Autism Classification Using Extreme Learning Network

Introduction to Autism and Extreme Machine Learning

What is Autism Spectrum Disorder (ASD)?

Autism Spectrum Disorder (ASD) is a developmental disorder that affects communication, behavior, and social interaction. It is called a "spectrum" disorder because there is a wide variation in the type and severity of symptoms people experience.

Signs and Symptoms of Autism

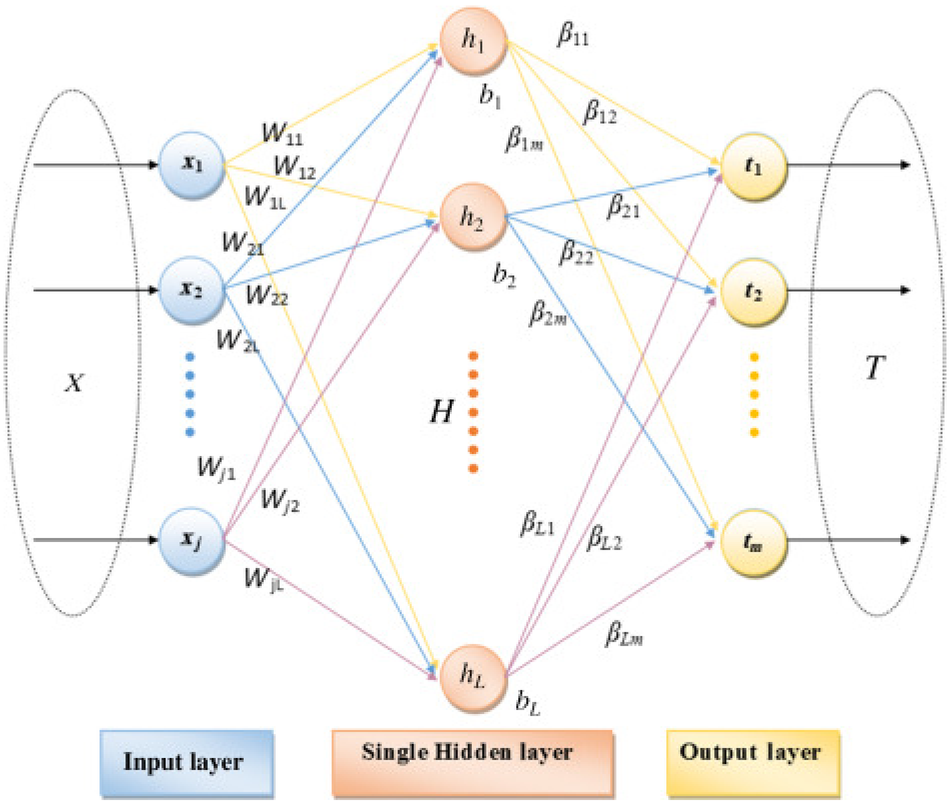
- **Social Communication Challenges**: Difficulty in understanding social cues, maintaining conversations, and forming relationships.
- **Repetitive Behaviors**: Repetitive movements, speech, or use of objects.

- **Restricted Interests**: Intense interest in specific topics or activities.
- **Sensory Sensitivities**: Unusual reactions to sensory input such as sounds, lights, textures, or smells.
- **Developmental Delays**: Delays in reaching milestones in speech, motor skills, or cognitive abilities.

Early diagnosis and intervention can significantly improve the outcomes for individuals with ASD. Machine learning (ML) has become a powerful tool in developing predictive models to aid in early detection and intervention.

Extreme Learning Networks and the Pseudo-Moore Inverse

Extreme Learning Networks (ELNs)



Extreme Learning Networks are a type of single-hidden layer feedforward neural network (SLFN) where the input weights and biases are randomly assigned and remain fixed. Only the output weights are learned. This unique approach allows for extremely fast training compared to traditional neural networks, where all weights are typically learned through iterative processes like backpropagation.

Moore-Penrose Pseudoinverse

The Moore-Penrose pseudoinverse is a generalization of the matrix inverse for non-square or singular matrices. It is used to find the least-squares solution to a system of linear equations.

For a matrix A, the Moore-Penrose pseudoinverse, denoted as A^+, satisfies the following properties:

1. $AA^+A = A$

2. $A^+AA^+ = A^+$
3. $(AA^+)^T = AA^+$
4. $(A^+A)^T = A^+A$

Application in Extreme Learning Networks

In the context of ELNs, the Moore-Penrose pseudoinverse is used to compute the output weights.

ELN Formulation

1. **Input to Hidden Layer Transformation:** Given input data $\mathbf{X} \in \mathbb{R}^{N \times d}$, where N is the number of samples and d is the number of features, we randomly initialize the input weights $\mathbf{W} \in \mathbb{R}^{d \times L}$ and biases $\mathbf{b} \in \mathbb{R}^{L}$, where L is the number of hidden neurons.

   The hidden layer output $\mathbf{H}$ can be computed as: \[$\mathbf{H} = g(\mathbf{XW} + \mathbf{b})$\] Here, g is the activation function applied element-wise.

2. **Hidden to Output Layer Transformation:** For the output layer, the network needs to learn the weights β such that: \[$\mathbf{H}\beta = \mathbf{Y}$\] where $\mathbf{Y} \in \mathbb{R}^{N \times m}$ is the target output, and m is the number of output neurons.

Computing the Output Weights

To find the optimal β, we use the Moore-Penrose pseudoinverse of $\mathbf{H}$:

\[$\beta = \mathbf{H}^+\mathbf{Y}$\]

Here's how the pseudoinverse $\mathbf{H}^+$ is computed:

1. **SVD Decomposition:** Perform Singular Value Decomposition (SVD) on $\mathbf{H}$: \[$\mathbf{H} = \mathbf{U\Sigma V}^T$\] where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices, and $\mathbf{\Sigma}$ is a diagonal matrix with singular values.

2. **Inverse of Singular Values:** Compute the pseudoinverse of $\mathbf{\Sigma}$, denoted as $\mathbf{\Sigma}^+$, by taking the reciprocal of each non-zero singular value and transposing the resulting matrix.

3. **Construct the Pseudoinverse:** The Moore-Penrose pseudoinverse of $\mathbf{H}$ is then: \[$\mathbf{H}^+ = \mathbf{V\Sigma}^+\mathbf{U}^T$\]

4. **Compute Output Weights:** Finally, compute the output weights β: \[$\beta = \mathbf{H}^+\mathbf{Y}$\]

Components of ELN

1. **Input Layer**:
   - The input layer consists of neurons that receive the input features from the dataset.
2. **Hidden Layer**:
   - The hidden layer contains a set of neurons with randomly assigned weights and biases. These parameters are fixed and not updated during training. The activation function is typically a non-linear function such as the sigmoid or ReLU.

- The number of neurons in the hidden layer can be chosen based on the complexity of the problem.

3. **Output Layer**:
   - The output layer generates the final predictions. The weights connecting the hidden layer to the output layer are the only parameters that are learned during training.

Training the ELN

Training an ELN involves the following steps:

1. **Initialization**:
   - Randomly initialize the weights and biases for the hidden layer.
2. **Feature Transformation**:
   - Apply the hidden layer transformation to the input data, resulting in a new feature representation.
3. **Output Weight Calculation**:
   - Calculate the output weights using a least squares solution to minimize the error between the predicted and actual outputs.

ELN Workflow

Here is a complete workflow for training an ELN on a dataset:

```python
# Define the number of neurons in the hidden layer
n_hidden_neurons = 100

# Randomly initialize the weights and biases for the hidden layer
input_size = X_train.shape[1]
W = np.random.randn(input_size, n_hidden_neurons)
b = np.random.randn(n_hidden_neurons)

# Activation function (e.g., sigmoid)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Transform the input data using the hidden layer
H_train = sigmoid(np.dot(X_train, W) + b)
H_test = sigmoid(np.dot(X_test, W) + b)

# Calculate the output weights (beta)
beta = np.dot(pinv(H_train), y_train)

# Make predictions
y_train_pred = np.dot(H_train, beta)
y_test_pred = np.dot(H_test, beta)

# Convert predictions to binary labels
y_train_pred = np.where(y_train_pred > 0.5, 1, 0)
```

```
y_test_pred = np.where(y_test_pred > 0.5, 1, 0)

# Evaluate the ELN
accuracy = accuracy_score(y_test, y_test_pred)
precision = precision_score(y_test, y_test_pred)
recall = recall_score(y_test, y_test_pred)
f1 = f1_score(y_test, y_test_pred)

print(f'Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')
```

Advantages of ELN

- **Fast Training**: Due to the random initialization of the hidden layer weights and biases, and the least squares solution for output weights, ELNs train extremely quickly compared to traditional neural networks.
- **Good Generalization**: Despite the simplicity, ELNs often achieve good generalization performance on a variety of tasks.
- **Ease of Implementation**: The straightforward structure and training process make ELNs easy to implement.

Challenges

- **Sensitivity to Initialization**: The random initialization of hidden layer parameters can affect performance, and finding the right number of hidden neurons can require experimentation.
- **Fixed Hidden Layer**: The fixed nature of the hidden layer parameters means that they may not always capture the most relevant features, especially for highly complex tasks.

The ELN is a powerful tool for tasks requiring fast training and good generalization, and ongoing research continues to explore its potential applications and improve its performance.

CODING WORKSPACE

In [1]:
```
# import some dependencies
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder,StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from numpy.linalg import pinv
from sklearn.metrics import classification_report,accuracy_score
```

In [2]:
```
# read in the data
df = pd.read_csv("C:\\Datasets\\Autism\\autism_screening.csv")
```

```
df.head()
```

Out[2]:

	A1_Score	A2_Score	A3_Score	A4_Score	A5_Score	A6_Score	A7_Score	A8_Score	A9_Score
0	1	1	1	1	0	0	1	1	0
1	1	1	0	1	0	0	0	1	0
2	1	1	0	1	1	0	1	1	1
3	1	1	0	1	0	0	1	1	0
4	1	0	0	0	0	0	0	1	0

5 rows × 21 columns

In [3]:
```
df.shape
```

Out[3]:
```
(704, 21)
```

In [4]:
```
df.dtypes
```

Out[4]:
```
A1_Score              int64
A2_Score              int64
A3_Score              int64
A4_Score              int64
A5_Score              int64
A6_Score              int64
A7_Score              int64
A8_Score              int64
A9_Score              int64
A10_Score             int64
age                 float64
gender               object
ethnicity            object
jundice              object
austim               object
contry_of_res        object
used_app_before      object
result              float64
age_desc             object
relation             object
Class/ASD            object
dtype: object
```

In [5]:
```
df['ethnicity'].value_counts().head() # I did this to first understand how many row
```

Out[5]:
```
White-European     233
Asian              123
?                   95
Middle Eastern      92
Black               43
Name: ethnicity, dtype: int64
```

In [6]:
```
df['ethnicity'] = df['ethnicity'].replace({"?":"NotListed"})
```

In [7]:
```
df['ethnicity'].value_counts().head() # I replaced  the question mark
```

Out[7]:
```
White-European     233
Asian              123
NotListed           95
Middle Eastern      92
Black               43
Name: ethnicity, dtype: int64
```

In [8]:
```
df['relation'] = df['relation'].replace({"?":"NotListed"}) # same procedure to elin
```

In [9]:
```
# age had two null values,I replaced that using the mean value
df['age'] = df['age'].fillna(df['age'].mean())
```

In [10]:
```
# Create a function that returns the frequent items in our columns
def frequent_items(column,df=df):
    """This function return the top 5 most appearing values in a column ,It also ir
    appearing values"""
    return print(f"The most appearing in {column} in order of appearance /n {df[col
```

In [11]:
```
df.columns
```

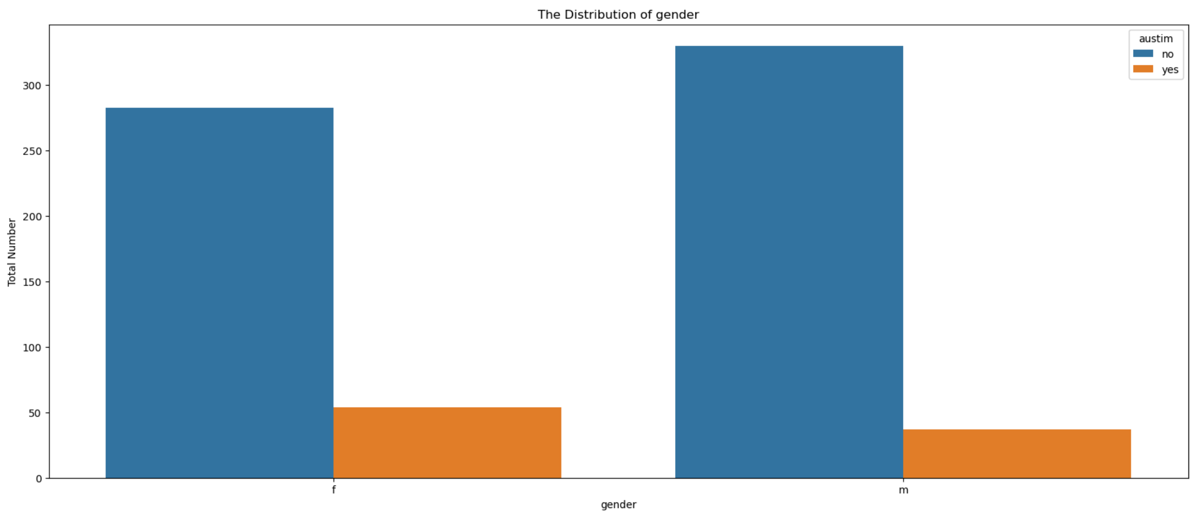
Out[11]:
```
Index(['A1_Score', 'A2_Score', 'A3_Score', 'A4_Score', 'A5_Score', 'A6_Score',
       'A7_Score', 'A8_Score', 'A9_Score', 'A10_Score', 'age', 'gender',
       'ethnicity', 'jundice', 'austim', 'contry_of_res', 'used_app_before',
       'result', 'age_desc', 'relation', 'Class/ASD'],
      dtype='object')
```

DATA EXPLORATION AND VISUALIZATION

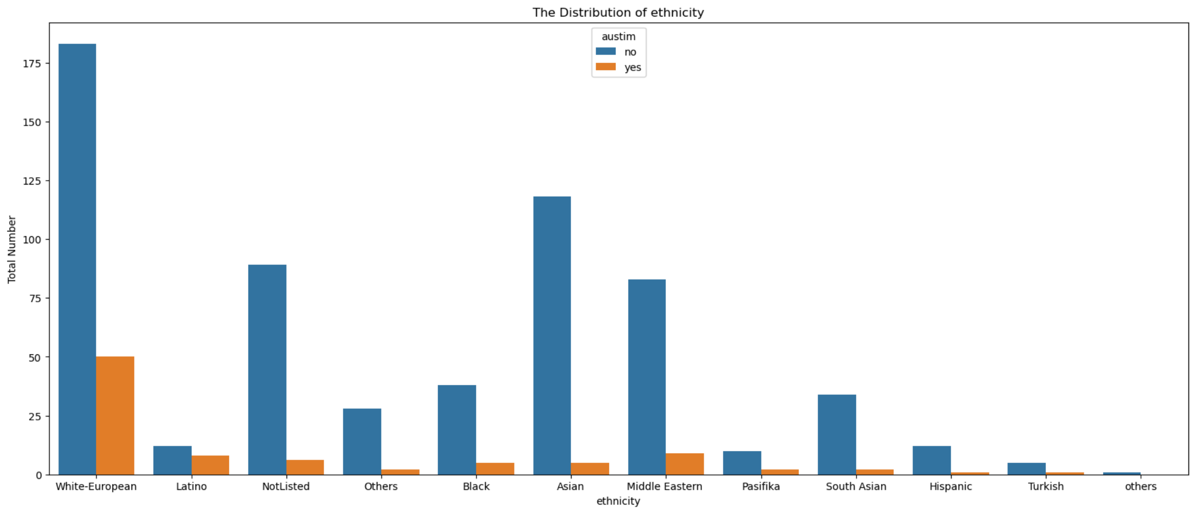
In [12]:
```
# To visualize some of the categorical data ,I will use the countplot from the seab
#that returns the visualization.Also a histogram plot will be handy for univariate

def draw_countplot(column,hue=None,df=df):
    fig=plt.figure(figsize=(20,8))
    sns.countplot(x=column,data=df,hue=hue)
    plt.xlabel(column)
    plt.ylabel("Total Number")
    plt.title(f"The Distribution of {column}")
    plt.show()
def draw_histogram(column,hue=None,df=df):
    fig=plt.figure(figsize=(20,8))
    sns.histplot(x=column,data=df,hue=hue)
    plt.xlabel(column)
    plt.ylabel("Total Number")
    plt.title(f"The Distribution of {column}")
    plt.show()
```

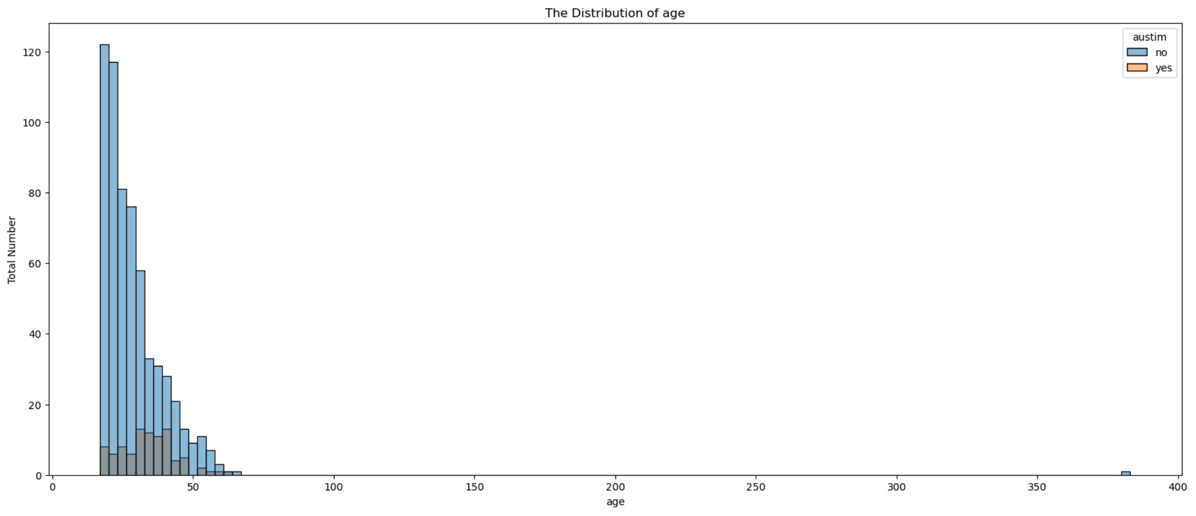
In [13]:
```
# 1. Which gender dominates the Autism disorder
draw_countplot("gender",hue='austim') # A lot of females have Autistic syndrome
```



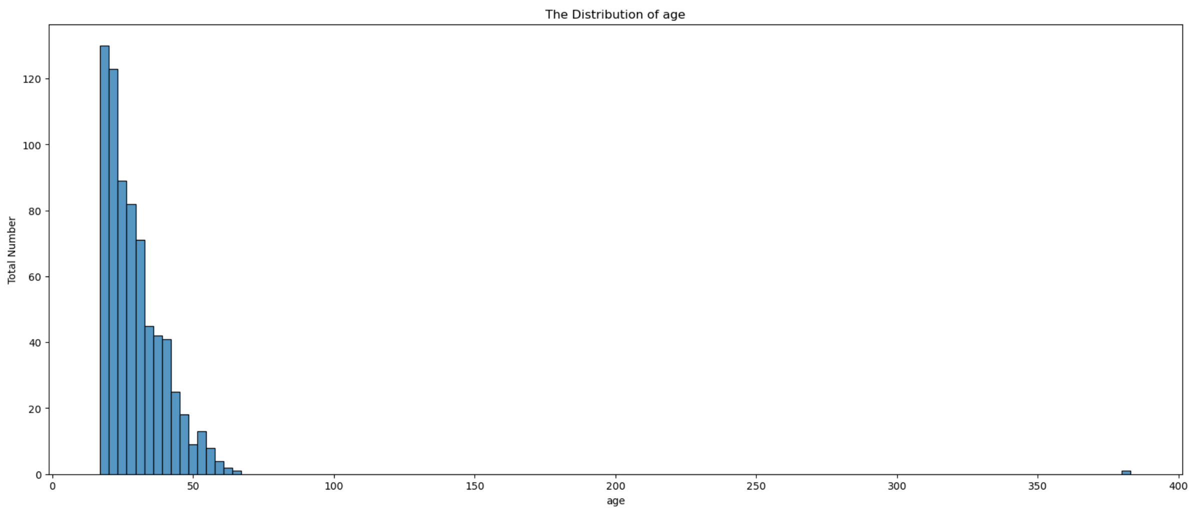
```
In [14]: # 2 . Which ethnicity has more Autism cases ?
         draw_countplot("ethnicity",hue="austim") # White Europeans dominate with Autism
```



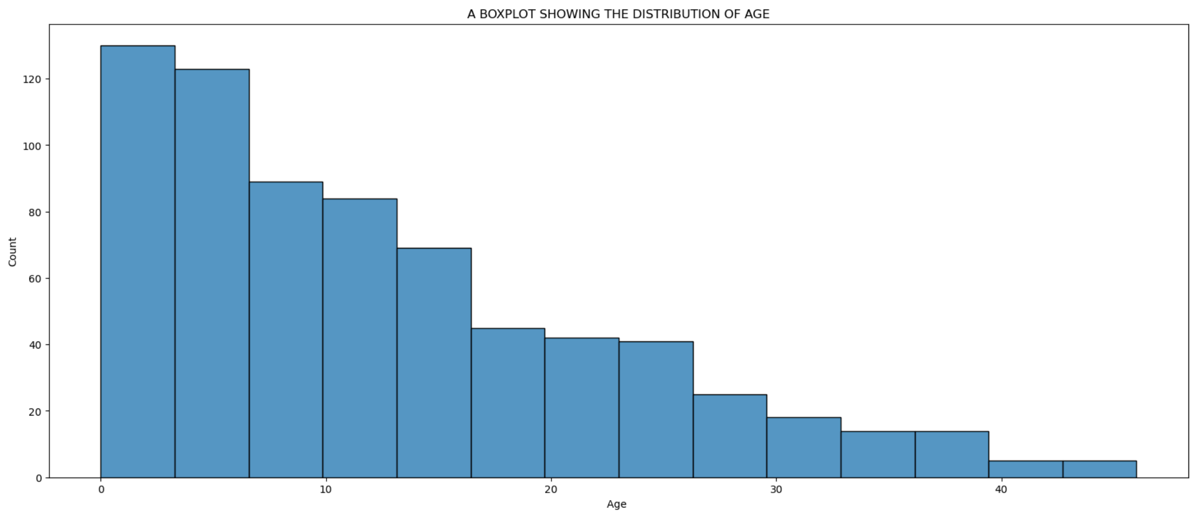
```
In [15]: # 3 Which age group has the most cases of Autism?
         draw_histogram("age",hue='austim') # A majority of the confirmed cases are aged 30
```



```
In [16]: # 4.Which age dominates the survey?
         draw_histogram("age")
```



In [43]:
```python
plt.figure(figsize=(20,8))
sns.histplot(x="age",data=df)
plt.xlabel("Age ")
plt.title("A BOXPLOT SHOWING THE DISTRIBUTION OF AGE")
plt.show()
```



In [18]:
```python
# TO ANALYZE AND VISUALIZE Categorical data I will be forced to convert them into c
lbl_encoder = {}
categorical_columns = ['age', 'gender','ethnicity', 'jundice', 'austim', 'contry_of
                        ,'result', 'age_desc', 'relation', 'Class/ASD','used_app_bef
for columns in categorical_columns:
    lbl_encoder = LabelEncoder()
    df[columns] = lbl_encoder.fit_transform(df[columns])
```

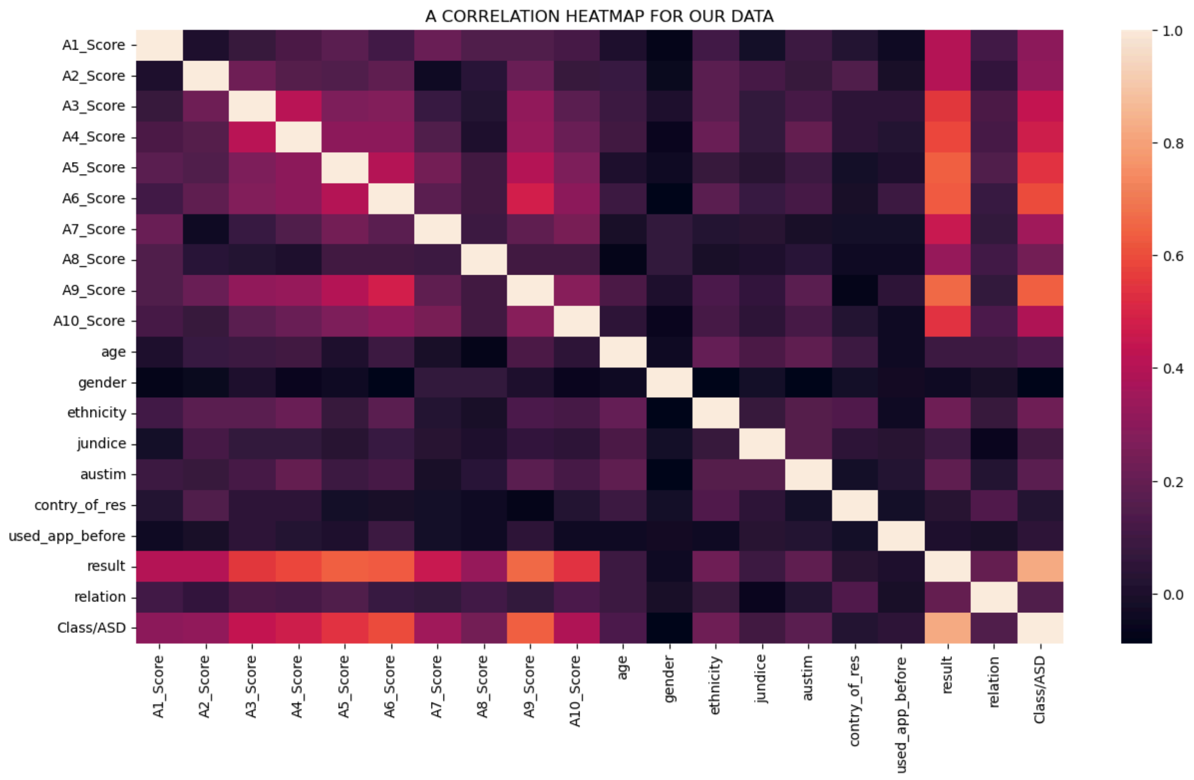
In [19]:
```python
df.drop(['age_desc'],axis=1,inplace=True)
```

In [20]:
```python
df.head() # Now we have no categorical columns
```

Out[20]:

	A1_Score	A2_Score	A3_Score	A4_Score	A5_Score	A6_Score	A7_Score	A8_Score	A9_Score
0	1	1	1	1	0	0	1	1	0
1	1	1	0	1	0	0	0	1	0
2	1	1	0	1	1	0	1	1	1
3	1	1	0	1	0	0	1	1	0
4	1	0	0	0	0	0	0	1	0

In [21]:
```
# Visualize it in a heatmap
plt.figure(figsize=(15,8))
sns.heatmap(df.corr())
plt.title("A CORRELATION HEATMAP FOR OUR DATA")
plt.show()
```



EXTREME LEARNING NETWORK

In [22]:
```
#Implementation of the Extreme Learning Network (ELN) as a Python class.
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

class ExtremeLearningNetwork(BaseEstimator, ClassifierMixin):
    def __init__(self, n_hidden_neurons=100, activation_function='sigmoid'):
        self.n_hidden_neurons = n_hidden_neurons
        self.activation_function = activation_function
        self.W = None
        self.b = None
        self.beta = None
        self.scaler = StandardScaler()

    def _sigmoid(self, x):
        return 1 / (1 + np.exp(-x))
    def _activation(self, X):
        if self.activation_function == 'sigmoid':
            return self._sigmoid(X)
        else:
            raise ValueError("Unsupported activation function")

    def fit(self, X, y):
        # Standardize the data
        X = self.scaler.fit_transform(X)

        # Initialize weights and biases for the hidden layer
        input_size = X.shape[1]
        self.W = np.random.randn(input_size, self.n_hidden_neurons)
        self.b = np.random.randn(self.n_hidden_neurons)

        # Compute hidden layer output
        H = self._activation(np.dot(X, self.W) + self.b)

        # Compute output weights (beta) using pseudoinverse
        self.beta = np.dot(np.linalg.pinv(H), y)

    def predict(self, X):
        # Standardize the data
        X = self.scaler.transform(X)

        # Compute hidden layer output
        H = self._activation(np.dot(X, self.W) + self.b)

        # Compute output
        y_pred = np.dot(H, self.beta)

        # Convert predictions to binary labels
        return np.where(y_pred > 0.5, 1, 0)

    def score(self, X, y):
        y_pred = self.predict(X)
        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred)
        recall = recall_score(y_test, y_pred)
        f1 = f1_score(y_test, y_pred)
        return {
            'accuracy': accuracy,
            'precision': precision,
            'recall': recall,
            'f1_score': f1
        }
```

Explanation of the ELN Class

- `__init__` **Method**: Initializes the ELN with the number of hidden neurons and the activation function. Also initializes the weights and biases to `None`.
- `_sigmoid` **Method**: Defines the sigmoid activation function.
- `_activation` **Method**: Applies the chosen activation function to the input.
- `fit` **Method**: Trains the ELN by standardizing the data, initializing the hidden layer parameters, computing the hidden layer output, and calculating the output weights using the pseudoinverse.
- `predict` **Method**: Predicts the labels for the input data by computing the hidden layer output and applying the learned output weights.

- `score` **Method**: Evaluates the model by calculating the accuracy, precision, recall, and F1 score of the predictions.

.

USING THE ELN CLASS

In [42]:
```
# Splitting the dataset into training and testing
# SELECTING  the  target features
X = df.drop(['contry_of_res','used_app_before','austim'],axis=1)
y = df['austim']
X = StandardScaler().fit_transform(X)
X_train,X_test,y_train,y_test = train_test_split(X,y,random_state=42,test_size=0.2

# Create and train the ELN model
eln = ExtremeLearningNetwork(n_hidden_neurons=100, activation_function='sigmoid')
eln.fit(X_train, y_train)

# Predict and evaluate the model
results = eln.score(X_test, y_test)
print(f'Accuracy: {results["accuracy"]}')
print(f'Precision: {results["precision"]}')
print(f'Recall: {results["recall"]}')
print(f'F1 Score: {results["f1_score"]}')
```

```
Accuracy: 0.9078014184397163
Precision: 0.5
Recall: 0.07692307692307693
F1 Score: 0.13333333333333336
```

COMPARING ELN TO CLASSICAL ML

In [34]:
```
rfc = RandomForestClassifier()
rfc.fit(X_train,y_train)
rf_pred = rfc.predict(X_test)
print(f"The classification report for Random forest classifier is \n{classificatior
```

```
The classification report for Random forest classifier is 
              precision    recall  f1-score   support

           0       0.91      0.97      0.94       128
           1       0.00      0.00      0.00        13

    accuracy                           0.88       141
   macro avg       0.45      0.48      0.47       141
weighted avg       0.82      0.88      0.85       141
```

Conclusion

- Extreme Learning Networks (ELNs) are advantageous due to their rapid training times compared to traditional Machine Learning algorithms. In our study, ELNs achieved an accuracy score of approximately 90.8%, outperforming the Random Forest classifier which had an accuracy of 88%. This demonstrates that ELNs can be more accurate in some cases. However, it is important to note the precision and recall values, which indicate that while ELNs are fast and generally accurate, they may struggle with class imbalance or detecting minority classes. Specifically, the ELN had a precision of 0.5 and a recall of 0.077, indicating a need for further refinement in identifying the minority class.

- The Random Forest classifier had a precision of 0.91 and a recall of 0.97 for the majority class, but it failed to identify the minority class, resulting in an F1 score of 0. Despite the slightly lower overall accuracy, the Random Forest classifier showed a strong performance for the majority class, highlighting the trade-offs between different algorithms and the importance of selecting the appropriate model based on the specific requirements of the task.