DocAssist(Building Intelligent Medical Decision Support System)

Problem Statement

The objective of this project is to develop an intelligent medical decision support system that analyzes patient data to assist doctors in making informed decisions about the best treatment options for individual patients. By leveraging machine learning and data analysis, the system will provide personalized treatment recommendations based on the patient's medical history, symptoms, lab results, and other relevant factors.

Solution:-

1) Libraries

In [1]:
```
# importing all necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from collections import Counter
import shap
```

1. `pandas`: Pandas is a powerful data manipulation and analysis library for Python. It provides data structures and functions necessary to work with structured data, such as data frames, which are similar to tables in a database or spreadsheet.

2. `matplotlib.pyplot`: Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. `pyplot` is a module in Matplotlib that provides a MATLAB-like interface for creating plots and visualizations.

3. `sklearn.model_selection`: This module from scikit-learn (sklearn) provides functions to split data into training and testing sets for model evaluation and selection. It includes utilities such as train_test_split for splitting datasets.

4. `sklearn.ensemble`: This module from scikit-learn contains ensemble-based learning algorithms, including Random Forest, which is a popular ensemble method for classification and regression tasks.

5. `sklearn.impute`: This module from scikit-learn provides tools for imputing missing values in datasets. The SimpleImputer class, for example, allows filling missing values with a specified strategy, such as mean, median, or most frequent value.

6. `sklearn.preprocessing`: This module from scikit-learn includes various functions for preprocessing data before fitting a machine learning model. It provides tools for scaling features, encoding categorical variables, and transforming data.

7. `sklearn.metrics`: This module from scikit-learn contains functions for evaluating the performance of machine learning models. It includes metrics such as accuracy_score, which computes the accuracy of classification models.

8. `collections.Counter`: Counter is a built-in Python class that provides a convenient way to count the occurrences of elements in a collection, such as a list or a dictionary.

9. `shap`: SHAP (SHapley Additive exPlanations) is a library for explaining the output of machine learning models. It uses Shapley values from cooperative game theory to explain the contribution of each feature to the model's prediction for a specific instance. This can help understand the model's decision-making process and interpret its predictions.

2) Data Collection

```
In [2]: # Function to collect patient's medical data from dataset
        def collect_patient_data():

            # Code to collect data from dataset
            patient_data = pd.read_excel("med_dataset.xlsx")
            return patient_data
```

The code defines a function `collect_patient_data()` that reads medical data from an Excel file named "med_dataset.xlsx" into a Pandas DataFrame and returns it.

3) Data Preprocessing

```
In [3]: # Function to preprocess patient data
        def preprocess_data(patient_data):

            # Remove duplicates
            patient_data = patient_data.drop_duplicates()

            # Impute missing values
            imputer = SimpleImputer(strategy='mean')
            patient_data['AGE'] = imputer.fit_transform(patient_data[['AGE']])

            # Initialize OneHotEncoder
            label_encoder = LabelEncoder()

            # Fit and transform the data
            patient_data['SEX'] = label_encoder.fit_transform(patient_data['SEX'])
            return patient_data
```

The code defines a function called `preprocess_data(patient_data)` intended to preprocess patient data stored in a Pandas DataFrame.

1. Removes duplicate rows from the patient data.

2. Imputes missing values in the 'AGE' column using the mean value.

3. Encodes categorical data in the 'SEX' column using LabelEncoder, converting categorical values into numerical representations.

4. Returns the preprocessed patient data DataFrame after applying the specified preprocessing steps.

4) Feature Engineering

```
In [4]: # Function to perform feature engineering
        def feature_engineering(patient_data):

            # Performing feature selection if needed
            features = patient_data.drop(columns=['SOURCE'])
            return features
```

The code defines a function called `feature_engineering(patient_data)` that performs feature engineering on patient data stored in a Pandas DataFrame.

1. Removes the 'SOURCE' column from the patient data, assuming it's not needed for further analysis or modeling.

2. Returns the modified DataFrame containing the selected features after feature engineering.

5) Model Developement

```
In [5]: def develop_model(features, labels):

            # Splitting data into train and test sets
            X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)

            # Training random forest classifier
            model = RandomForestClassifier()
            model.fit(X_train, y_train)

            # Making predictions on the testing data
            y_pred = model.predict(X_test)

            # Evaluating the model's accuracy
            accuracy = accuracy_score(y_test, y_pred)
            print('Accuracy:', accuracy)
            return model
```

The code defines a function `develop_model(features, labels)` that develops a machine learning model using a random forest classifier.

1. Splits the input features and corresponding labels into training and testing sets using the `train_test_split` function from scikit-learn. The testing set size is 20% of the data, and a random state of 42 is used for reproducibility.

2. Initializes a random forest classifier model.

3. Trains the model using the training data (`X_train` and `y_train`) with the `fit` method.

4. Makes predictions on the testing data (`X_test`) using the trained model.

5. Evaluates the accuracy of the model by comparing the predicted labels (`y_pred`) with the actual labels (`y_test`) using the `accuracy_score` function from scikit-learn.

6. Prints the accuracy of the model on the testing data.

7. Returns the trained random forest classifier model.

6) Treatment Recommendations

```
In [6]: # Function to generate treatment recommendations
        def generate_recommendations(model, patient_data):

            # Using trained model to predict treatment outcomes
            predictions = model.predict(patient_data)

            # Defining treatment options
            treatment_options = ["Treatment 0", "Treatment 1"]

            # Counting occurrences of each treatment recommendations
            recommendations_count = Counter([treatment_options[prediction] for prediction in predictions])

            # Displaying count of each treatment
            for treatment, count in recommendations_count.items():
                print(f"{treatment}: {count}")
            return recommendations_count
```

The code defines a function `generate_recommendations(model, patient_data)` that utilizes a trained machine learning model to generate treatment recommendations based on patient data.

1. Uses the trained model (`model`) to predict treatment outcomes for the given `patient_data`.

2. Defines treatment options as a list (`treatment_options`) containing the possible treatments.

3. Counts the occurrences of each treatment recommendation by applying the model's predictions and mapping them to treatment options.

4. Displays the count of each treatment recommendation.

5. Returns a dictionary (`recommendations_count`) containing the count of each treatment recommendation.

7) Model Interpretability

```
In [7]: # Function to interpret model predictions
def interpret_predictions(model, features):

    # Initializing explainer
    explainer = shap.TreeExplainer(model)

    # Calculating SHAP values
    shap_values = explainer.shap_values(features)

    # Interpreting and visualizing SHAP values
    shap.summary_plot(shap_values, features)

    # Getting feature importances
    feature_importances = model.feature_importances_

    # Getting feature names
    feature_names = features.columns

    # Sortting feature importances in descending order
    sorted_indices = feature_importances.argsort()[::-1]

    # Plotting feature importances
    plt.figure(figsize=(10, 6))
    plt.bar(range(len(feature_importances)), feature_importances[sorted_indices])
    plt.xticks(range(len(feature_importances)), [feature_names[i] for i in sorted_indices], rotation=90)
    plt.xlabel('Features')
    plt.ylabel('Importance')
    plt.title('Feature Importance')
    plt.tight_layout()
    plt.show()
```

The code defines a function `interpret_predictions(model, features)` that interprets the predictions of a machine learning model using SHAP (SHapley Additive exPlanations) values and visualizes the feature importances.

1. Initializes a SHAP explainer (`explainer`) for the given `model`.

2. Calculates SHAP values (`shap_values`) for the input features (`features`) using the explainer.

3. Interprets and visualizes the SHAP values using `shap.summary_plot()` to provide a summary of feature effects on model predictions.

4. Retrieves feature importances (`feature_importances`) from the model.

5. Retrieves feature names (`feature_names`) from the input features.

6. Sorts the feature importances in descending order.

7. Plots the feature importances using a bar plot, showing the importance of each feature in the model's predictions.

8. Displays the feature importance plot.

This function helps in understanding how each feature contributes to the model's predictions and provides insights into the model's behavior.

8) Main Function

```
In [8]: # Main function
def main():

    # Collectting patient data
    patient_data = collect_patient_data()

    # Preprocessing patient data
    processed_data = preprocess_data(patient_data)

    # Performming feature engineering
    features = feature_engineering(processed_data)

    # Assuming 'labels' are available for training
    labels = processed_data['SOURCE']

    # Developing machine learning model
    model = develop_model(features, labels)

    # Generating treatment recommendations
    recommendations = generate_recommendations(model, features)

    # Interpreting model predictions
    interpret_predictions(model, features)
```

The code defines a main function `main()` that orchestrates the entire process of collecting patient data, preprocessing it, performing feature engineering, developing a machine learning model, generating treatment recommendations based on the model, and interpreting the model predictions.

1. Collects patient data using the `collect_patient_data()` function.

2. Preprocesses the collected patient data using the `preprocess_data()` function.

3. Performs feature engineering on the preprocessed data using the `feature_engineering()` function.

4. Assumes that labels for training are available.

5. Develops a machine learning model using the `develop_model()` function.

6. Generates treatment recommendations based on the developed model using the `generate_recommendations()` function.

7. Interprets the model predictions and visualizes feature importances using the `interpret_predictions()` function.

This main function encapsulates the entire workflow from data collection to model interpretation, providing a structured and modular approach to the analysis process.

9) Dispalying the result

```
In [9]: if __name__ == "__main__":
            main()
```

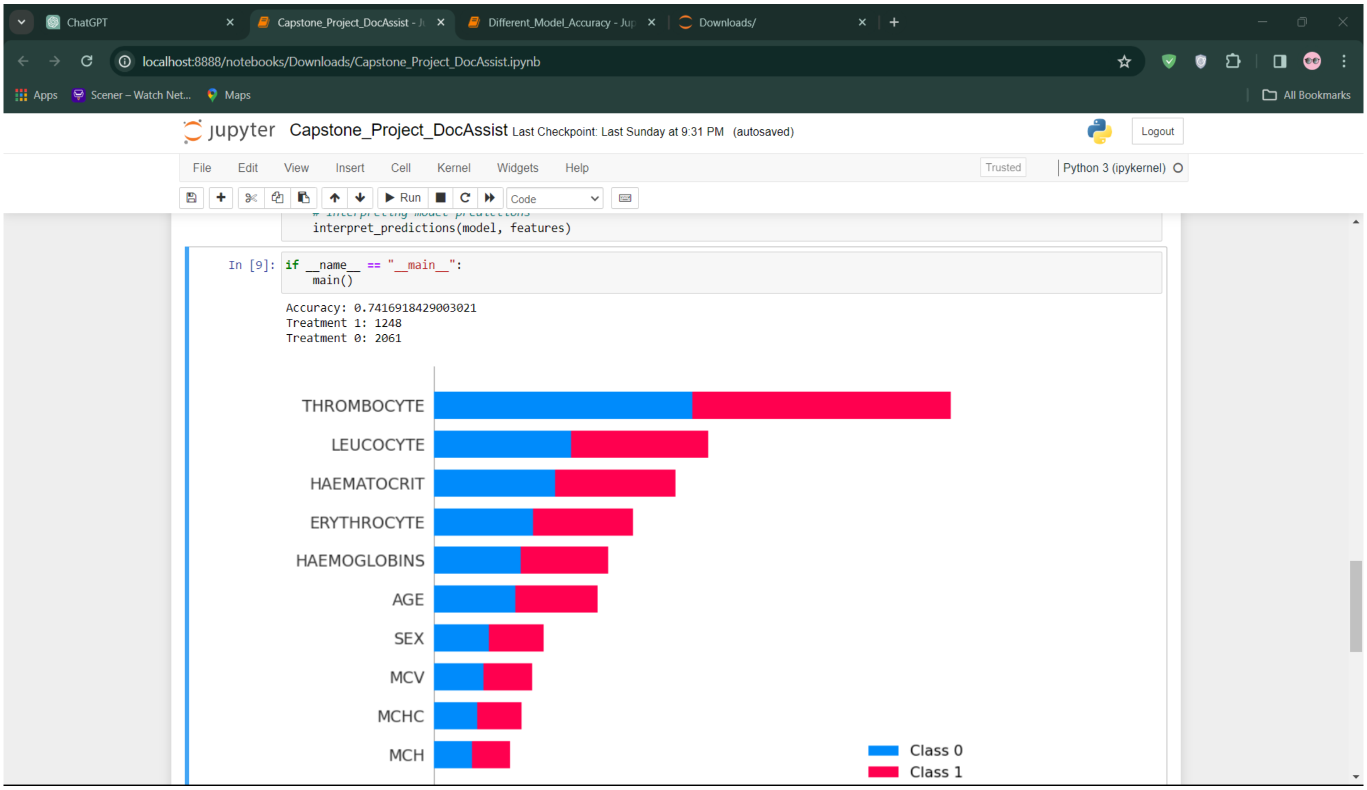
The code checks if the script is being run as the main program (`__name__ == "__main__"`). If it is, it calls the `main()` function, which orchestrates the entire process of collecting patient data, preprocessing it, developing a machine learning model, generating treatment recommendations, and interpreting model predictions.

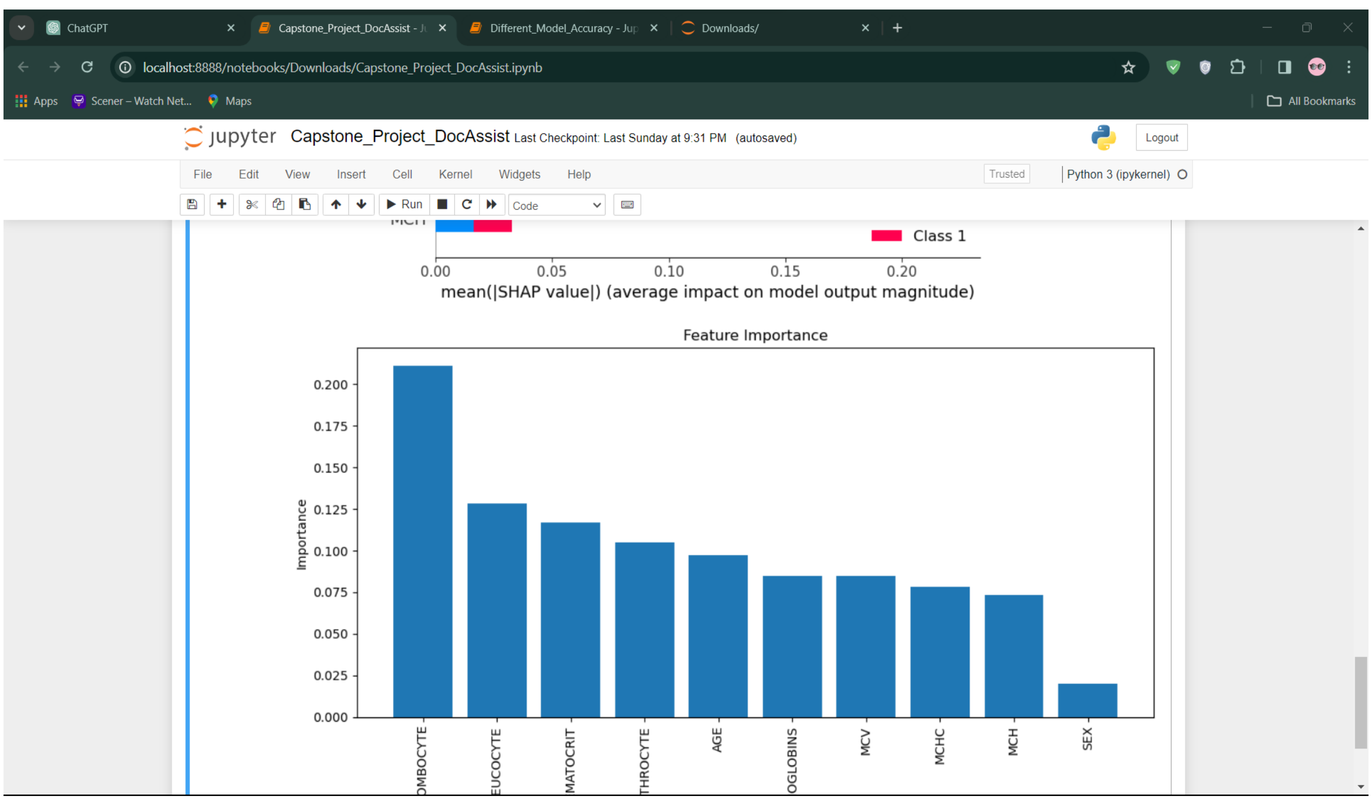
- The `if __name__ == "__main__":` statement ensures that the `main()` function is executed only when the script is run directly, not when it's imported as a module into another script.

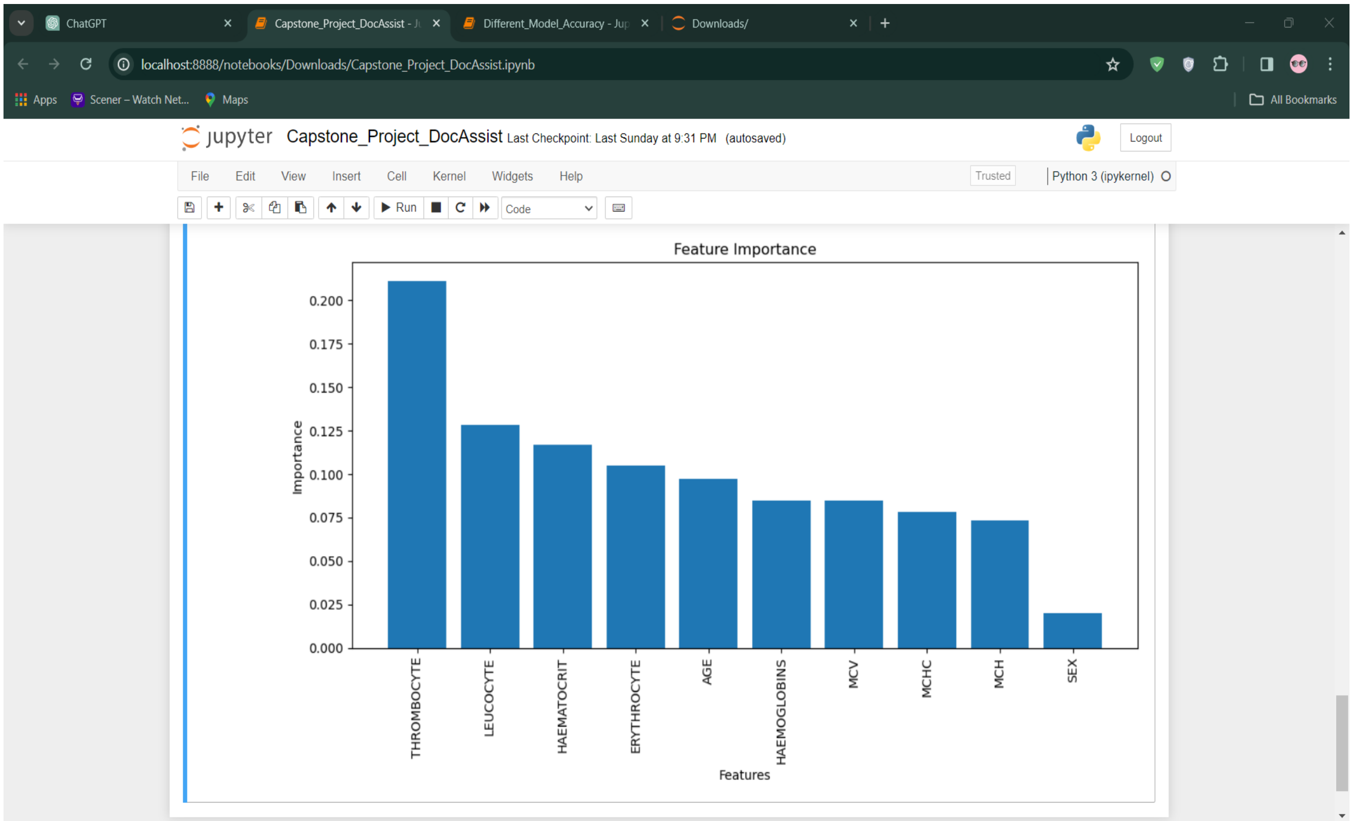
- If the script is run directly, it calls the `main()` function, initiating the execution of the entire workflow defined within the `main()` function.

- This structure allows the script to be reusable as a module in other scripts without automatically running the main function when imported. Instead, it provides the flexibility to use the functions and classes defined in the script without executing the main workflow.

10) The Bar Chart and Output of the DocAssist







This is the result of the code.

11) Different Model With Accuracy

1:- Decision Tree Classifier

```
Different Models and their accuracy.
```

```
In [6]: import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load the dataset into a DataFrame (replace 'data.csv' with your dataset filename)
data = pd.read_excel('dataset.xlsx')

# Split the dataset into features (X) and target variable (y)
X = data.drop(columns=['SEX','SOURCE'])
y = data['SOURCE']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the decision tree classifier
clf = DecisionTreeClassifier()

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = clf.predict(X_test)

# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.648036253776435
```

2:- Random Forest Classifier

In [7]:
```
#Random Forest

from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load the dataset into a DataFrame (replace 'data.csv' with your dataset filename)
data = pd.read_excel('dataset.xlsx')

# Split the dataset into features (X) and target variable (y)
X = data.drop(columns=['SEX','SOURCE'])
y = data['SOURCE']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the Random Forest classifier
clf = RandomForestClassifier()

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = clf.predict(X_test)

# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.7356495468277946
```

3:- Logistic Regression

In [8]:
```
#Logistic Regression


import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Load the dataset into a DataFrame
data = pd.read_excel('dataset.xlsx')

# Split the dataset into features (X) and target variable (y)
X = data.drop(columns=['SEX','SOURCE'])
y = data['SOURCE']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the logistic regression model
clf = LogisticRegression()

# Train the model on the training data
clf.fit(X_train, y_train)

# Make predictions on the testing data
y_pred = clf.predict(X_test)

# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.7039274924471299
```

4:- Neural Network Model

```
In [9]: #Neural Network Model

        import pandas as pd
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import StandardScaler
        from sklearn.neural_network import MLPClassifier
        from sklearn.metrics import accuracy_score

        # Load the dataset into a DataFrame
        data = pd.read_excel('dataset.xlsx')

        # Split the dataset into features (X) and target variable (y)
        X = data.drop(columns=['SEX','SOURCE'])
        y = data['SOURCE']

        # Split the data into training and testing sets
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

        # Standardize the features by scaling
        scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_test_scaled = scaler.transform(X_test)

        # Initialize the neural network model
        # Specify the architecture of the neural network using the hidden_layer_sizes parameter
        # Here, we have one hidden layer with 100 neurons
        # You can adjust the number of neurons and layers based on your data and task
        clf = MLPClassifier(hidden_layer_sizes=(100,), random_state=42)

        # Train the model on the scaled training data
        clf.fit(X_train_scaled, y_train)

        # Make predictions on the scaled testing data
        y_pred = clf.predict(X_test_scaled)

        # Evaluate the model's accuracy
        accuracy = accuracy_score(y_test, y_pred)
```

```
# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.729607250755287
```

5:- KNN

In [10]:
```
#KNN

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load the dataset into a DataFrame
# Load the dataset into a DataFrame
data = pd.read_excel('dataset.xlsx')

# Split the dataset into features (X) and target variable (y)
X = data.drop(columns=['SEX','SOURCE'])
y = data['SOURCE']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features by scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Initialize the KNN classifier
clf = KNeighborsClassifier(n_neighbors=5)

# Train the model on the scaled training data
clf.fit(X_train_scaled, y_train)

# Make predictions on the scaled testing data
y_pred = clf.predict(X_test_scaled)

# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)
print('KNN Accuracy:', accuracy)
```


```
# Evaluate the model's accuracy
accuracy = accuracy_score(y_test, y_pred)
print('KNN Accuracy:', accuracy)
```


```
KNN Accuracy: 0.7084592145015106
```