize the SilhouetteVisualizer to visualize the silhouette score of the clusterin
# 'colors' argument defines the color scheme for the visualization
visualizer = SilhouetteVisualizer(kmeans, colors='yellowbrick')
# Fit the visualizer on the scaled data (df_scaled)
visualizer.fit(df_scaled)
# Display the silhouette visualization
visualizer.show()
```

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# Assuming df_scaled is your scaled data (from your previous steps)
X = df_scaled

# Perform k-means clustering with k=3 (based on your analysis)
kmeans = KMeans(n_clusters=3, random_state=45, n_init='auto')
kmeans.fit(X)
# Get the cluster labels for each data point after fitting the model
labels = kmeans.labels_

# Compute the Within-Cluster Sum of Squares (WSS), which measures how compact the cluste
# This is calculated as the sum of squared distances between data points and their respe
wss = kmeans.inertia_

# Compute the Average Silhouette Score, which assesses the quality of clustering
# Higher values (closer to 1) indicate well-separated clusters, while negative values su
silhouette_avg = silhouette_score(X, labels)

# Print the evaluation metrics
print("WSS:", wss)
print("Average Silhouette Score:", silhouette_avg)
```



Description

This code demonstrates the application of the K-Means clustering algorithm with four clusters (k=4) to identify groups within the dataset. The clustering model calculates Within-Cluster Sum of Squares (WSS) and Silhouette Score to evaluate the clustering performance. The WSS value, which measures cluster compactness by summing squared distances between points and their centroids, is 1184.57, indicating improved cohesion compared to the k=3 analysis. The Silhouette Score, used to evaluate the separation and cohesion of clusters, is 0.5911, higher than the previous score of 0.5569, signifying better-defined and more distinct clusters. The centroids of the four clusters are identified, and each data point is labeled with its corresponding cluster. A Silhouette Plot provides a visual representation of the clustering quality, supporting the conclusion that the clusters formed with k=4 are more homogeneous and well-separated than those formed with fewer clusters. This analysis highlights the improvement in ..clustering performance achieved by increasing the number of clusters

- Clustering [K=5]:

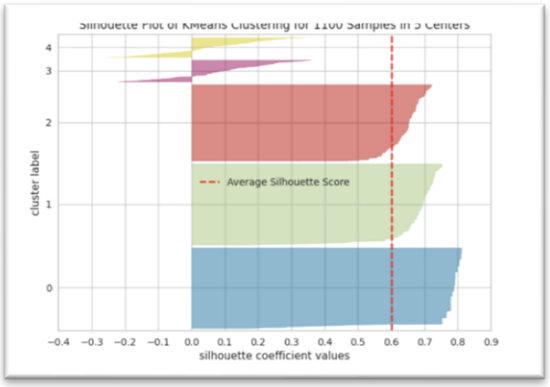
code

```
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(45)

# Perform K-means clustering for K=5
kmeans = KMeans(n_clusters=5, random_state=45, n_init='auto')  # Set n_init explicitly a
kmeans_result = kmeans.fit(df_scaled)

# Print the cluster centers (centroids) of the 5 clusters
print("Cluster Centers:")
print(kmeans_result.cluster_centers_)
# Print the labels of the clusters assigned to each data point
print("\nCluster Labels:")
print(kmeans_result.labels_)
```

```
Cluster Centers:
[[ 1.06943686 -0.15744835  1.16434078 -1.06841722 -1.05479207]
 [-1.0876475   0.98198051 -1.07459326  1.19443506  1.19357164]
 [ 0.2463754  -1.41841629 -0.13137956 -0.12369554 -0.03742675]
 [-0.50442531  0.98198051  1.03861477  0.33604139 -0.091833  ]
 [-0.26188321  0.98198051 -0.94831853 -0.46654201 -0.37824321]]

Cluster Labels:
[2 1 2 ... 0 1 1]
```

```
# Install the necessary packages (kneed and yellowbrick) if they are not already install
!pip install kneed yellowbrick
# Import the SilhouetteVisualizer from yellowbrick and KMeans from scikit-learn
from yellowbrick.cluster import SilhouetteVisualizer
from sklearn.cluster import KMeans

# Perform K-means clustering with K=5 clusters
kmeans = KMeans(n_clusters=5, n_init='auto', random_state=45)  # Random state for reprod
# Create a SilhouetteVisualizer to evaluate the quality of the clusters
visualizer = SilhouetteVisualizer(kmeans, colors='yellowbrick')
# Fit the visualizer to the scaled data (df_scaled) and display the silhouette plot
visualizer.fit(df_scaled)
visualizer.show()
```

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# Assuming df_scaled is your scaled data (from your previous steps)
X = df_scaled

# Perform k-means clustering with k=3 (based on your analysis)
kmeans = KMeans(n_clusters=3, random_state=45, n_init='auto')
kmeans.fit(X)
# Get the cluster labels for each data point after fitting the model
labels = kmeans.labels_

# Compute the Within-Cluster Sum of Squares (WSS), which measures how compact the cluste
# This is calculated as the sum of squared distances between data points and their respe
wss = kmeans.inertia_

# Compute the Average Silhouette Score, which assesses the quality of clustering
# Higher values (closer to 1) indicate well-separated clusters, while negative values su
silhouette_avg = silhouette_score(X, labels)

# Print the evaluation metrics
print("WSS:", wss)
print("Average Silhouette Score:", silhouette_avg)
```



Description

This code demonstrates the application of the K-Means clustering algorithm with four clusters (k=4) to identify groups within the dataset. The clustering model calculates Within-Cluster Sum of Squares (WSS) and Silhouette Score to evaluate the clustering performance. The WSS value, which measures cluster compactness by summing squared distances between points and their centroids, is 1184.57, indicating improved cohesion compared to the k=3 analysis. The Silhouette Score, used to evaluate the separation and cohesion of clusters, is 0.5911, higher than the previous score of 0.5569, signifying better-defined and more distinct clusters. The centroids of the four clusters are identified, and each data point is labeled with its corresponding cluster. A Silhouette Plot provides a visual representation of the clustering quality, supporting the conclusion that the clusters formed with k=4 are more homogeneous and well-separated than those formed with fewer clusters. This analysis highlights the ..improvement in clustering performance achieved by increasing the number of clusters

Mining task	Comparison Criteria
Clustering	

# of clusters	K=3	K=4	K=5
Average Silhouette width	0.5569280455512305	0.591095806851136	0.6022920828329283
total within-cluster sum of square	1563.3293320237235	1184.5678169269129	1002.1736709966773

Findings

We began by selecting a dataset that represents student information to estimate the likelihood and levels of stress they may experience. This enables us to predict the factors contributing to elevated stress levels and identify both the dependent and influencing factors.

To achieve accurate and efficient results, we applied several preprocessing techniques to enhance the quality and performance of the data. We utilized various plotting methods, such as boxplots and histograms, to visualize the data, which helped us better understand it and apply appropriate preprocessing techniques.

Based on the plots and other commands, we found that our dataset had no missing values or outliers.

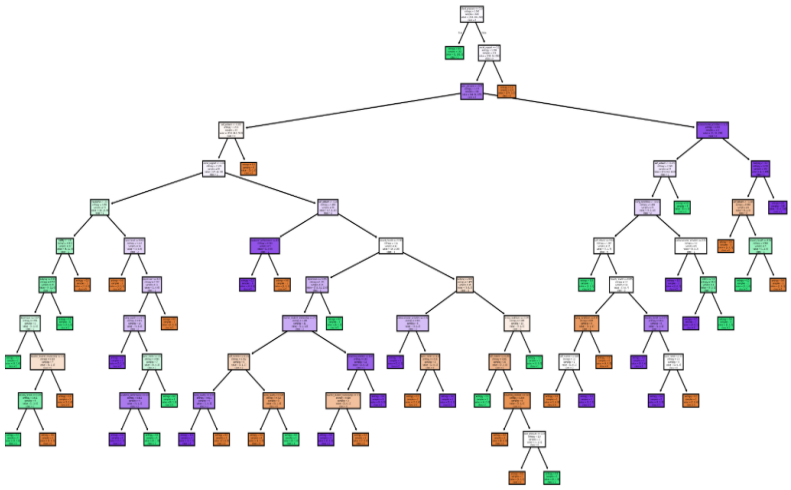
Additionally, we performed data transformation by normalizing and discretizing some attributes to ensure equal weight for each feature and facilitate handling the data during data mining tasks.

As a result, we proceeded with the data mining tasks of classification and clustering. For classification, we used the decision tree method to construct our model. We experimented with three different training and

testing data splits to identify the best configuration for construction and evaluation. The following results were obtained:

- 60% train, 40% test, Accuracy: 0.8840
- 70% train, 30% test, Accuracy: 0.8727
- 80% train, 20% test, Accuracy: 0.8545

With 60% training data and 40% test data, the first model achieved the highest accuracy, meaning that the majority of the tuples were correctly classified.



From the decision tree, we can derive the following insights:

- Root of the Tree: The root node of this decision tree is "blood_pressure." It splits the data into two subsets based on whether the blood pressure is higher or lower than a specified threshold, indicating that this attribute has significant discriminatory power.

- First Level Split: After the root, the tree further splits based on the "sleep_quality" attribute. This suggests that, following blood pressure, sleep quality plays a crucial role in influencing the outcome.

- Nodes and Branches: Each node in the tree represents a decision point where the dataset is divided based on the value of a specific attribute. The branches extending from each node split the dataset into subsets according to the conditions set at each node (e.g., whether an attribute's value is less than or greater than a specific threshold).

- Leaf Nodes: The leaf nodes, located at the end of each branch, represent the final classification outcome. The path from the root through the intermediate nodes leads to these leaf nodes, which in this dataset correspond to the "Teacher-Student Relationship" and "Peer Pressure."

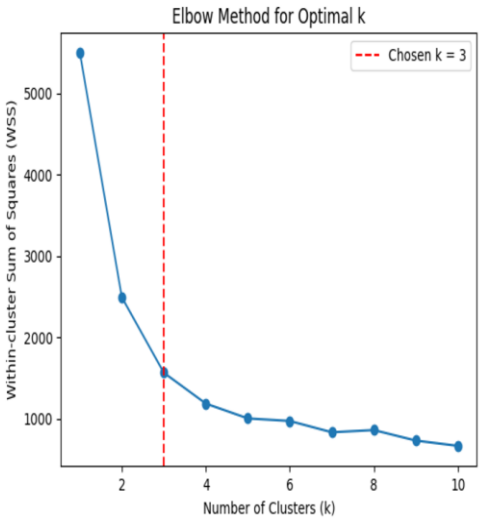
- Class Label and Gini Value: Each leaf node is associated with a class label, which is determined by the Gini index, a measure of impurity or homogeneity used to evaluate the effectiveness of the splits.

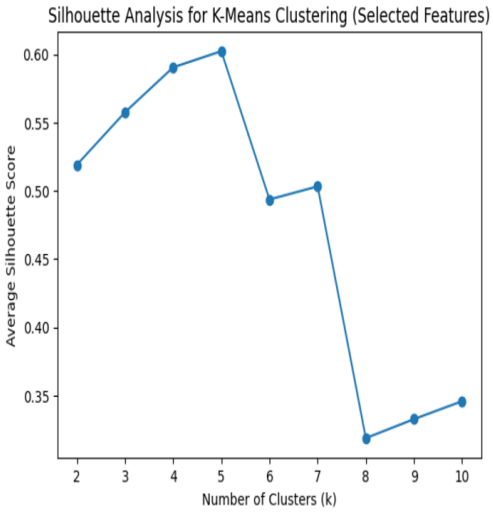
For our clustering analysis, we employed the K-means algorithm with varying numbers of clusters (K) to identify the optimal configuration. The evaluation yielded the following results:

- **K = 3**
  - Average Silhouette Score: 0.5569
  - WSS: 1563.33
- **K = 4**
  - Average Silhouette Score: 0.5911
  - WSS: 1184.57
- **K = 5**
  - Average Silhouette Score: 0.6023
  - WSS: 1002.17

An analysis of these results revealed that the model with K=5 clusters achieved the highest average silhouette width, indicating strong cohesion within clusters and clear separation between them. Additionally, the

visualizations of the clusters in Figures 3, 4, and 5 reinforced this observation, showing distinct groupings with minimal overlap, especially for K=5. Therefore, we can conclude that the K=5 clustering model is the most suitable choice for this dataset.





References

[1] R. Xnach, "Student stress factors: A comprehensive analysis," *Kaggle*. [Online]. Available: https://www.kaggle.com/datasets/rxnach/student-stress-factors-a-comprehensive-analysis. .

[2] *Labs and Lecture Slides*, College of Computer Science, Department of Information Technology, King Saud University.