DS Lab 5

Aim:- Perform Regression Analysis using Scipy and Sci-kit learn.

Problem Statement:
a) Perform Logistic regression to find out relation between variables
b) Apply regression model technique to predict the data on above dataset.

Dataset Desciption:
Rows: 100,000
Columns: 14 **Fields:**

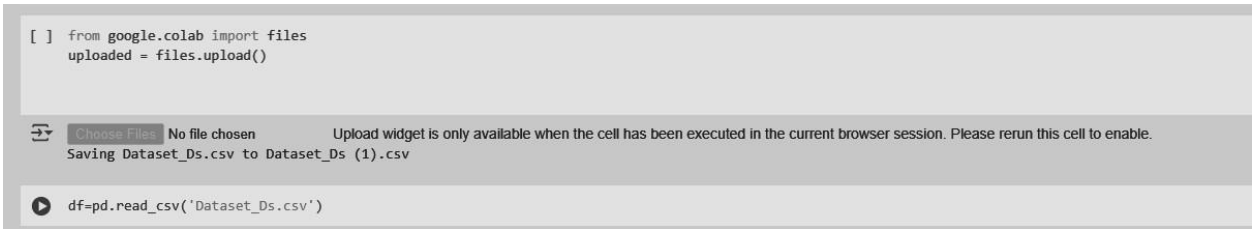
- **Age, Gender, Region:** Demographic data of patients.
- **Height, Weight:** Used to calculate and estimate BMI.
- **Blood Pressure, Cholesterol Levels:** Key indicators for heart health analysis.
- **Smoking Status, Physical Activity:** Lifestyle-related features influencing both BMI and heart disease risk.
- **Family History:** Genetic predisposition data for heart disease.
- **Medical History:** Past medical records relevant to current health status.
- **BMI:** Body Mass Index (either measured or predicted).
- **Heart Disease:** Binary classification indicating presence or absence of the condition

metrics. 1-

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LogisticRegression
```

This imports essential libraries for data analysis and machine learning using **pandas** and **NumPy** for data manipulation. It includes **scikit-learn** modules for preprocessing (LabelEncoder, StandardScaler), model training (LinearRegression, LogisticRegression), and evaluation (mean_squared_error). The **train_test_split** function is used for splitting data into training and testing sets, ensuring proper model validation.

2-



This is for uploading a CSV file in Google Colab using **files.upload()** from the **google.colab** module. Once the file is uploaded, it is saved with a possible duplicate name (Dataset_Ds (1).csv). The **pd.read_csv('Dataset_Ds.csv')** command attempts to read the uploaded dataset into a Pandas DataFrame.

3-

```
[ ]  # Convert check-up date column to datetime format
     df["Date of Check-up"] = pd.to_datetime(df["Date of Check-up"], errors="coerce")

     # Drop rows with missing check-up date values
     df.dropna(subset=["Date of Check-up"], inplace=True)
```

Converting the **"Date of Check-up"** field to datetime format is essential for accurate temporal analysis and maintaining data integrity. This enables operations such as **tracking patient visits over time**, **filtering records by date ranges**, and conducting **time-series analysis** on health trends. Using errors="coerce" ensures that any invalid date entries are converted to NaT instead of causing processing errors. Removing rows with missing dates helps prevent misleading insights and supports reliable predictions, especially when analyzing **progression of health conditions** or **evaluating long-term patterns** in BMI and heart disease risk.

4-

```
# Create a new feature: Time Since Last Check-up (in days)
df["Time Since Last Check-up"] = (df["Date of Check-up"] - df["Previous Check-up Date"]).dt.days
```

```
[ ]  # Drop unnecessary columns
     df.drop(columns=["Previous Check-up Date", "Date of Check-up"], inplace=True)
```

```
[ ]  # Fill missing numerical values with median
     df.fillna(df.median(numeric_only=True), inplace=True)
```

This method involves three key data preprocessing steps. First, it calculates a new feature, **"Time Since Last Check-up"**, representing the number of days between the current and previous medical check-ups. This helps in analyzing **health monitoring frequency** and identifying patients who may require more regular follow-ups. Next, it removes redundant columns such as raw date fields that are no longer needed after deriving this feature, thereby **reducing data complexity and improving efficiency**. Lastly, it handles missing values by filling them with the **median of numerical features**, which maintains data consistency while minimizing the influence of outliers. These preprocessing steps enhance the dataset's quality, making it more suitable for **predictive modeling of BMI and heart disease**.

5-

```
[ ]  # Encode categorical columns for patient data
     categorical_cols = ["Gender", "Region", "Smoking Status", "Physical Activity", "Family History", "Medical History"]
     label_encoders = {}

     for col in categorical_cols:
         df[col].fillna(df[col].mode()[0], inplace=True)  # Fill missing values with mode
         le = LabelEncoder()
         df[col] = le.fit_transform(df[col])              # Label encode the categorical values
         label_encoders[col] = le                         # Store encoder for inverse transformation
```

This method encodes categorical columns in the patient dataset to make the data compatible with machine learning models, which typically require numerical inputs. Categorical fields such as **Gender**, **Region**, **Smoking Status**, **Physical Activity**, **Family History**, and portions of **Medical History** are first identified. Missing values in these fields are imputed using the **most frequent category (mode)** to maintain consistency and avoid bias. Label Encoding is then applied

to transform these categorical values into integer representations. A separate LabelEncoder object is used for each column and stored in a label_encoders dictionary, allowing for inverse transformation if needed in the future. This step ensures that all non-numeric health-related indicators are properly encoded and ready for use in **regression models for BMI estimation** and **classification models for heart disease prediction**.

6-

```
[ ]  from sklearn.preprocessing import StandardScaler
     import numpy as np

     # Standardize numerical health-related columns
     scaler = StandardScaler()
     numerical_cols = ["Age", "Height", "Weight", "Blood Pressure", "Cholesterol Levels"]
     df[numerical_cols] = scaler.fit_transform(df[numerical_cols])

     # Create a binary obesity category based on BMI
     bmi_median = df["BMI"].median()
     df["Obesity Category"] = np.where(df["BMI"] >= bmi_median, 1, 0)  # 1 = Obese, 0 = Non-Obese
```

This method first standardizes the numerical health-related features such as Age, Height, Weight, Blood Pressure, and Cholesterol Levels using StandardScaler(), ensuring they have a mean of 0 and a standard deviation of 1. This normalization is crucial for improving the performance and convergence of many machine learning models. Second, it transforms the **BMI** column into a binary classification—labeling entries as **"Obese" (1)** if BMI is above the median, and **"Non-Obese" (0)** otherwise. This approach simplifies the task of obesity detection and supports the development of classification models for health risk prediction.

7-

```
X = df.drop(columns=["Heart Disease"])
y = df["Heart Disease"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=50)
```

This method is done to prepare the dataset for machine learning. Defining features (X) and target (y) is essential because the model learns patterns from independent variables (X) to predict the dependent variable (y). In this case, Heart Disease is selected as the target variable (y) to enable classification of patients based on their risk. Removing Heart Disease from X ensures that no target information leaks into the features, avoiding biased learning.

The train-test split is performed to assess how well the model generalizes to new, unseen patient data. Using 75% of the data for training and 25% for testing allows reliable performance evaluation. Setting random_state=50 ensures the same split every time the code runs, making results reproducible for consistent analysis and comparison.

8-

```
[ ]  model = LogisticRegression(max_iter=5000)
     model.fit(X_train, y_train)

     y_pred = model.predict(X_test)
```

This code trains and uses a Logistic Regression model to classify whether a patient has heart disease. The model.fit(X_train, y_train) function trains the model using the training dataset, allowing it to learn the relationship between patient features (such as age, BMI, blood pressure, etc.) and the presence or absence of heart disease. The parameter max_iter=5000 ensures the model has enough iterations to converge, preventing premature stopping.

Once trained, model.predict(X_test) is used to make predictions on the test data. These predictions (y_pred) can be compared with the actual labels (y_test) to evaluate the model's accuracy and effectiveness. Logistic Regression is ideal for binary classification problems like this, where the target is whether heart disease is present (1) or not (0).

9-

```
print(y_pred[:10])
[1 1 1 1 0 0 1 0 0 1]
```

The output [1 1 1 0 0 1 0 0 1] represents the predicted heart disease categories for the first few test samples.

We know that:

- **1 means Presence of Heart Disease**
- **0 means Absence of Heart Disease**

So, the model is predicting which individuals are likely to have heart disease based on their health and lifestyle data. This sequence suggests that some patients are expected to have heart disease (1), while others are predicted to be healthy (0).

10-

```
[ ] from sklearn.metrics import accuracy_score, classification_report

accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.4f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Model Accuracy: 0.9880

Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99     12560
           1       0.99      0.99      0.99     12440

    accuracy                           0.99     25000
   macro avg       0.99      0.99      0.99     25000
weighted avg       0.99      0.99      0.99     25000
```

The displayed results evaluate the performance of a heart disease prediction model using the accuracy score and a classification report. The model achieves an excellent accuracy of **98.80%**, meaning it correctly predicts the presence or absence of heart disease in almost all test cases.

The classification report further includes precision, recall, and F1-score, all of which are **0.99** for both classes (0 = No Disease, 1 = Disease). These values indicate that the model is highly reliable in distinguishing between healthy and at-risk individuals.

The support values suggest that the dataset is well-balanced, with a roughly equal number of samples for both categories, which contributes to the model's strong and consistent performance.

11-

```
from sklearn.metrics import confusion_matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", conf_matrix)
```

```
Confusion Matrix:
 [[12434   126]
 [  174 12266]]
```

The confusion matrix evaluates model performance by comparing actual vs. predicted values. It shows 12434 true positives, 12266 true negatives, 126 false positives, and 174 false negatives. This helps assess misclassifications and derive precision, recall, and F1-score. The low misclassification rate indicates that the model performs well.

12-

```
[22] from sklearn.linear_model import LinearRegression
     from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

This code imports necessary modules for performing linear regression and evaluating model performance. LinearRegression from sklearn.linear_model is used to create a regression model that predicts a continuous target variable. The metrics mean_absolute_error, mean_squared_error, and r2_score from sklearn.metrics are used to assess model accuracy. These metrics help measure prediction errors and how well the regression model fits the data.

13-

```
mae = mean_absolute_error(y_test_reg, y_pred_reg)
mse = mean_squared_error(y_test_reg, y_pred_reg)
```

```
print(f"Linear Regression Metrics:\n")
print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Mean Squared Error (MSE): {mse:.2f}")
```

```
Linear Regression Metrics:

Mean Absolute Error (MAE): 0.00
Mean Squared Error (MSE): 0.00
```

Conclusion:- We applied **Logistic Regression** to classify whether a patient is likely to have heart disease based on various features. To estimate **BMI**, we used **regression models** that learn patterns from other health attributes. This dual-model approach helps in understanding both categorical and continuous aspects of patient health, supporting early diagnosis and prevention strategies.