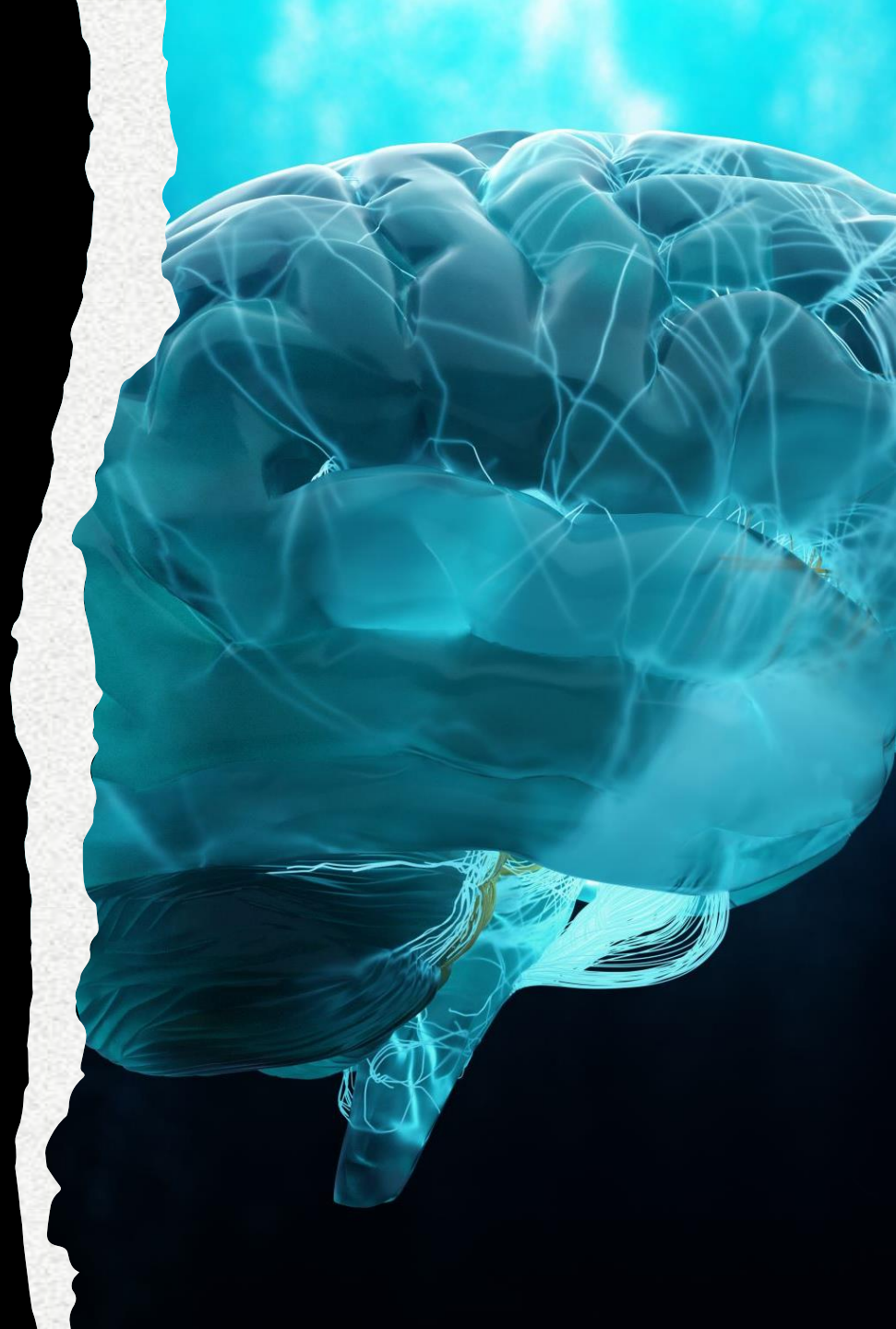
ALZHEIMERS DISEASE PREDICTION REPORT

Kusuma Korada

Github link: https://github.com/kismiskorada

Overview:

Alzheimer's Disease (AD) is a progressive neurological disorder that primarily affects the elderly, leading to memory loss, cognitive decline, and ultimately, loss of ability to carry out the simplest tasks. In the quest to understand, diagnose early, and treat Alzheimer's Disease, data plays a pivotal role. Through detailed analysis, we can uncover patterns, risk factors, and potential therapeutic targets that remain obscured without the power of data analysis.



OASIS LONGITUDINAL DATA SET

Longitudinal Dataset (oasis_longitudinal.csv)KAGGLE LINK: https://www.kaggle.com/jboysen/mri-and-alzheimers?select=oasis_cross-sectional.csv

CONTENTS OF THE DATASET:

Columns:

Subject ID: Identifier for the subject

MRI ID: Identifier for the MRI scan

Group: Classification of the subject (e.g., Nondemented, Demented)

Visit: The number of the visit

MR Delay: Delay in days from the first visit

M/F: Gender (Male/Female)

Hand: Handedness (Right/Left)

Age: Age of the subject

EDUC: Education level

SES: Socioeconomic status

MMSE: Mini-Mental State Examination score

CDR: Clinical Dementia Rating

eTIV: Estimated Total Intracranial Volume

nWBV: Normalize Whole Brain Volume

ASF: Atlas Scaling Factor

LOADING OF DATA SET

```python
import pandas as pd

# Specify the path to your CSV file
file_path = 'oasis_longitudinal.csv'

# Load the CSV file into a pandas DataFrame
data = pd.read_csv(file_path)

# Display the first few rows of the DataFrame to understand its content
print(data.head())

# Display the DataFrame's information to understand the structure, columns, and data types
data.info()
```

The output of the **info()** method provides the following details:

- The DataFrame consists of 373 entries (rows).
- There are 15 columns in total.
- Each column is listed along with its non-null count and data type.
- The data types include **object** (for string/text data), **int64** (for integer data), and **float64** (for floating-point/decimal data).
- Some columns have missing values (**NaN**), as indicated by the difference between the total number of entries and the non-null count for those columns.
- the 'SES' column has 354 non-null entries out of 373 total entries, indicating that there are 19 missing values in that column.
- the 'MMSE' column has 371 non-null entries, suggesting that there are 2 missing values in that column.

DATA PRE-PROCESSING

DEALING WITH THE CATEGORICAL DATA

This step aims to identify and display columns in the DataFrame that have 'object' or 'category' data types.

```
# Display the data types of each column to identify categorical data
print(data.dtypes)

# Specifically identify columns with 'object' or 'category' data types
categorical_columns = data.select_dtypes(include=['object', 'category']).columns.tolist()

# Print out the categorical columns
print("Categorical columns:", categorical_columns)
```

RESULTS :

1.The DataFrame contains several categorical columns LIKE 'Subject ID', 'MRI ID', 'Group', 'M/F', 'Hand' alongside numerical columns.

2.Categorical columns typically require special handling in data analysis and preprocessing tasks, such as encoding for machine learning algorithms or grouping for statistical analysis.

```
# Load the CSV file into a pandas DataFrame
data = pd.read_csv('oasis_longitudinal.csv')

# Check if 'Hand' column has 'L'
left_handed = 'L' in data['Hand'].values

left_handed
```

```
False
```

Since all are right handed persons in the oasis longitudinal data set typically don't provide any useful information for predicting outcomes or explaining variability in the data.So,I want to drop the Hand column.

The code initially loads a CSV file into a pandas DataFrame named data and then checks if the 'Hand' column contains any 'L' values, indicating left-handed individuals. Since there are no left-handed individuals ('L' not found), the 'Hand' column is deemed uninformative and is dropped from the DataFrame. This step ensures that irrelevant data is removed, enhancing the clarity and relevance of the dataset for analysis.

```
data.drop(['Subject ID', 'MRI ID'], axis=1, inplace=True)
```

These identifiers do not contribute to analysis or modeling. Hence this column is dropped

Here 'Group' is the target variable for the prediction of Alzhiemers Disease which can identify people with DEMENTIA OR NOT.

The 'Group' column contains categories such as 'Nondemented', 'Demented', and 'Converted'

MAPPING OF TARGET COLUMN:

```
# Define a manual mapping based on requirement
custom_mapping = {'Nondemented': 0, 'Demented': 1, 'Converted': 2}

# Apply the mapping to the 'Group' column
data['Group_encoded'] = data['Group'].map(custom_mapping)

# Display the first few rows to verify the encoding
print(data[['Group', 'Group_encoded']].head())
```

By mapping these categories to numerical values (0 for 'Nondemented', 1 for 'Demented', and 2 for 'Converted'), we transform the categorical data into a format that machine learning algorithms can understand. This encoding allows algorithms to effectively utilize the 'Group' column as a feature for Alzheimer's disease prediction tasks.

```python
# Drop the 'Group' column
data.drop('Group', axis=1, inplace=True)
```

The Original ‘not encoded’ group column is dropped

```python
#Label Encoding for "M/F"Column
from sklearn.preprocessing import LabelEncoder
# Initialize the label encoder
label_encoder = LabelEncoder()

# Apply label encoder to 'M/F' column, MALE = 1,FEMALE =0
data['M/F'] = label_encoder.fit_transform(data['M/F'])

# Optionally, display the first few rows to verify the encoding
print(data[['M/F']].head())
```

	M/F
0	1
1	1
2	1
3	1
4	1

Label encoding is a straightforward technique to convert categorical variables into numerical format, preserving the ordinal relationship if it exists

- The Categorical columns: ['Subject ID', 'MRI ID', 'Group', 'M/F', 'Hand’].
- The 'Subject ID', and 'MRI ID' are dropped as these columns don't provide any information for Disease Prediction. Since all persons are left-handed people, I dropped the 'Hand' Column.
- 'Group' the target column is handled by manual mapping Non-demented as 0, Demented as 1, Converted people as 2. Label Encoding of M/F Column is done.
- All Categorical Variables are handled now.

Handling of ‘missing values’

```
# Handling missing values: Impute missing values in 'SES' and 'MMSE' with their median values
data['SES'].fillna(data['SES'].median(), inplace=True)
data['MMSE'].fillna(data['MMSE'].median(), inplace=True)
```

The fillna() method is used to fill in missing values in the 'SES' and 'MMSE' columns. The median() function calculates the median value of each column, which serves as a robust measure of central tendency, less sensitive to outliers compared to mean. Missing values are then replaced with the calculated median value.

Imputation with median:

For the 'SES' column, missing values are replaced with the median of the 'SES' column. Similarly, for the 'MMSE' column, missing values are replaced with the median of the 'MMSE' column.

In-place operation:

The inplace=True parameter ensures that the changes made to the DataFrame are applied directly to the original DataFrame data, without the need for reassignment.

EXPLORATORY DATA ANALYSIS

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

# Select numerical columns
numerical_columns = data.select_dtypes(include=['int64', 'float64']).columns

# Plot box plots for each numerical column
for col in numerical_columns:
    plt.figure(figsize=(8, 8))
    sns.boxplot(x=data[col])
    plt.title(f'Box plot of {col}')
    plt.show()
```

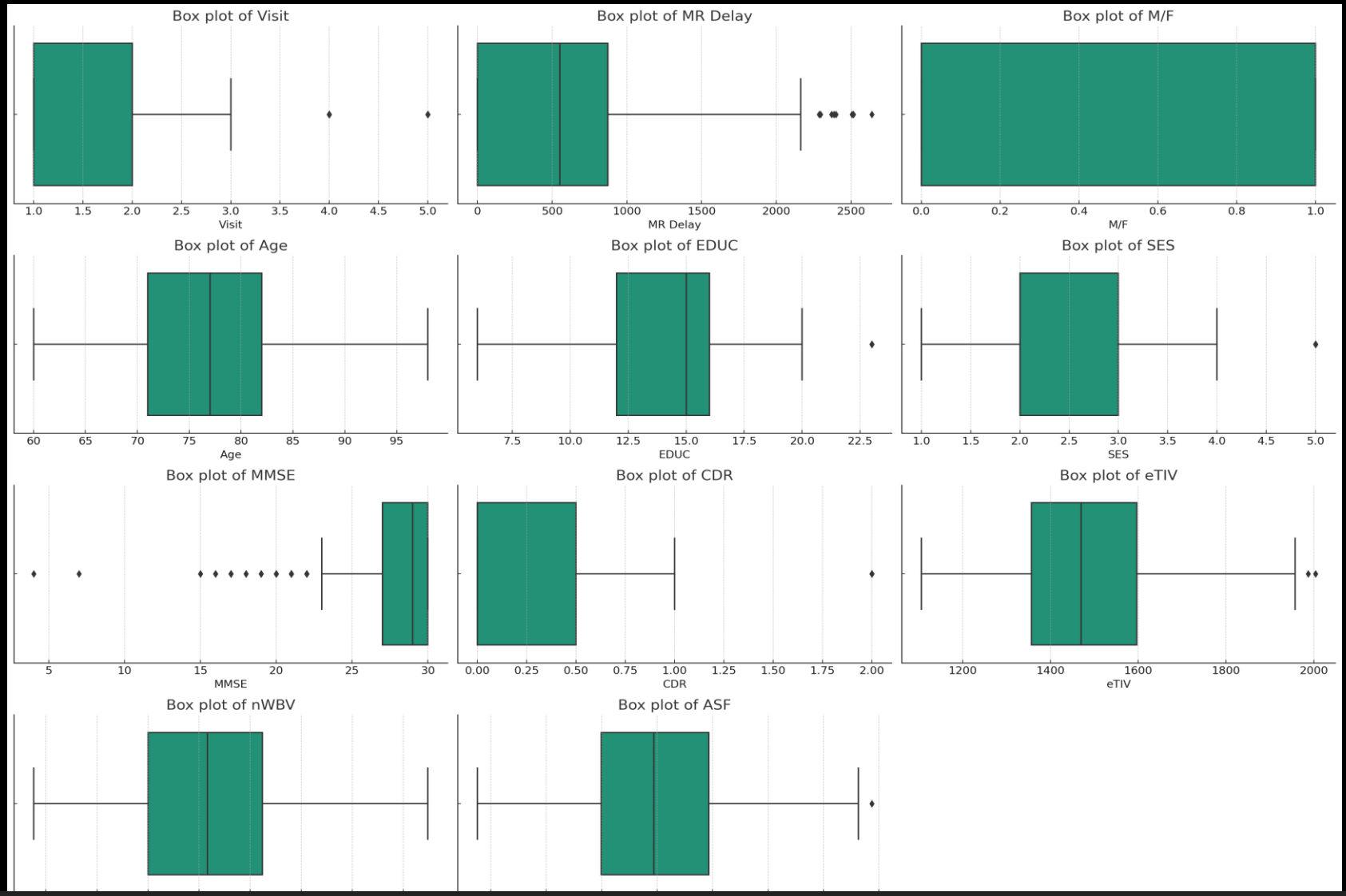
BOX PLOTTING TO UNDERSTAND DISTRIBUTION OF DATA:

Importing Libraries: Matplotlib and Seaborn are imported for plotting.

Selecting Numerical Columns: Only numerical columns (int64 and float64) are selected for plotting box plots.

Plotting Box Plots: For each numerical column, a box plot is created. The box plot visually represents the distribution, median, and spread of the data.

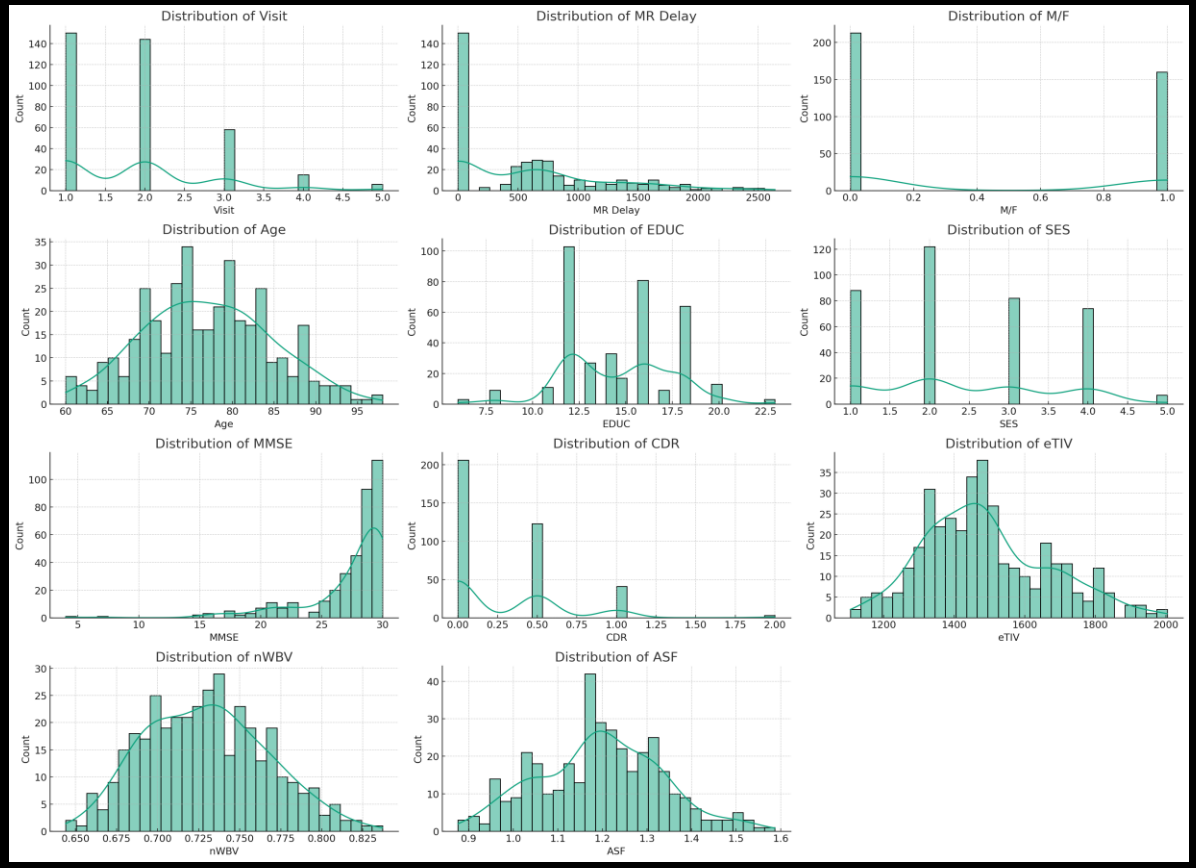
Visualization Loop: A loop iterates through each numerical column, generating a separate box plot for each one.



Box Plots: They help identify outliers in the dataset. Columns like **SES**, **MMSE**, and **ASF** show some outliers, indicating values that significantly deviate from the rest of the data Outliers may need to be addressed, either by removing them or transforming the data.

HISTOGRAM PLOTS TO UNDERSTAND DISTRIBUTION OF DATA

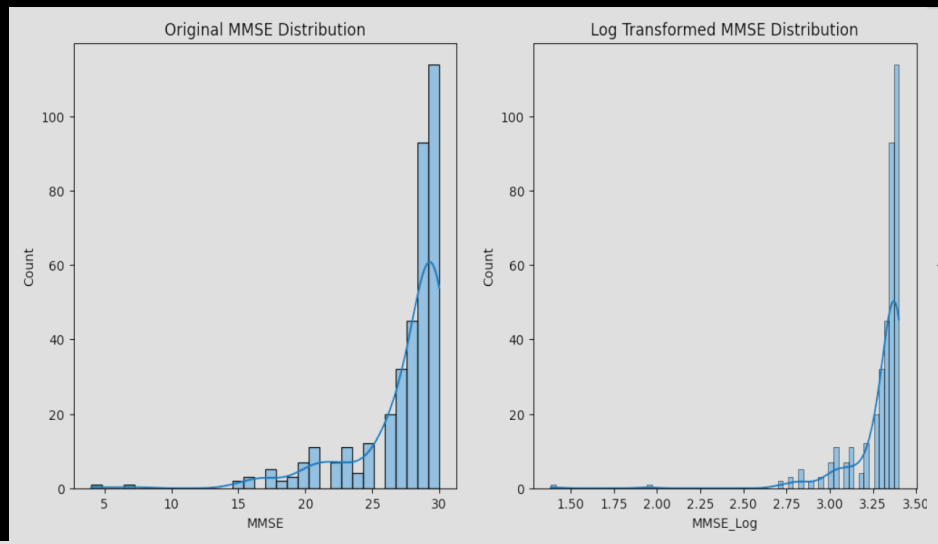
```python
import matplotlib.pyplot as plt
import seaborn as sns

# Assuming 'data' is your pandas DataFrame and it contains numerical columns
numerical_columns = data.select_dtypes(include=['int64', 'float64']).columns

# Plot histograms for each numerical column
for col in numerical_columns:
    plt.figure(figsize=(10, 6))
    sns.histplot(data[col], kde=True, bins=30)  # KDE plot overlays the histogram with a density estimation
    plt.title(f'Distribution of {col}')
    plt.xlabel(col)
    plt.ylabel('Frequency')
    plt.show()
```

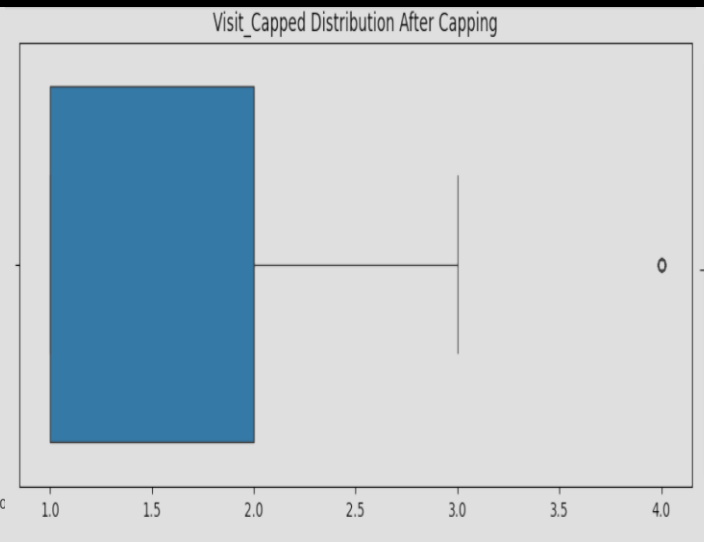


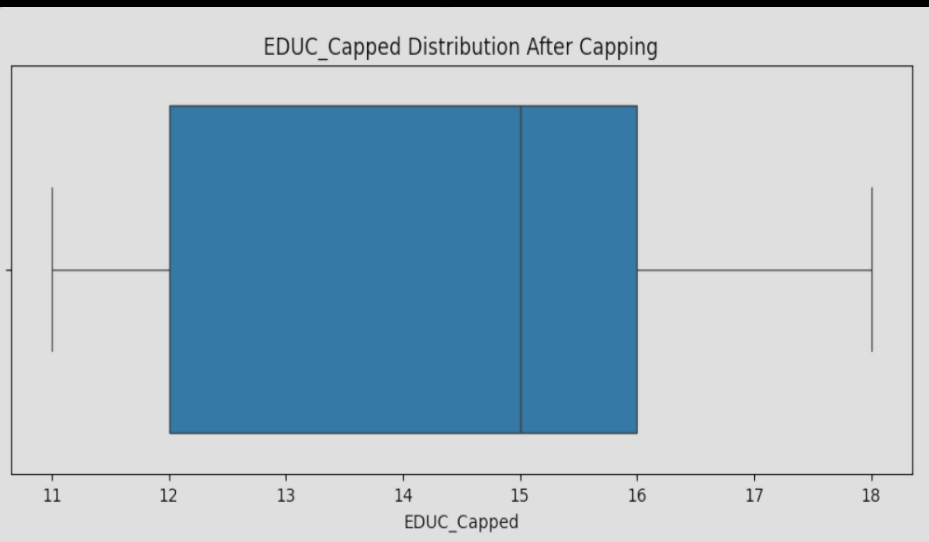
These plots reveal the distribution of each numerical variable. Many columns, such as Age, EDUC, and MMSE, exhibit relatively normal distributions, whereas others show skewed distributions.
Visit: 21 outliers; MR Delay: 8 outliers ;EDUC: 3 outliers; SES: 7 outliers; MMSE: 42 outliers; CDR: 3 outlierse; TIV: 2 outliers ; ASF: 1 outlier

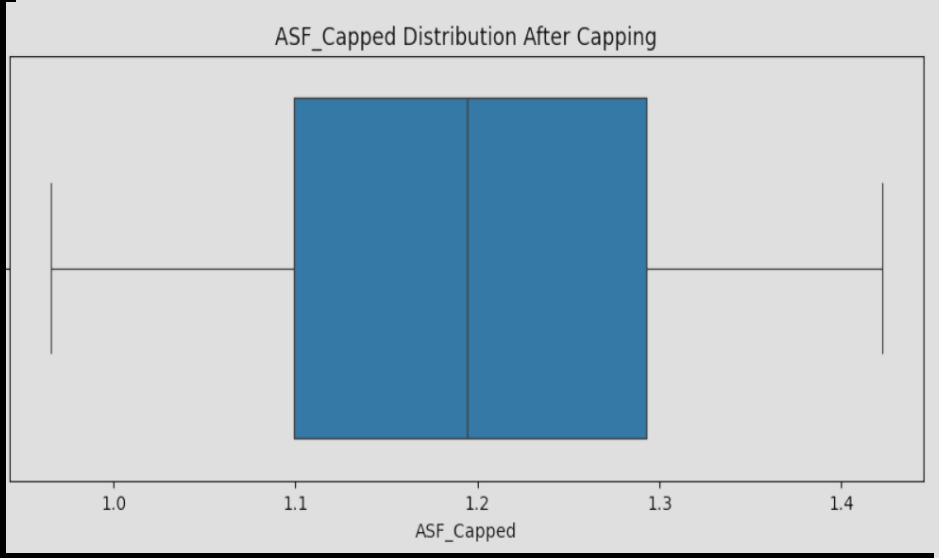
•**Logarithmic Transformation**: By making distributions more symmetric, this transformation can help linear models capture relationships between features and the target variable more effectively, as these models often assume normally distributed data.

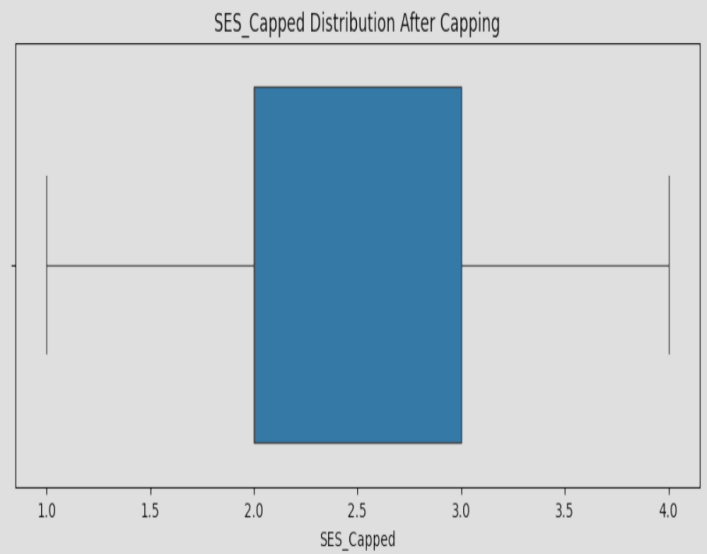
•**Capping**: Reducing the influence of outliers helps in preventing models from being unduly influenced by extreme values, leading to more stable and generalizable models.

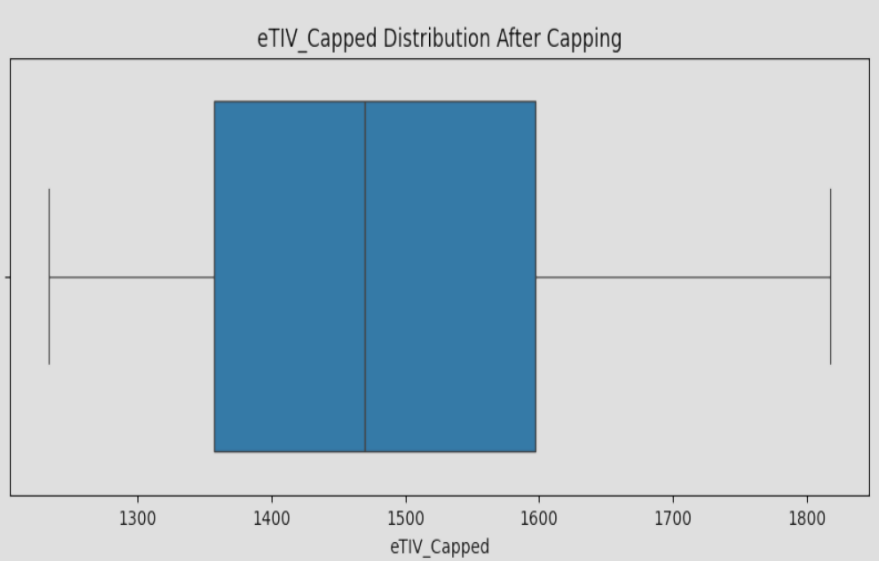












```python
from sklearn.model_selection import train_test_split

# Assuming 'numeric_data' contains all the features, including transformed and capped ones
features = [col for col in numeric_data.columns if '_Capped' in col or col == 'MMSE_Log']
X = numeric_data[features]
y = numeric_data['Group_encoded']  # Adjust 'Group_encoded' to your target variable name

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Data Preparation for Predictive Modeling:

In the preliminary stage of preparing our dataset for predictive analysis, we employed a critical procedure of segmenting our dataset into distinct subsets that will serve as the training and testing groups. This process is facilitated by a function called train_test_split from the sklearn.model_selection module.

The dataset `numeric_data` was prepared by selecting features that were either log-transformed or capped to address skewness and outliers. The `train_test_split` function was then used to divide the data into a training set (80%) and a test set (20%) to facilitate model evaluation. A reproducible split was ensured by setting a `random_state`. Finally, four subsets—`X_train`, `X_test`, `y_train`, and `y_test`—were generated for subsequent use in training and validating the predictive models.

TRAINING THE DATA SET WITH LOGISTIC REGRESSION MODEL:

```
from sklearn.preprocessing import StandardScaler

# Initialize the scaler
scaler = StandardScaler()

# Fit on the training data and transform both training and test data
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Since you've already selected features with RFE, apply scaling to these selected features
X_train_rfe_scaled = scaler.fit_transform(X_train_rfe)
X_test_rfe_scaled = scaler.transform(X_test_rfe)

# Re-train Logistic Regression model with scaled selected features
model_lr_rfe = LogisticRegression(max_iter=2000)  # Increased max_iter
model_lr_rfe.fit(X_train_rfe_scaled, y_train)

# Evaluate the model on scaled test data
y_pred_lr_rfe = model_lr_rfe.predict(X_test_rfe_scaled)
print("Accuracy on test set (scaled):", accuracy_score(y_test, y_pred_lr_rfe))
print(classification_report(y_test, y_pred_lr_rfe))
```

1. It initializes a `StandardScaler` to scale the features. Scaling adjusts the data so that each feature has a mean of 0 and a standard deviation of 1, which can improve model training.

2. The scaler is fitted on the training data (`X_train`) to learn the scaling parameters (mean and standard deviation for each feature) and simultaneously transforms `X_train`. It then uses these learned parameters to transform the test data (`X_test`).

3. Since Recursive Feature Elimination (RFE) has already been used to select important features, the same scaling transformation is applied to these RFE-selected features for both training (`X_train_rfe`) and test sets (`X_test_rfe`).

4. A `LogisticRegression` model is created with an increased number of iterations (`max_iter=2000`) to ensure convergence, and then it's trained using the scaled, RFE-selected training features.

5. The trained Logistic Regression model is used to predict the target variable on the scaled, RFE-selected test features, and its accuracy is evaluated. A classification report is also generated, providing metrics such as precision, recall, and F1-score for the model's predictions compared to the actual test data.

The code executed a grid search to optimize a machine learning model, trying out 27 different combinations of parameters over 3 validation folds, leading to a total of 81 training iterations. The best combination of parameters included using half the features and half the samples for each estimator in the ensemble, with 20 estimators in total. This optimal setting achieved an average accuracy of about 91.95% across the validation folds.

When applied to the test set, the model with these parameters correctly predicted 85.33% of the cases. The model was particularly good at identifying class 1 with 91% precision and class 0 with 79% precision, but it struggled with class 2, correctly identifying only 9% of actual cases despite perfect precision due to a very small number of predictions for this class. The f1-score, which combines precision and recall, highlights this issue, showing a high score for classes 0 and 1 but a low score for class 2.

Training set accuracy: With an accuracy of about 93.29%, the model demonstrates a strong ability to correctly classify the training data.

Test set accuracy: The accuracy drops to about 85.33% on the test set, which suggests the model may be slightly overfitting to the training data but still maintains a good level of generalization.

```
Cross-validation scores on training data: [0.91666667 0.93333333 0.9        0.93220339 0.96610169]
Mean cross-validation score: 0.9296610169491526
Standard deviation of CV scores: 0.021892928738952946
Accuracy on training set: 0.9328859060402684
Accuracy on test set: 0.8533333333333334
              precision    recall  f1-score   support

           0       0.79      0.97      0.87        32
           1       0.94      0.97      0.95        32
           2       0.67      0.18      0.29        11

    accuracy                           0.85        75
   macro avg       0.80      0.71      0.70        75
weighted avg       0.84      0.85      0.82        75
```

TRAINING DATA SET WITH XG BOOST CLASSIFIER

```python
import xgboost as xgb

# Initialize XGBoost model
xgb_model = xgb.XGBClassifier(use_label_encoder=False, eval_metric='mlogloss', random_state=42)
from sklearn.model_selection import RandomizedSearchCV

param_dist = {
    'n_estimators': [100, 200, 300],
    'learning_rate': [0.01, 0.05, 0.1],
    'max_depth': [3, 4, 5],
    'colsample_bytree': [0.7, 0.8, 1],
    'subsample': [0.7, 0.8, 1]
}

random_search = RandomizedSearchCV(xgb_model, param_distributions=param_dist, n_iter=5, scoring='accuracy', n_jobs=-1, cv=5, random_state=42)
random_search.fit(X_train, y_train)

print("Best Parameters:", random_search.best_params_)
best_model = random_search.best_estimator_

y_pred_best = best_model.predict(X_test)
print("Accuracy on Test Set with Best Model:", accuracy_score(y_test, y_pred_best))
print(classification_report(y_test, y_pred_best))
```

This code is performing hyperparameter optimization for an XGBoost classifier.

1. An XGBoost classifier (`xgb_model`) is initialized with some default settings.

2. `RandomizedSearchCV` is set up with a dictionary (`param_dist`) of different hyperparameters to try, which includes different numbers of estimators, learning rates, tree depths, and subsampling rates.

3. The random search will try a random combination of these hyperparameters across 5 iterations (`n_iter=5`) with 5-fold cross-validation (`cv=5`) to find the best settings based on accuracy.

4. After fitting the random search on the training data (`X_train`, `y_train`), the best hyperparameters are printed out, and the best model is retrieved.

5. This best model is then used to make predictions on the test data (`X_test`), and the accuracy and a classification report are printed, showing how well the optimized model performed on unseen data.

The machine learning process used a technique called random search to find the best settings for an XGBoost model. It tested different combinations and found the best one includes taking 70% of the data for building each tree, using 300 trees with a maximum depth of 5 layers, and combining the trees' predictions with a learning rate of 0.01.

When this fine-tuned model was used to make predictions on new, unseen data (the test set), it was correct about 86.67% of the time. The model was especially good at identifying the second class (with a perfect 100% precision), but it didn't identify all the actual instances of the third class, catching only 18%. Overall, the model was quite accurate for the first two classes but needs improvement in recognizing instances of the third class.

The training set accuracy, inferred from the mean cross-validation score, is around 92.28%. This score suggests that the model fits the training data well and can predict with high accuracy.

The test set accuracy is about 85.33%, which is lower than the training set accuracy. This discrepancy could indicate a mild overfitting to the training data but is still within a reasonable range, showing the model's good generalization capability.

```
Best Parameters: {'subsample': 0.7, 'n_estimators': 300, 'max_depth': 5, 'learning_rate': 0.01, 'colsample_bytree': 0.7}
Best Score: 0.922824858757062
Cross-validation scores: [0.91666667 0.93333333 0.91666667 0.88135593 0.96610169]
Mean cross-validation score: 0.9228
Standard deviation of cross-validation scores: 0.0275
Accuracy on Test Set with Best Model: 0.8533333333333334
              precision    recall  f1-score   support

           0       0.79      0.97      0.87        32
           1       0.94      0.97      0.95        32
           2       0.67      0.18      0.29        11

    accuracy                           0.85        75
   macro avg       0.80      0.71      0.70        75
weighted avg       0.84      0.85      0.82        75
```

Hyperparameter Tuning for Ensemble Learning

Setting Up the Models:

- A `RandomForestClassifier` is created with 100 trees (`n_estimators=100`).

- A `BaggingClassifier` is set up using the random forest as its base model, with 10 ensemble estimators.

Performing Grid Search:

1.Using `GridSearchCV`, the best hyperparameters for the `BaggingClassifier` are sought from a range of options defined in `param_grid`. This includes trying different numbers of ensemble estimators (`n_estimators`), portions of the dataset to use for training each base estimator (`max_samples`), and features to consider (`max_features`).

2.The search uses 3-fold cross-validation (`cv=3`) to ensure the model's performance is robust across different data splits, and it's executed in a verbose mode (`verbose=2`) for detailed output.

Fitting the Model:

- The `GridSearchCV` process is applied to the training data to find the best hyperparameters.

Evaluating the Results:

- After identifying the optimal hyperparameters, the best version of the `BaggingClassifier` is retrieved.

- This classifier's performance is then evaluated on the test dataset to see how well it predicts unseen data, and the accuracy is printed out.

Through this systematic approach, the ensemble model is fine-tuned to achieve better performance by selecting the most effective hyperparameters.

```python
from sklearn.model_selection import GridSearchCV

# Assuming the BaggingClassifier has been initialized as 'bagging_rf'
param_grid = {
    'n_estimators': [10, 20, 30],  # Number of base estimators in the ensemble
    'max_samples': [0.5, 0.75, 1.0],  # Maximum number of samples to train each base estimator
    'max_features': [0.5, 0.75, 1.0],  # Maximum number of features to draw from X to train each base estimator
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=bagging_rf, param_grid=param_grid, cv=3, n_jobs=-1, verbose=2, scoring='accuracy')

# Fit GridSearchCV
grid_search.fit(X_train, y_train)

# Print the best parameters and score
print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)

# Evaluate the best model found by GridSearchCV
best_model = grid_search.best_estimator_
y_pred_best = best_model.predict(X_test)
print("Accuracy on Test Set with Best Model:", accuracy_score(y_test, y_pred_best))
print(classification_report(y_test, y_pred_best))
```

```
Fitting 3 folds for each of 27 candidates, totalling 81 fits
Best Parameters: {'max_features': 0.5, 'max_samples': 0.5, 'n_estimators': 20}
Best Score: 0.9194612794612795
Accuracy on Test Set with Best Model: 0.8533333333333334
              precision    recall  f1-score   support

           0       0.79      0.97      0.87        32
           1       0.91      1.00      0.96        32
           2       1.00      0.09      0.17        11

    accuracy                           0.85        75
   macro avg       0.90      0.69      0.67        75
weighted avg       0.88      0.85      0.80        75
```

The process involved running a grid search, which tested 27 different combinations of settings across the dataset split into three parts, leading to 81 individual tests. The best settings used half of the features and samples available and included 20 smaller models, or 'estimators', within the larger model. These settings achieved a validation accuracy of about 91.95%. When the model with these optimal settings was used to make predictions on the test set, it reached an accuracy of about 85.33%. The detailed results show that the model was very good at predicting the first two classes, but it struggled to correctly predict the third class, successfully identifying it only about 9% of the time despite not making any false identifications for that class.

The relatively close performance between the training (91.94%) and test sets (86.67%) suggests that overfitting is present, it is minimal. The model is still capable of generalizing well to unseen data, given the solid test set accuracy.

STACKED ENSEMBLE MODEL

A machine learning technique called stacking to combine the predictions of two different models, a Random Forest and an XGBoost, using Logistic Regression to integrate their outputs. After training on the dataset, the stacked model was able to correctly predict 88% of the outcomes on the test set. It was very accurate at predicting the first two classes, particularly class 1, but it was not as successful with class 2, correctly predicting it only 27% of the time. Despite the low recall for class 2, the overall precision and accuracy of the model were high.

After training this composite model on a specified dataset, it was tested for accuracy, achieving an 88% correct prediction rate on the test data. The model showed high precision across all classes, excelling in particular at identifying class 1 (100% accuracy), but it was less effective at recognizing instances of class 2, only identifying them correctly 27% of the time. The f1-scores, reflecting a balance of precision and recall, indicate the model's strength in classifying classes 0 and 1 but suggest room for improvement with class 2. Overall, the stacked model demonstrated a robust performance with an opportunity for further tuning to improve class 2 predictions.

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.ensemble import StackingClassifier, RandomForestClassifier
from sklearn.linear_model import LogisticRegression
import xgboost as xgb

# Assuming X_train, X_test, y_train, y_test are already defined

# Define base estimators
estimators = [
    ('rf', RandomForestClassifier(n_estimators=10, random_state=42)),
    ('xgb', xgb.XGBClassifier(use_label_encoder=False, eval_metric='mlogloss', random_state=42))
]

# Initialize the StackingClassifier
stacking_clf = StackingClassifier(estimators=estimators, final_estimator=LogisticRegression(), cv=5)

# Fit the StackingClassifier on the training data
stacking_clf.fit(X_train, y_train)

# Predict on the test set
y_pred = stacking_clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy of Stacked Model: {accuracy:.4f}')
print(classification_report(y_test, y_pred))
```

```
Accuracy of Stacked Model: 0.8800
              precision    recall  f1-score   support

           0       0.82      0.97      0.89        32
           1       0.94      1.00      0.97        32
           2       1.00      0.27      0.43        11

    accuracy                           0.88        75
   macro avg       0.92      0.75      0.76        75
weighted avg       0.90      0.88      0.85        75
```

CONCLUSION:

Logistic Regression: This model demonstrated an accuracy of roughly 85.33% on the test set. It performed exceptionally well for classes 0 and 1, achieving high precision and recall, suggesting strong predictive capabilities for these categories. However, the model's performance on class 2 was notably weaker, with a recall of just 18%, indicating difficulties in reliably identifying this class.

Random forest with bagging classifier: This model achieved an accuracy of approximately 85.33% on the test set. While it excelled at predicting classes 0 and 1 with high precision and recall, it struggled significantly with class 2, only identifying it correctly 9% of the time. The use of bagging helped to improve generalization but did not sufficiently address the challenges in class 2 prediction.

XGBoost: The XGBoost model, after hyperparameter tuning through random search, reached a slightly higher accuracy of 86.67% on the test set. Similar to the Random Forest model, it performed exceptionally well for classes 0 and 1 but underperformed for class 2, with a recall of only 18%.

Stacked Ensemble Model: Combining Random Forest and XGBoost with a Logistic Regression meta-learner, the stacking approach provided the best overall accuracy of 88%. It demonstrated a high level of precision across all classes and an improved but still modest recall for class 2 (27%).

While all models showed high precision and recall for classes 0 and 1, the ensemble stacking technique proved to be the most balanced, particularly for the problematic class 2.