Name : Vedant Pawar

PRN : 202201040094

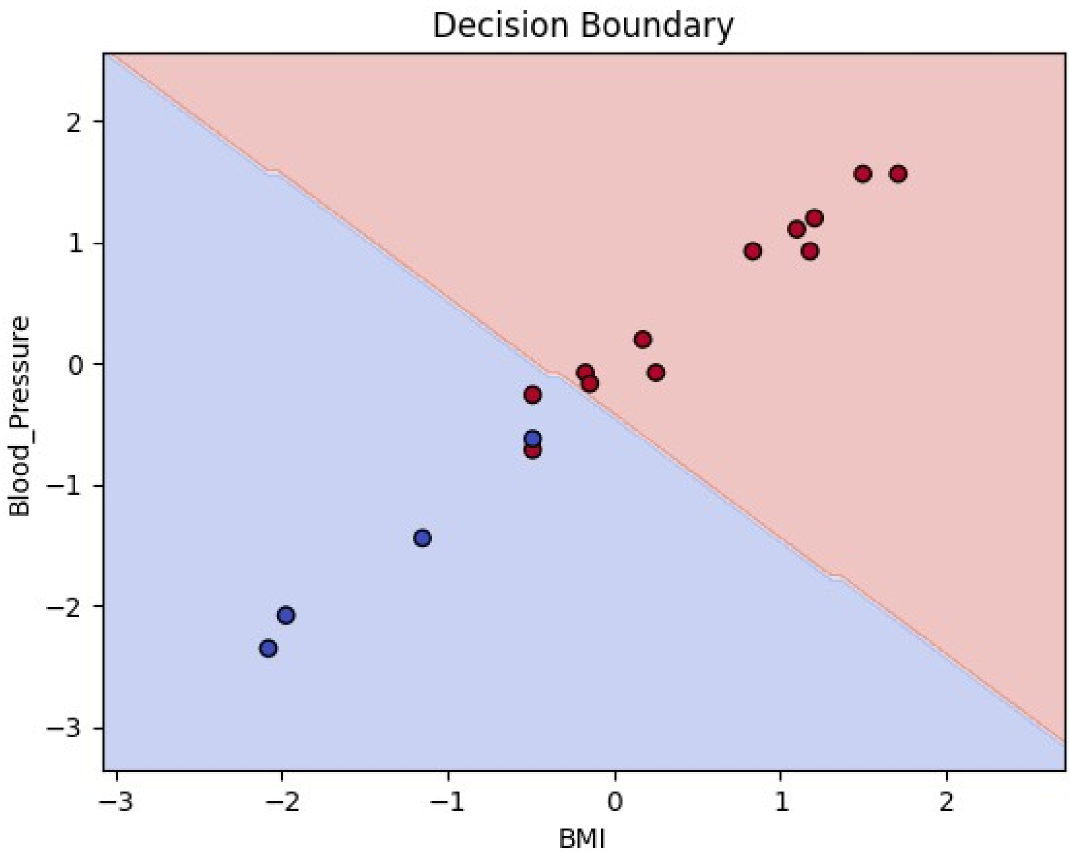
Batch : T(4)

Logistic Regression from Scratch

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

class LogisticRegressionScratch:
    def __init__(self, learning_rate=0.01, iterations=1000):
        self.learning_rate = learning_rate
        self.iterations = iterations
        self.weights = None
        self.bias = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        m, n = X.shape
        self.weights = np.zeros(n)
        self.bias = 0

        for _ in range(self.iterations):
            linear_model = np.dot(X, self.weights) + self.bias
            y_pred = self.sigmoid(linear_model)

            dw = (1/m) * np.dot(X.T, (y_pred - y))
            db = (1/m) * np.sum(y_pred - y)

            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_pred = self.sigmoid(linear_model)
        return (y_pred >= 0.5).astype(int)

# Load dataset
file_path = "/content/Records-Patient.csv"
df = pd.read_csv(file_path)
```

```python
# Selecting two features for visualization
target_col = 'Risk_Level (Target)'
features = ['BMI', 'Blood_Pressure']  # Change these as needed
X = df[features]
y = df[target_col]

# Splitting dataset
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardizing features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Training the model
model = LogisticRegressionScratch(learning_rate=0.1, iterations=1000)
model.fit(X_train, y_train)

# Predictions and accuracy
y_pred = model.predict(X_test)
accuracy = np.mean(y_pred == y_test)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Plotting decision boundary
def plot_decision_boundary(X, y, model):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                         np.linspace(y_min, y_max, 100))
    grid = np.c_[xx.ravel(), yy.ravel()]
    Z = model.predict(grid)
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.3, cmap='coolwarm')
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k',
cmap='coolwarm')
    plt.xlabel(features[0])
    plt.ylabel(features[1])
    plt.title('Decision Boundary')
    plt.show()

plot_decision_boundary(X_test, y_test.values, model)
```

```
Accuracy: 87.50%
```

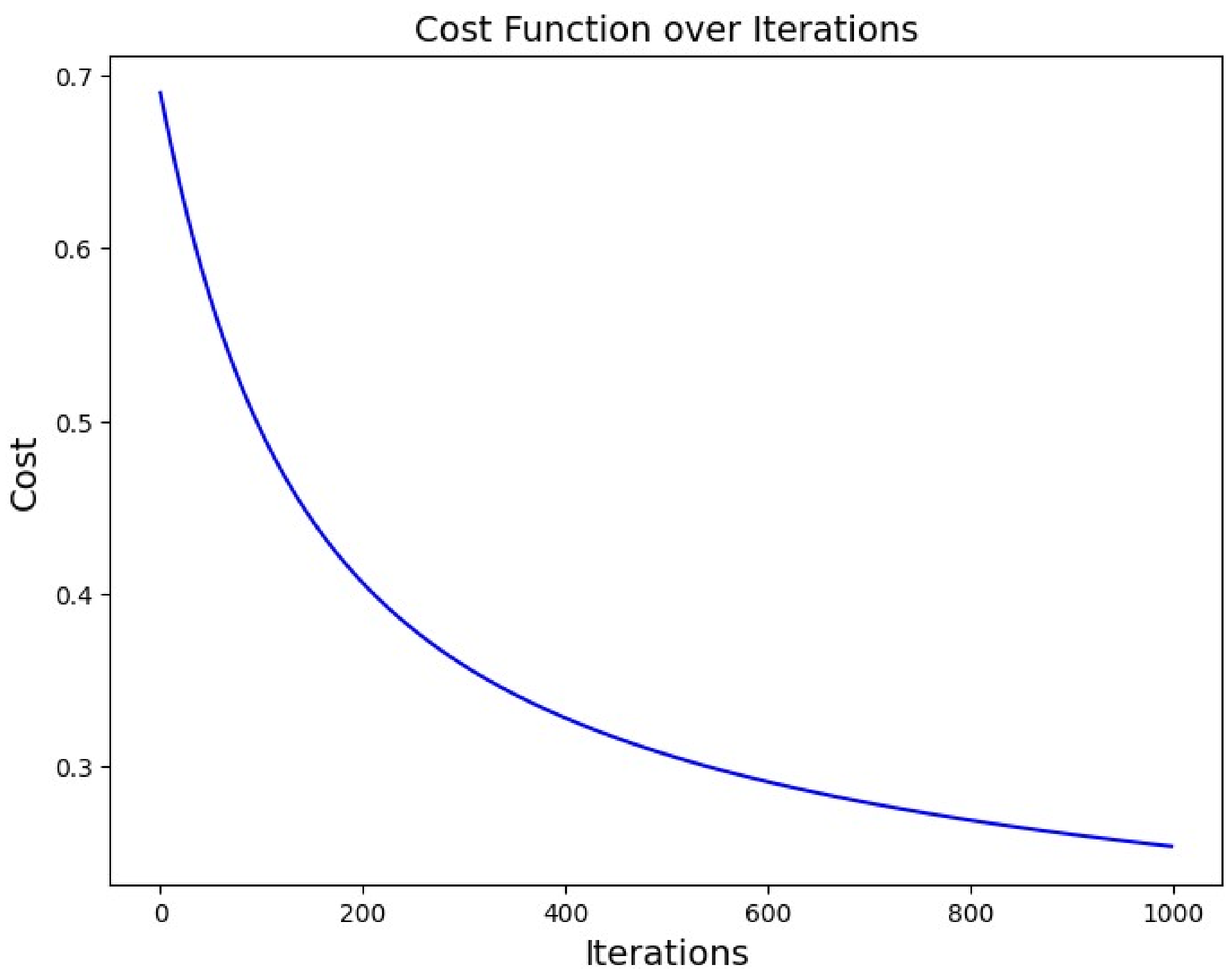



```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Logistic function (sigmoid)
def logistic(x):
    return 1 / (1 + np.exp(-x))

# Log loss function
def log_loss(y, y_dash):
    return - (y * np.log(y_dash)) - ((1 - y) * np.log(1 - y_dash))

# Cost function using vectorization
def cost_func_vec(y, y_dash):
    m = len(y)
    loss_vec = np.array([log_loss(y[i], y_dash[i]) for i in range(m)])
    cost = np.dot(loss_vec, np.ones(m)) / m
    return cost

# Cost function in terms of model parameters (using vectorization)
def cost_logreg_vec(X, y, w, b):
    m, n = X.shape
    z = np.matmul(X, w) + b
```

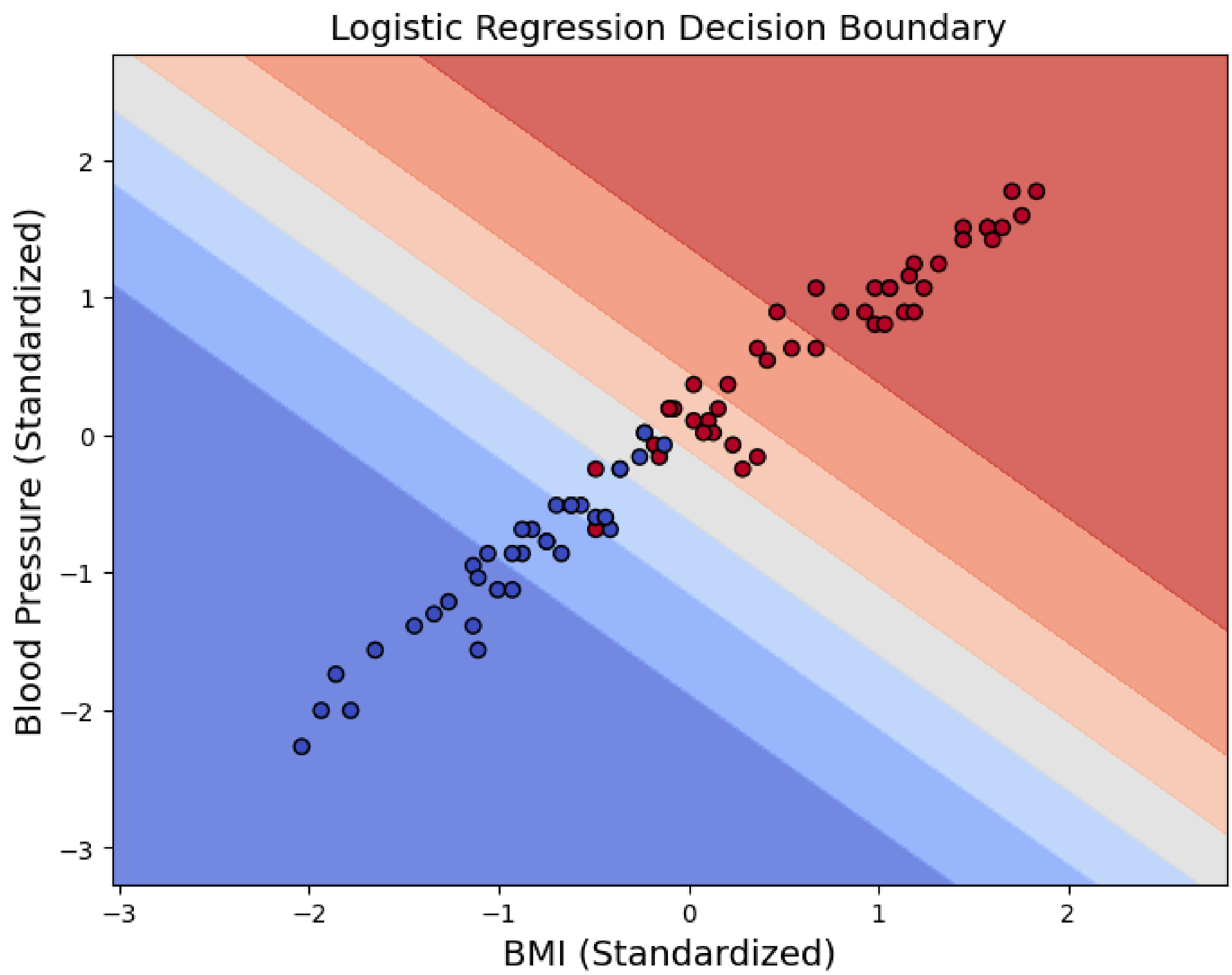
```
    y_dash = logistic(z)
    return cost_func_vec(y, y_dash)

# Gradient computation
def compute_gradients(X, y, w, b):
    m = len(y)
    z = np.matmul(X, w) + b
    y_dash = logistic(z)

    dw = np.dot(X.T, (y_dash - y)) / m
    db = np.sum(y_dash - y) / m

    return dw, db

# Gradient Descent for logistic regression
def gradient_descent(X, y, w, b, learning_rate, iterations):
    cost_history = []

    for i in range(iterations):
        dw, db = compute_gradients(X, y, w, b)

        # Update weights and bias
        w -= learning_rate * dw
        b -= learning_rate * db

        # Compute cost after updating
        cost = cost_logreg_vec(X, y, w, b)
        cost_history.append(cost)

    return w, b, cost_history

# Full logistic regression model
def logistic_regression(X, y, learning_rate=0.01, iterations=1000):
    m, n = X.shape
    w = np.zeros(n)  # Initialize weights as zero
    b = 0            # Initialize bias as zero

    w, b, cost_history = gradient_descent(X, y, w, b, learning_rate,
iterations)

    return w, b, cost_history

# Load dataset
file_path = "/content/Records-Patient.csv"
df = pd.read_csv(file_path)

# Selecting two features for visualization and the target
features = ['BMI', 'Blood_Pressure']  # Select two features for
visualization
target_col = 'Risk_Level (Target)'
```

```python
X = df[features].values
y = df[target_col].values

# Normalize the features
X_mean = np.mean(X, axis=0)
X_std = np.std(X, axis=0)
X = (X - X_mean) / X_std

# Train the logistic regression model
learning_rate = 0.01
iterations = 1000
w, b, cost_history = logistic_regression(X, y, learning_rate,
iterations)

# Plotting the cost function history
plt.figure(figsize=(8, 6))
plt.plot(range(len(cost_history)), cost_history, color='blue')
plt.xlabel("Iterations", fontsize=14)
plt.ylabel("Cost", fontsize=14)
plt.title("Cost Function over Iterations", fontsize=14)
plt.show()

# Predict function
def predict(X, w, b):
    z = np.matmul(X, w) + b
    return logistic(z)

# Example prediction
predictions = predict(X, w, b)
predictions = (predictions >= 0.5).astype(int)
print("Predictions:", predictions)

# Plotting the decision boundary
def plot_decision_boundary(X, y, w, b):
    plt.figure(figsize=(8, 6))

    # Create grid points to plot decision boundary
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                         np.linspace(y_min, y_max, 100))

    Z = predict(np.c_[xx.ravel(), yy.ravel()], w, b)
    Z = Z.reshape(xx.shape)

    # Plot decision boundary
    plt.contourf(xx, yy, Z, alpha=0.8, cmap='coolwarm')

    # Plot data points
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o',
```

```
cmap='coolwarm')

    plt.xlabel('BMI (Standardized)', fontsize=14)
    plt.ylabel('Blood Pressure (Standardized)', fontsize=14)
    plt.title("Logistic Regression Decision Boundary", fontsize=14)
    plt.show()

# Plot the decision boundary
plot_decision_boundary(X, y, w, b)
```




```
Predictions: [0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 0 0 1 0 1 0 1 0 0 1 0 1 0
1 1 1 0 1 0 1 1 0
 1 1 0 1 0 1 1 1 1 0 1 1 0 1 0 1 1 1 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0
1 1
 1 1 0 1 1 0]
```



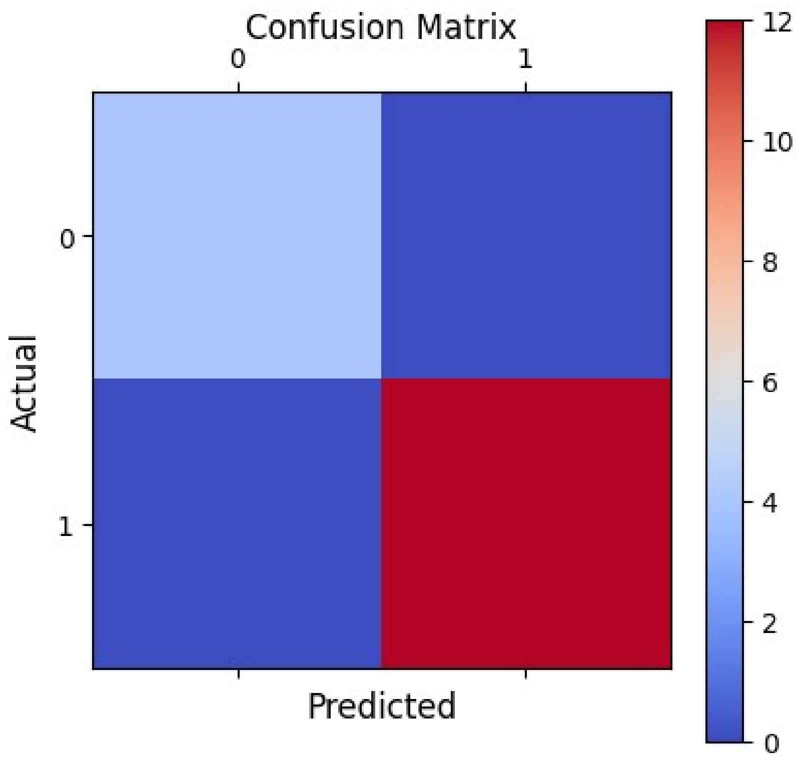
Logistic Regression with Library

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix

# Load the patient records dataset
file_path = "/content/Records-Patient.csv"  # Update the file path if
necessary
df = pd.read_csv(file_path)

# Features and target selection
X = df[['Age', 'BMI', 'Blood_Pressure', 'Cholesterol', 'Smoking',
```

```python
'Family_History']].values
y = df['Risk_Level (Target)'].values  # Target column

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

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize and train the Logistic Regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

# Display results
print(f'Accuracy: {accuracy:.2f}')
print('Confusion Matrix:\n', conf_matrix)
print('Classification Report:\n', class_report)

# Plot confusion matrix
plt.matshow(conf_matrix, cmap='coolwarm')
plt.title('Confusion Matrix', pad=20)
plt.colorbar()
plt.xlabel('Predicted', fontsize=12)
plt.ylabel('Actual', fontsize=12)
plt.show()
```

```
Accuracy: 1.00
Confusion Matrix:
 [[ 4  0]
 [ 0 12]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         4
           1       1.00      1.00      1.00        12

    accuracy                           1.00        16
   macro avg       1.00      1.00      1.00        16
```

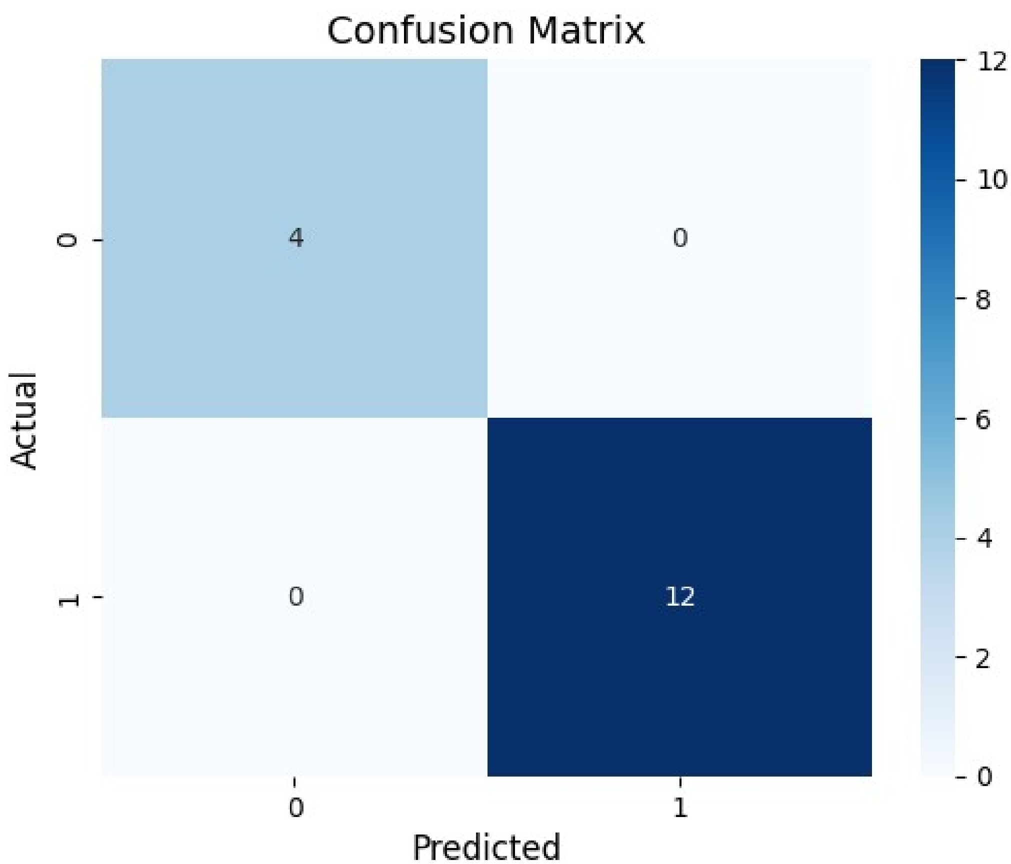
```
weighted avg       1.00      1.00      1.00        16
```




```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# Load the dataset
file_path = "/content/Records-Patient.csv"  # Update with your dataset
path
df = pd.read_csv(file_path)

# Features and target
X = df[['Age', 'BMI', 'Blood_Pressure', 'Cholesterol', 'Smoking',
'Family_History']].values
y = df['Risk_Level (Target)'].values

# Split the dataset into training and testing sets
```

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

# Feature Scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize and train the Logistic Regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
report = classification_report(y_test, y_pred)

# Print results
print(f"Accuracy: {accuracy:.4f}")
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", report)

# Visualization of the confusion matrix
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted", fontsize=12)
plt.ylabel("Actual", fontsize=12)
plt.title("Confusion Matrix", fontsize=14)
plt.show()
```

```
Accuracy: 1.0000
Confusion Matrix:
 [[ 4  0]
 [ 0 12]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         4
           1       1.00      1.00      1.00        12

    accuracy                           1.00        16
   macro avg       1.00      1.00      1.00        16
weighted avg       1.00      1.00      1.00        16
```

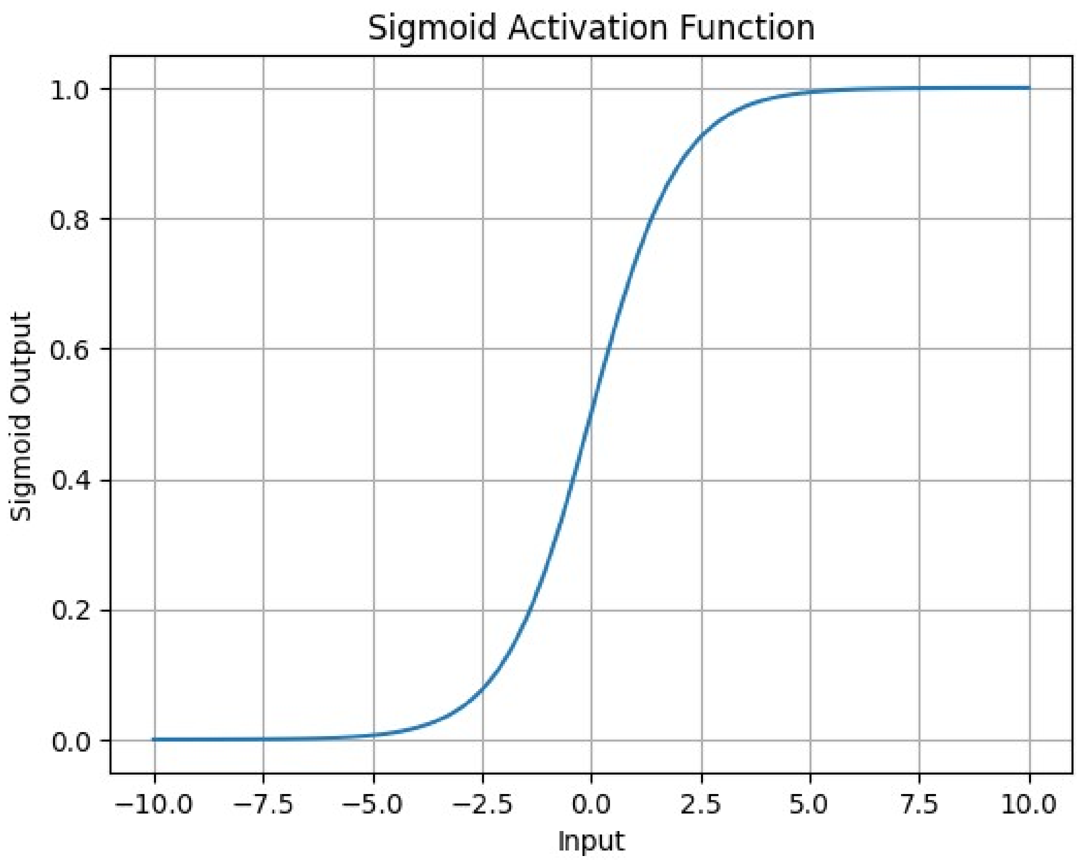


Activation Functions

Sigmoid Function

```
import math

def sigmoid(x):
  return 1 / (1 + math.exp(-x))

sigmoid(100)

1.0

sigmoid(-2)

0.11920292202211755

import numpy as np
import matplotlib.pyplot as plt

def plot_sigmoid():
```

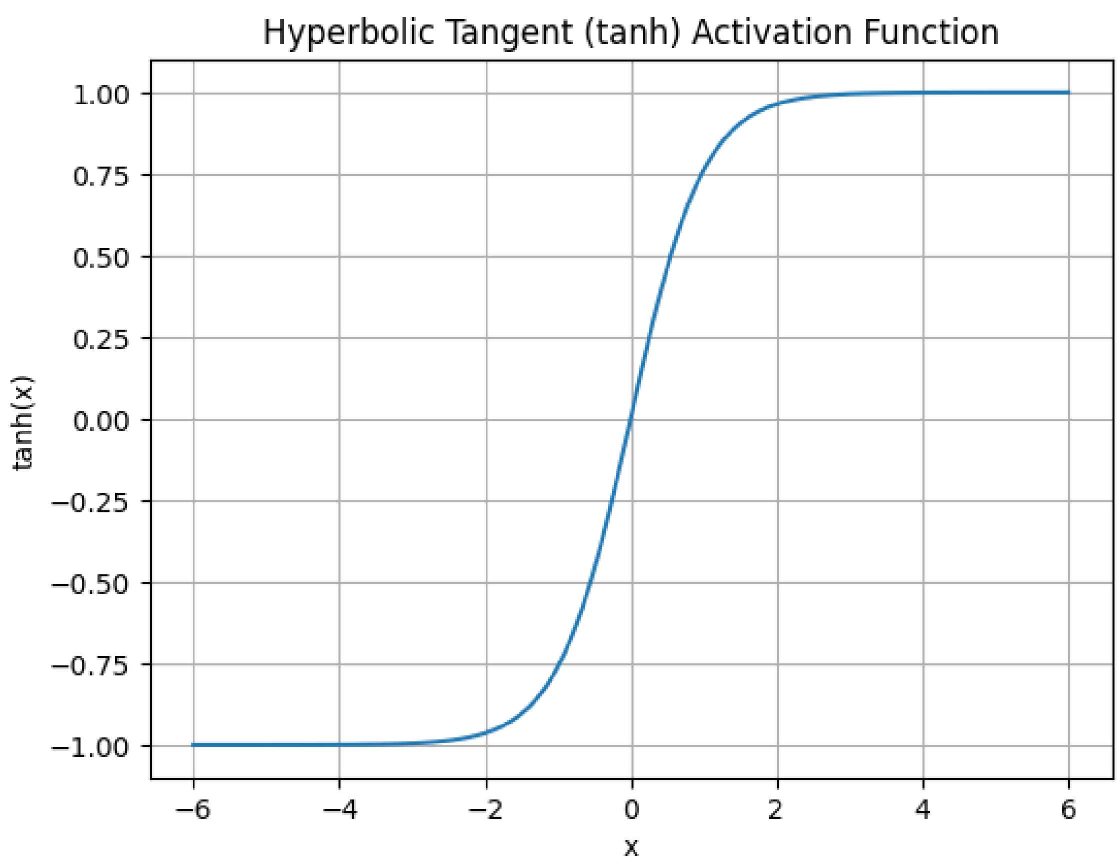
```
    x = np.linspace(-10, 10, 100)
    y = 1 / (1 + np.exp(-x))

    plt.plot(x, y)
    plt.xlabel('Input')
    plt.ylabel('Sigmoid Output')
    plt.title('Sigmoid Activation Function')
    plt.grid(True)
    plt.show()

plot_sigmoid()
```



TANH Function

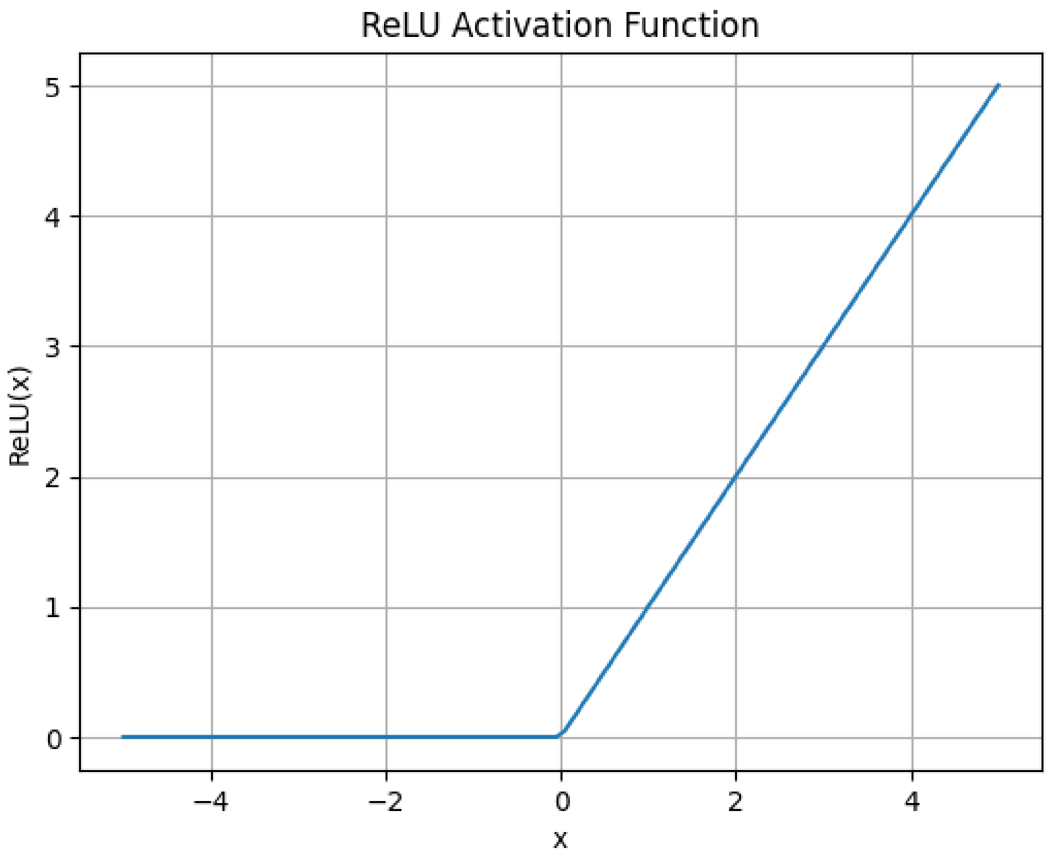
```
def tanh(x):
  return (math.exp(x)-math.exp(-x))/(math.exp(x)+math.exp(-x))

tanh(-2)

-0.964027580075817

tanh(12)
```

```
0.9999999999244972

import numpy as np
import matplotlib.pyplot as plt

def plot_tanh():

    x = np.linspace(-6, 6, 100)

    tanh = np.tanh(x)

    plt.plot(x, tanh)
    plt.title("Hyperbolic Tangent (tanh) Activation Function")
    plt.xlabel("x")
    plt.ylabel("tanh(x)")
    plt.grid(True)
    plt.show()


plot_tanh()
```



RELU

```
def relu(x):
  return max(0,x)

relu(-70)

0

relu(9)

9

import numpy as np
import matplotlib.pyplot as plt

def plot_relu():

    x = np.linspace(-5, 5, 100)

    relu = np.maximum(0, x)

    plt.plot(x, relu)
    plt.title("ReLU Activation Function")
    plt.xlabel("x")
    plt.ylabel("ReLU(x)")
    plt.grid(True)
    plt.show()

    plot_relu()
```



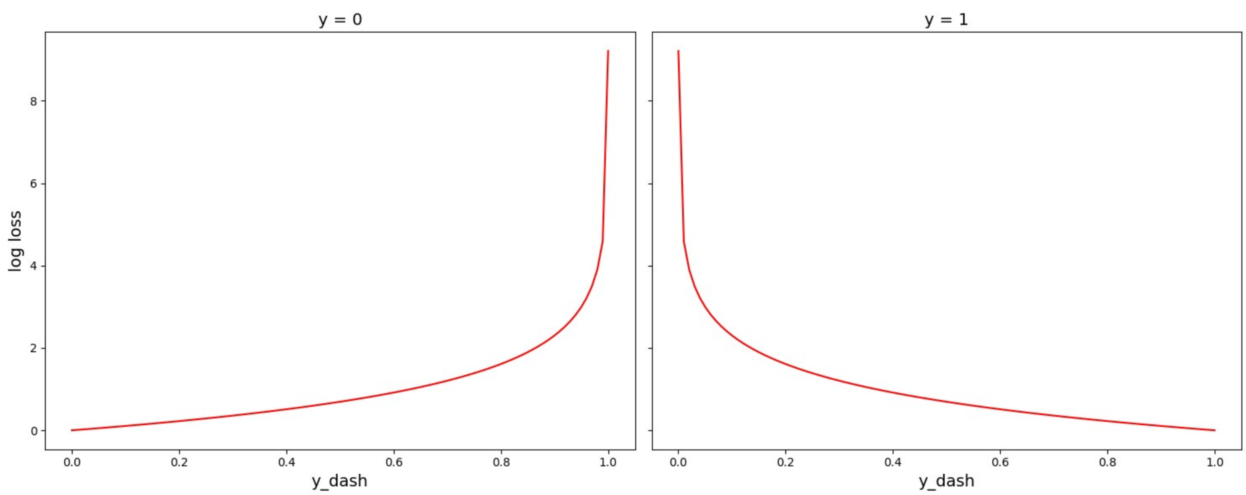
Log Loss Function

```
# Log loss
def log_loss(y, y_dash):
    """Computes log loss for inputs true value (0 or 1) and predicted
value (between 0 and 1)
    Args:
      y      (scalar): true value (0 or 1)
      y_dash (scalar): predicted value (probability of y being 1)
    Returns:
      loss (float): nonnegative loss corresponding to y and y_dash
    """
    loss = - (y * np.log(y_dash)) - ((1 - y) * np.log(1 - y_dash))
    return loss

y, y_dash = 1, 0.9
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")
y, y_dash = 0, 0.3
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")
y, y_dash = 1, 0.7
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")
```

```
y, y_dash = 0, 0.2
print(f"log_loss({y}, {y_dash}) = {log_loss(y, y_dash)}")
```

```
log_loss(1, 0.9) = 0.10536051565782628
log_loss(0, 0.3) = 0.35667494393873245
log_loss(1, 0.7) = 0.35667494393873245
log_loss(0, 0.2) = 0.2231435513142097
```

```
# Log loss for y = 0 and y = 1
fig, ax = plt.subplots(1, 2, figsize = (15, 6), sharex = True, sharey
= True)
y_dash = np.linspace(0.0001, 0.9999, 100)
ax[0].plot(y_dash, log_loss(0, y_dash), color = 'red')
ax[0].set_title("y = 0", fontsize = 14)
ax[0].set_xlabel("y_dash", fontsize = 14)
ax[0].set_ylabel("log loss", fontsize = 14)
ax[1].plot(y_dash, log_loss(1, y_dash), color = 'red')
ax[1].set_title("y = 1", fontsize = 14)
ax[1].set_xlabel("y_dash", fontsize = 14)
plt.tight_layout()
plt.show()
```



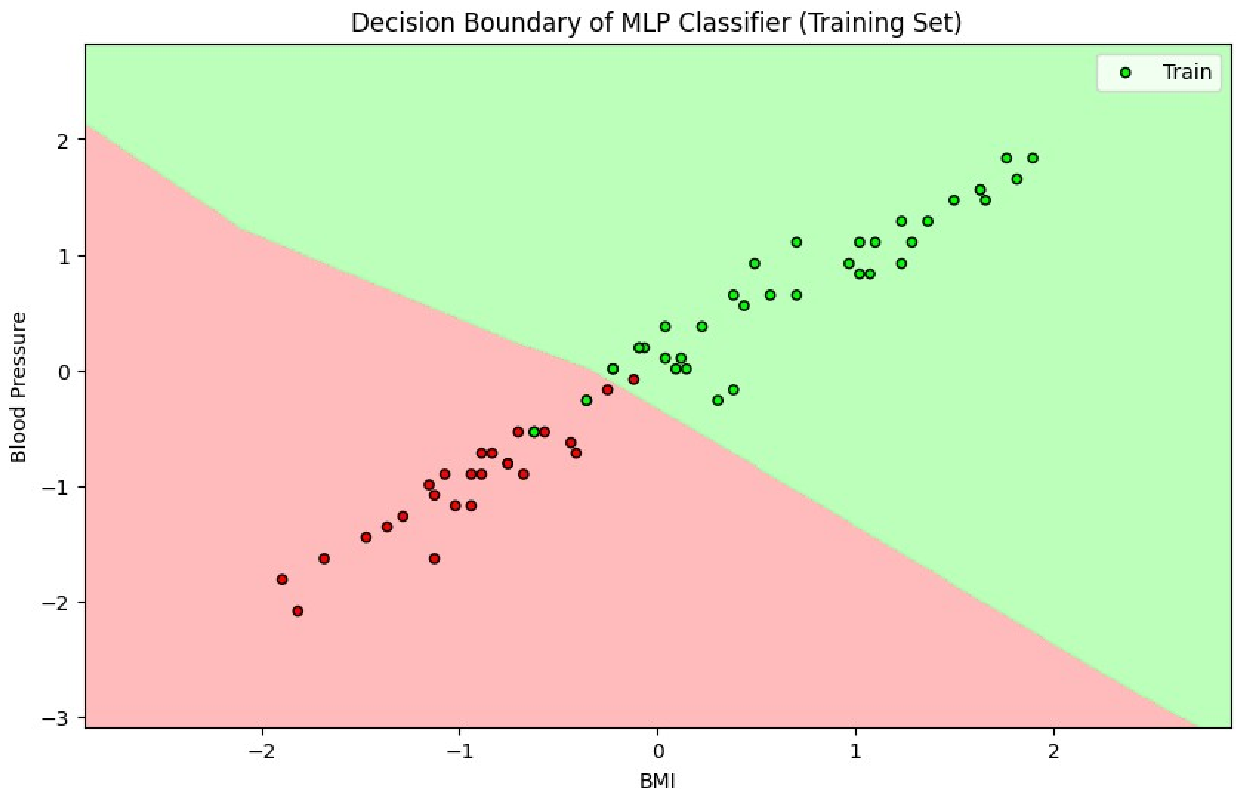
Sklearn Implementation of MultiLayer Perceptron(MLP)

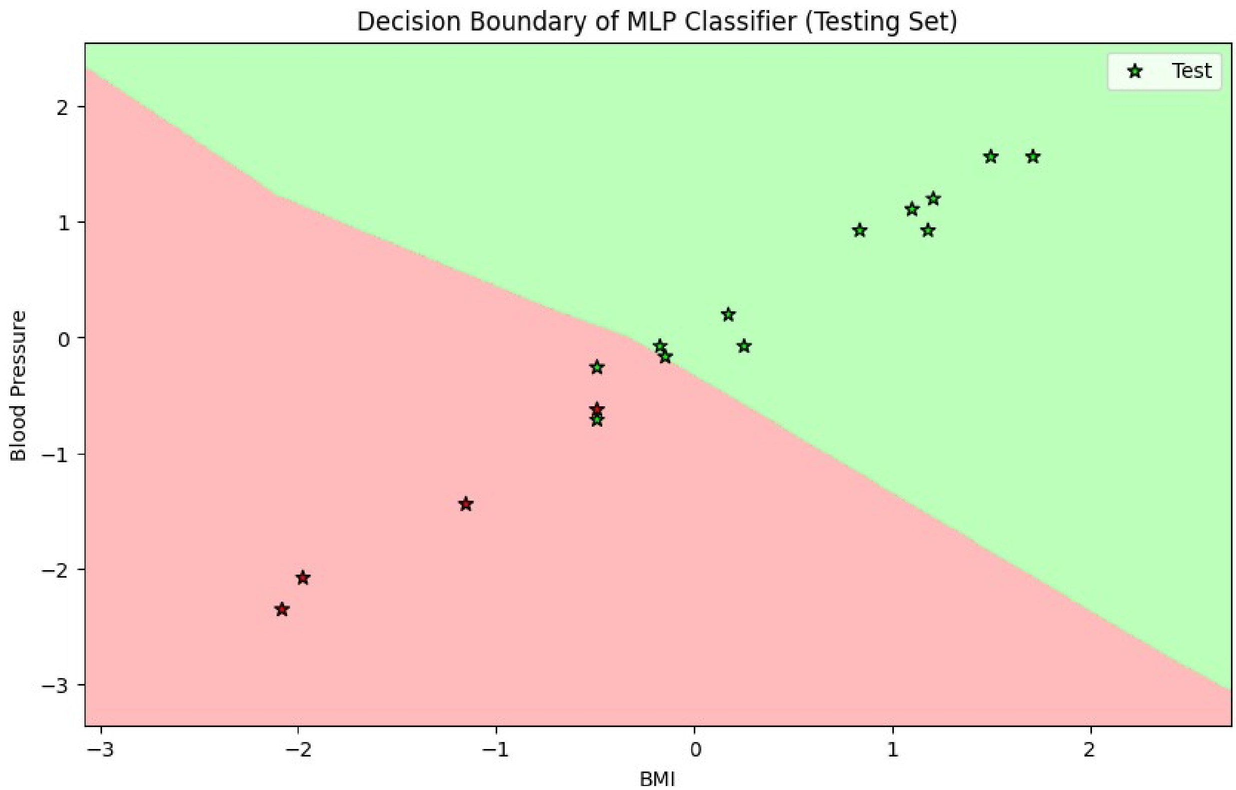
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, classification_report
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```python
from matplotlib.colors import ListedColormap

# Load the Patient Records dataset
data = pd.read_csv("/content/Records-Patient.csv")
X = data.iloc[:, 1:-1].values  # Use all features except Patient ID
and Label
y = data.iloc[:, -1].values  # Target labels

# Split the dataset 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
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Define the main MLP classifier (trained on all features)
mlp = MLPClassifier(hidden_layer_sizes=(10, 10), max_iter=1000,
random_state=42)
mlp.fit(X_train, y_train)

# Make predictions
y_pred = mlp.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print("\nClassification Report:\n", classification_report(y_test,
y_pred))

# ============================
# Train a separate classifier for visualization (only on BMI & Blood
Pressure)
# ============================
X_vis = data.iloc[:, [2, 3]].values  # Selecting BMI and Blood
Pressure
y_vis = data.iloc[:, -1].values  # Target labels

# Split and scale the visualization dataset
X_vis_train, X_vis_test, y_vis_train, y_vis_test =
train_test_split(X_vis, y_vis, test_size=0.2, random_state=42)
scaler_vis = StandardScaler()
X_vis_train = scaler_vis.fit_transform(X_vis_train)
X_vis_test = scaler_vis.transform(X_vis_test)

# Define a new MLP model for visualization
mlp_vis = MLPClassifier(hidden_layer_sizes=(10, 10), max_iter=1000,
random_state=42)
mlp_vis.fit(X_vis_train, y_vis_train)
```

```python
# ============================
# Plot decision boundary for training set
# ============================
x_min, x_max = X_vis_train[:, 0].min() - 1, X_vis_train[:, 0].max() +
1
y_min, y_max = X_vis_train[:, 1].min() - 1, X_vis_train[:, 1].max() +
1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))

Z_train = mlp_vis.predict(np.c_[xx.ravel(), yy.ravel()])
Z_train = Z_train.reshape(xx.shape)

# Define color maps
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA'])
cmap_bold = ListedColormap(['#FF0000', '#00FF00'])

plt.figure(figsize=(10, 6))
plt.contourf(xx, yy, Z_train, alpha=0.8, cmap=cmap_light)
plt.scatter(X_vis_train[:, 0], X_vis_train[:, 1], c=y_vis_train,
cmap=cmap_bold, edgecolor='k', s=20, label='Train')
plt.title("Decision Boundary of MLP Classifier (Training Set)")
plt.xlabel("BMI")
plt.ylabel("Blood Pressure")
plt.legend()
plt.show()

# ============================
# Plot decision boundary for testing set
# ============================
x_min, x_max = X_vis_test[:, 0].min() - 1, X_vis_test[:, 0].max() + 1
y_min, y_max = X_vis_test[:, 1].min() - 1, X_vis_test[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01),
                     np.arange(y_min, y_max, 0.01))

Z_test = mlp_vis.predict(np.c_[xx.ravel(), yy.ravel()])
Z_test = Z_test.reshape(xx.shape)

plt.figure(figsize=(10, 6))
plt.contourf(xx, yy, Z_test, alpha=0.8, cmap=cmap_light)
plt.scatter(X_vis_test[:, 0], X_vis_test[:, 1], c=y_vis_test,
cmap=cmap_bold, edgecolor='k', s=50, label='Test', marker='*')
plt.title("Decision Boundary of MLP Classifier (Testing Set)")
plt.xlabel("BMI")
plt.ylabel("Blood Pressure")
plt.legend()
plt.show()
```

```
Accuracy: 1.0

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         4
           1       1.00      1.00      1.00        12

    accuracy                           1.00        16
   macro avg       1.00      1.00      1.00        16
weighted avg       1.00      1.00      1.00        16
```





Keras Implementation of MultiLayer Perceptron(MLP)

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report,
ConfusionMatrixDisplay

# Step 1: Prepare a synthetic dataset
X, y = make_classification(
    n_samples=1000, n_features=20, n_informative=15, n_redundant=5,
random_state=42
)

# Split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

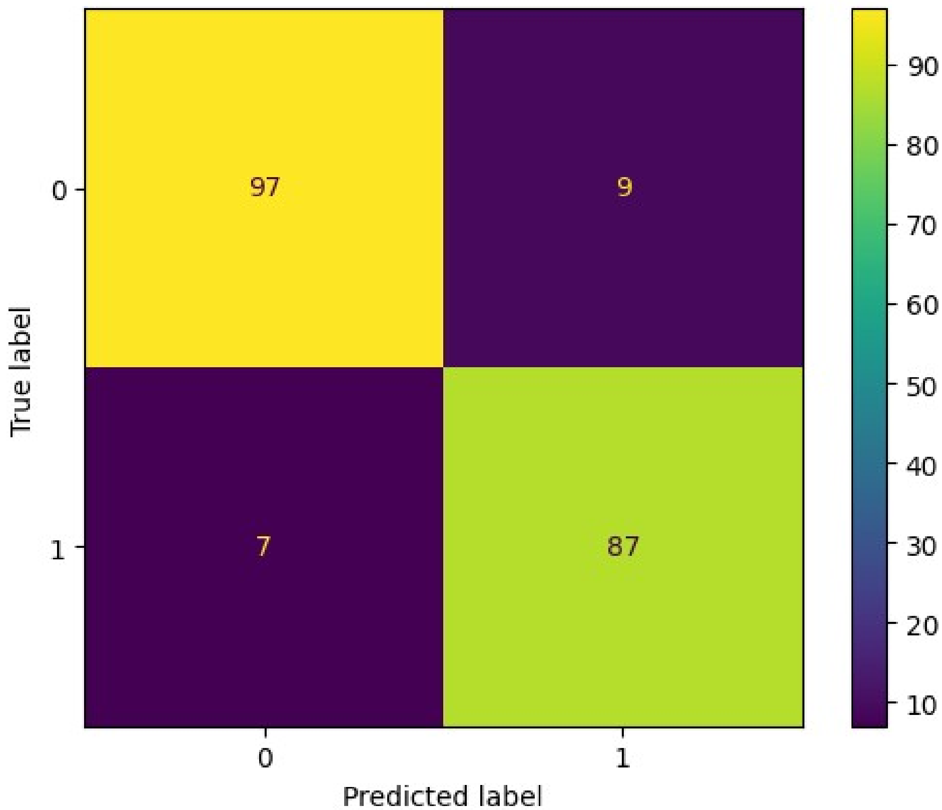
```python
# Standardize the data (helps with convergence and performance)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)  # Fit the scaler on training data and transform it
X_test = scaler.transform(X_test)  # Transform the testing data using the same scaler

# Step 2: Build the ANN model
model = Sequential([
    Dense(32, activation='relu', input_dim=X_train.shape[1]),  # Hidden layer with 32 neurons and ReLU activation
    Dense(16, activation='relu'),  # Another hidden layer with 16 neurons and ReLU activation
    Dense(1, activation='sigmoid')  # Output layer with 1 neuron and sigmoid activation for binary classification
])

# Step 3: Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Step 4: Train the model
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2, verbose=1)

# Step 5: Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"\nTest Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")

# Step 6: Generate predictions
y_pred = (model.predict(X_test) > 0.5).astype(int)

# Step 7: Classification Report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Step 8: Visualize confusion matrix
ConfusionMatrixDisplay.from_predictions(y_test, y_pred)
plt.show()
```

```
Epoch 1/20
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 16ms/step - accuracy: 0.5341 - loss:
1.0082 - val_accuracy: 0.4938 - val_loss: 0.8741
Epoch 2/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.5357 - loss:
0.7633 - val_accuracy: 0.5375 - val_loss: 0.7021
Epoch 3/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6023 - loss:
0.6426 - val_accuracy: 0.6938 - val_loss: 0.6158
Epoch 4/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.7382 - loss:
0.5541 - val_accuracy: 0.7188 - val_loss: 0.5557
Epoch 5/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7891 - loss:
0.5064 - val_accuracy: 0.7625 - val_loss: 0.5119
Epoch 6/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8585 - loss:
0.4480 - val_accuracy: 0.8125 - val_loss: 0.4713
Epoch 7/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8883 - loss:
0.3956 - val_accuracy: 0.8313 - val_loss: 0.4364
Epoch 8/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.8906 - loss:
0.3623 - val_accuracy: 0.8250 - val_loss: 0.4074
Epoch 9/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9176 - loss:
0.3330 - val_accuracy: 0.8500 - val_loss: 0.3845
Epoch 10/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9126 - loss:
0.3112 - val_accuracy: 0.8500 - val_loss: 0.3647
Epoch 11/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9110 - loss:
0.2744 - val_accuracy: 0.8562 - val_loss: 0.3479
Epoch 12/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9133 - loss:
0.2567 - val_accuracy: 0.8500 - val_loss: 0.3302
Epoch 13/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9112 - loss:
0.2367 - val_accuracy: 0.8500 - val_loss: 0.3161
Epoch 14/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9092 - loss:
0.2279 - val_accuracy: 0.8562 - val_loss: 0.3067
Epoch 15/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9466 - loss:
0.1986 - val_accuracy: 0.8562 - val_loss: 0.2936
Epoch 16/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9436 - loss:
0.1851 - val_accuracy: 0.8625 - val_loss: 0.2910
Epoch 17/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9365 - loss:
0.1860 - val_accuracy: 0.8750 - val_loss: 0.2795
```

```
Epoch 18/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9467 - loss:
0.1775 - val_accuracy: 0.8687 - val_loss: 0.2751
Epoch 19/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9427 - loss:
0.1688 - val_accuracy: 0.8813 - val_loss: 0.2625
Epoch 20/20
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9309 - loss:
0.1689 - val_accuracy: 0.8813 - val_loss: 0.2585
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9064 - loss: 0.2178


Test Loss: 0.1964
Test Accuracy: 0.9200
7/7 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step

Classification Report:
              precision    recall  f1-score   support

           0       0.93      0.92      0.92       106
           1       0.91      0.93      0.92        94

    accuracy                           0.92       200
   macro avg       0.92      0.92      0.92       200
weighted avg       0.92      0.92      0.92       200
```

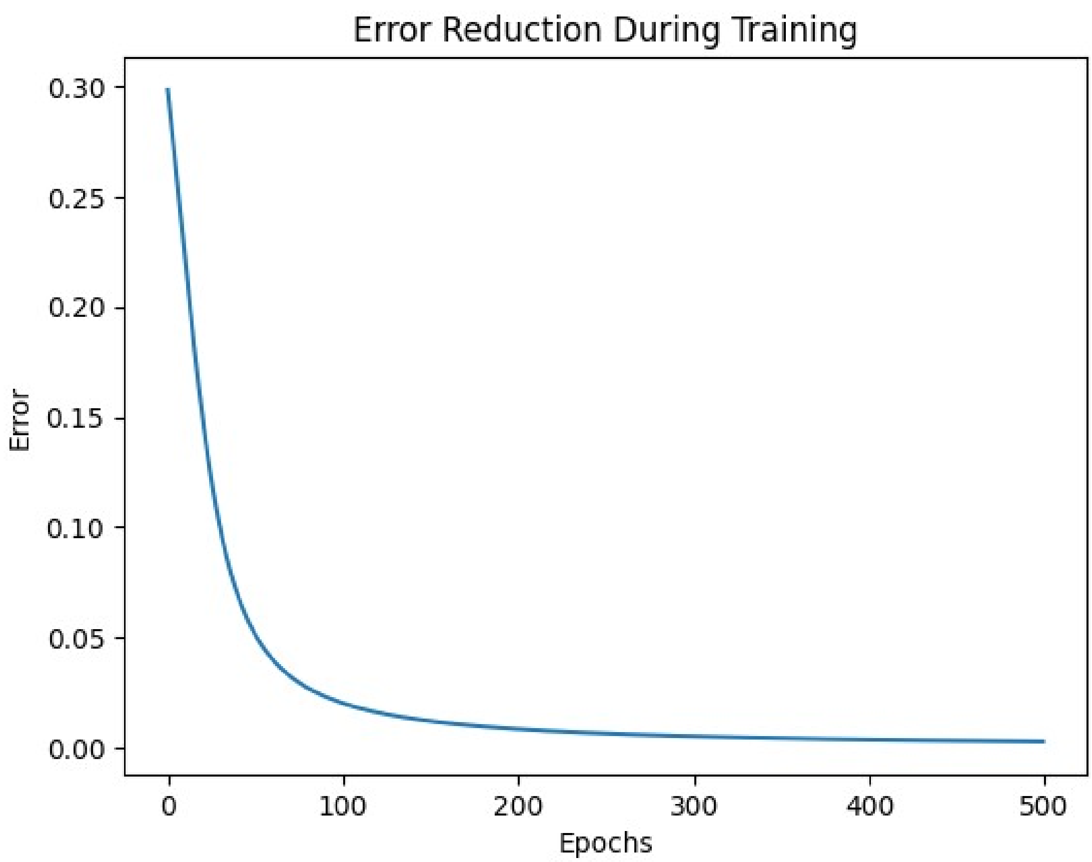


Backward Propogation from Sratch

```
import math
import numpy as np
import matplotlib.pyplot as plt

# Sigmoid Activation Function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of Sigmoid Function
def sigmoid_derivative(x):
    return x * (1 - x)

# Feedforward function
def feed_forward(b1, b2, w1, w2, x):
    hidden = []
    output = []

    # Hidden Layer Activation
    hidden.append(sigmoid(b1 + np.dot(w1[:2], x)))  # First neuron
    hidden.append(sigmoid(b1 + np.dot(w1[2:], x)))  # Second neuron
```

```
    # Output Layer Activation
    output.append(sigmoid(b2 + np.dot(w2[:2], hidden)))  # First
output neuron
    output.append(sigmoid(b2 + np.dot(w2[2:], hidden)))  # Second
output neuron

    return hidden, output

# Error Calculation
def find_error(output, desired):
    return sum((np.array(output) - np.array(desired))**2) / 2

# Backpropagation
def back_propagate(w1, w2, hidden, output, desired, x, alpha):
    # Compute error terms for output layer
    delta_output = [(output[i] - desired[i]) *
sigmoid_derivative(output[i]) for i in range(len(output))]

    # Compute error terms for hidden layer
    delta_hidden = []
    for i in range(len(hidden)):
        temp = sum(delta_output[j] * w2[i + j * len(hidden)] for j in
range(len(output)))
        delta_hidden.append(temp * sigmoid_derivative(hidden[i]))

    # Update weights for hidden-to-output layer
    for i in range(len(w2)):
        w2[i] -= alpha * delta_output[i // len(hidden)] * hidden[i %
len(hidden)]

    # Update weights for input-to-hidden layer
    for i in range(len(w1)):
        w1[i] -= alpha * delta_hidden[i // len(x)] * x[i % len(x)]

# Initialization
w1 = [0.15, 0.20, 0.25, 0.30]  # Weights for input to hidden layer
w2 = [0.40, 0.45, 0.50, 0.55]  # Weights for hidden to output layer
x = [0.05, 0.10]  # Input values
b1 = 0.35  # Bias for hidden layer
b2 = 0.60  # Bias for output layer
desired = [0.01, 0.99]  # Desired output
epochs = 500  # Training iterations
alpha = 0.5  # Learning rate
error = []

# Training Loop
for _ in range(epochs):
    hidden, output = feed_forward(b1, b2, w1, w2, x)
    error.append(find_error(output, desired))
    back_propagate(w1, w2, hidden, output, desired, x, alpha)
```

```
# Plot Error Reduction Over Time
plt.plot(error)
plt.xlabel("Epochs")
plt.ylabel("Error")
plt.title("Error Reduction During Training")
plt.show()

# Final Output after Training
print("Final Output:", output)
```




```
Final Output: [0.06389456919363433, 0.9404153154587165]
```

Link of ipynb file:
https://colab.research.google.com/drive/1N6h82U250j5U4lCFPQ0aLbzefTKoO-JM?usp=sharing