Lab Assignment 3

Date:24/01/25

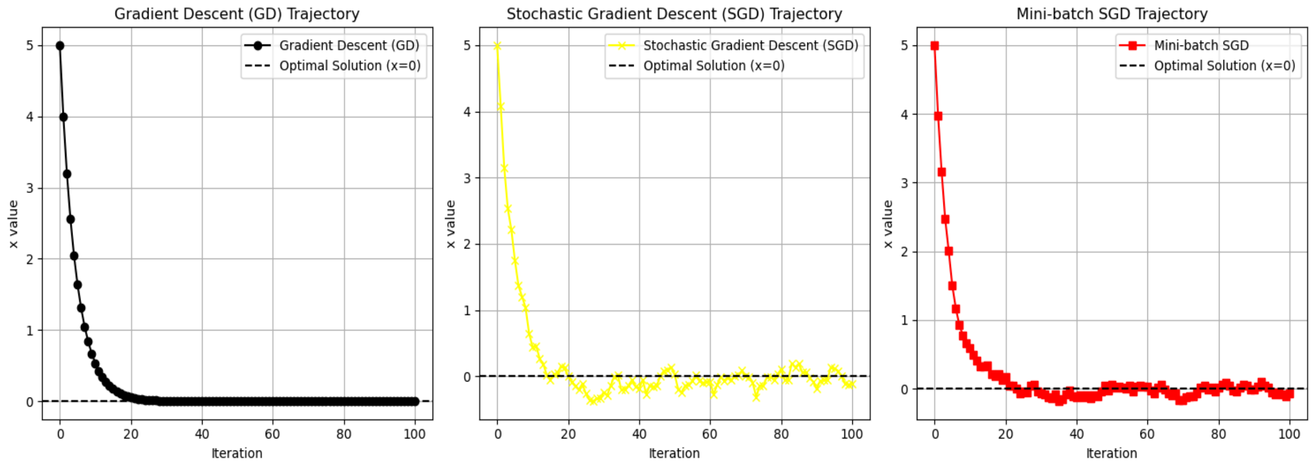
Q.1 - Demonstrate the behavior of Gradient Descent and its variants—Batch Gradient Descent (GD), Stochastic Gradient Descent (SGD), and Mini-batch Stochastic Gradient Descent (Mini-batch SGD) on both simple and complex functions, along with plots showing their computational cost and trajectory convergence behavior. Return the optimal final values of the following function f(x) = x^2 and G(x) = x^4+x^3+x^2 for a range of x between -20 to 20, learning rate =.01 and epoch size =50 and plot the trajectory of convergence.

<u>CODE</u>

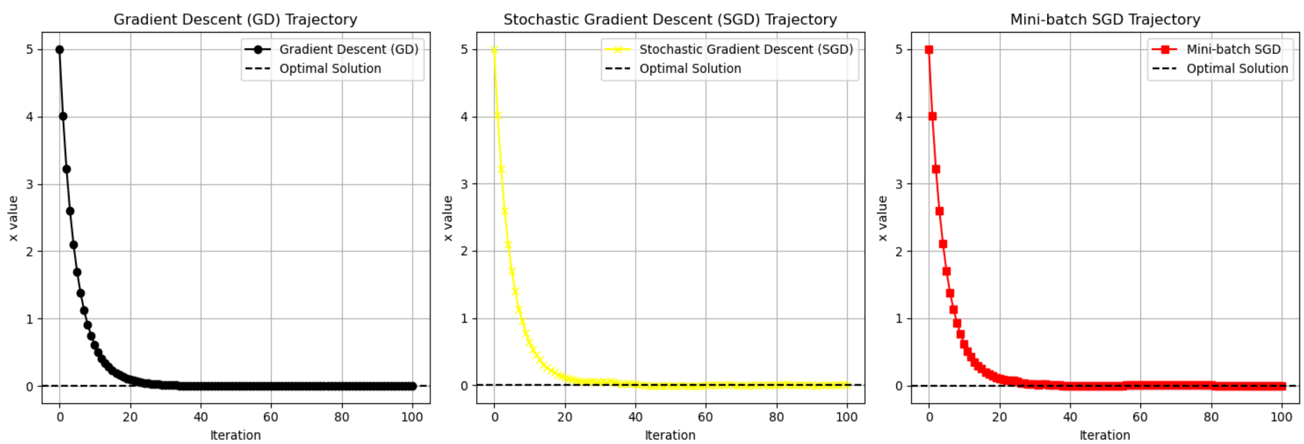
```
import numpy as np
import matplotlib.pyplot as plt

# Function 1: Quadratic Function
def f1(x):
    return x**2
def grad_f1(x):
    return 2 * x

# Function 2: Cubic Polynomial Function
def f2(x):
    return x**4 + x**3 + x**2
def grad_f2(x):
    return 4 * x**3 + 3 * x**2 + 2 * x

# Function 3: Cosine Function
def f3(x):
    return np.cos(np.pi * x)
def grad_f3(x):
    return -np.pi * np.sin(np.pi * x)
# Parameters
x0 = 5
eta = 0.1
num_iterations = 100

# Gradient Descent (GD)
x_gd = x0
gd_trajectory = [x_gd]
for _ in range(num_iterations):
    x_gd = x_gd - eta * grad_f(x_gd)
    gd_trajectory.append(x_gd)

# Stochastic Gradient Descent (SGD)
x_sgd = x0
sgd_trajectory = [x_sgd]
for _ in range(num_iterations):
    # Adding some noise to simulate SGD
    noisy_grad = grad_f(x_sgd) + np.random.normal(0, 1)
```

```
    x_sgd = x_sgd - eta * noisy_grad
    sgd_trajectory.append(x_sgd)

# Mini-batch Stochastic Gradient Descent (Mini-batch SGD)
x_mbsgd = x0
mbsgd_trajectory = [x_mbsgd]
batch_size = 5
for _ in range(num_iterations):
    # Adding less noise to simulate mini-batch SGD
    noisy_grad = grad_f(x_mbsgd) + np.random.normal(0, 0.5)
    x_mbsgd = x_mbsgd - eta * noisy_grad
    mbsgd_trajectory.append(x_mbsgd)

# Print final values of x
print(f"Final value of x (GD): {x_gd:.4f}")
print(f"Final value of x (SGD): {x_sgd:.4f}")
print(f"Final value of x (Mini-batch SGD): {x_mbsgd:.4f}")

# Plotting the trajectories in separate graphs
plt.figure(figsize=(15, 5))

# Gradient Descent (GD)
plt.subplot(1, 3, 1)
plt.plot(gd_trajectory, label='Gradient Descent (GD)', marker='o', color='black')
plt.axhline(0, color='black', linestyle='--', label='Optimal Solution (x=0)')
plt.xlabel('Iteration')
plt.ylabel('x value')
plt.title('Gradient Descent (GD) Trajectory')
plt.legend()
plt.grid(True)

# Stochastic Gradient Descent (SGD)
plt.subplot(1, 3, 2)
plt.plot(sgd_trajectory, label='Stochastic Gradient Descent (SGD)', marker='x', color='yellow')
plt.axhline(0, color='black', linestyle='--', label='Optimal Solution (x=0)')
plt.xlabel('Iteration')
plt.ylabel('x value')
plt.title('Stochastic Gradient Descent (SGD) Trajectory')
plt.legend()
plt.grid(True)

# Mini-batch Stochastic Gradient Descent (Mini-batch SGD)
plt.subplot(1, 3, 3)
plt.plot(mbsgd_trajectory, label='Mini-batch SGD', marker='s', color='red')
plt.axhline(0, color='black', linestyle='--', label='Optimal Solution (x=0)')
plt.xlabel('Iteration')
plt.ylabel('x value')
plt.title('Mini-batch SGD Trajectory')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

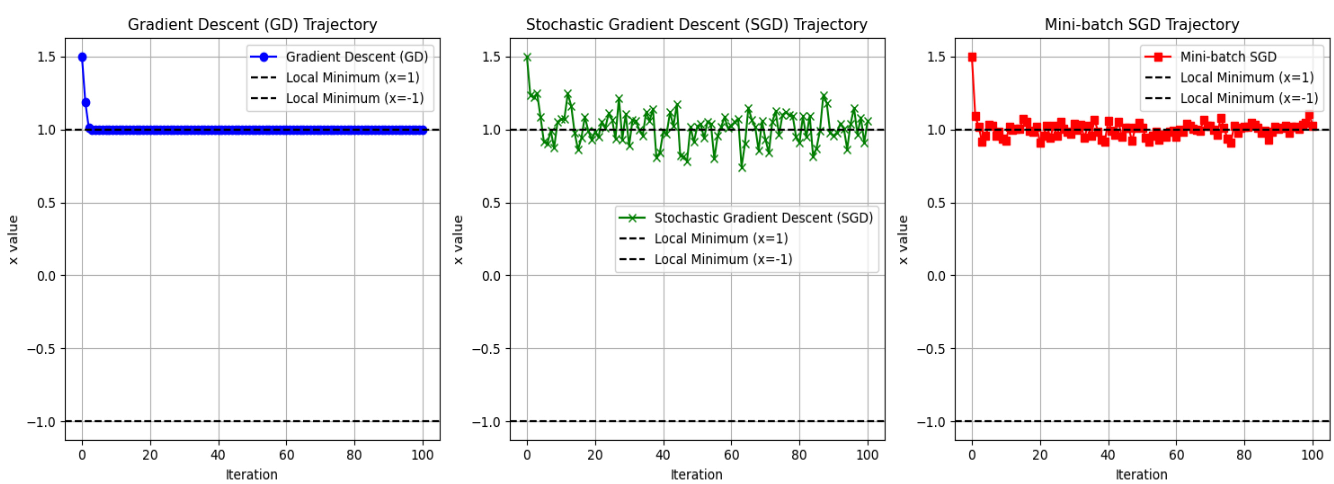
Function 1: Quadratic Function



 # Function 2: Cubic Polynomial Function



 # Function 3: Cosine Function



Q.2 Diabetes Prediction with Pima Indians Diabetes Dataset : Develop and train a deep neural network to predict the onset of diabetes using the Pima Indians Diabetes dataset. The objective is to perform binary classification to determine whether a patient has diabetes based on diagnostic measurements. This task involves: 1. Data Exploration and Preprocessing: Understanding and preparing medical data for modeling. 2. Model Architecture & Implementation: Building and training a deep neural network for classification. 3. Interpretation: Evaluating model performance and discussing improvements and real-world implications.

1. Data Exploration and Preprocessing: Understanding and preparing medical data for modeling.

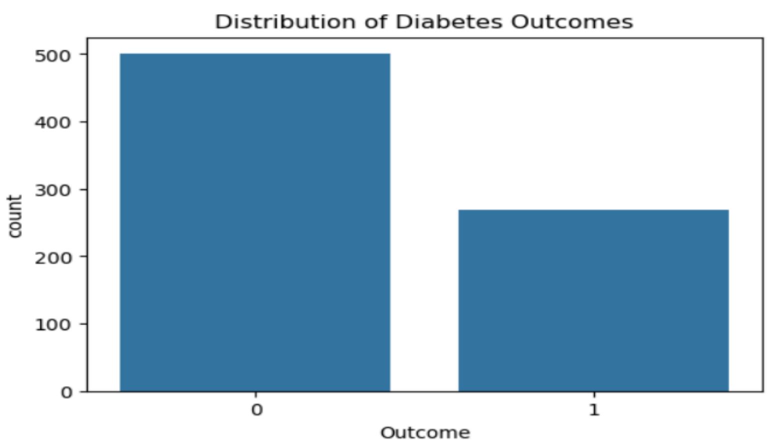
```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import SMOTE

# Load the dataset
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"
columns = [
    "Pregnancies", "Glucose", "BloodPressure", "SkinThickness", "Insulin",
    "BMI", "DiabetesPedigreeFunction", "Age", "Outcome"
]
data = pd.read_csv(url, names=columns)

# Display the first few rows
print(data.head())
```

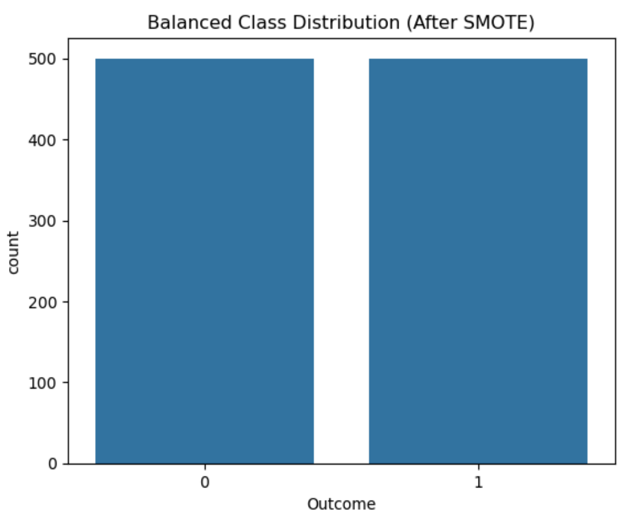
OUTPUT

```
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Pregnancies               768 non-null    int64
 1   Glucose                   768 non-null    int64
 2   BloodPressure             768 non-null    int64
 3   SkinThickness             768 non-null    int64
 4   Insulin                   768 non-null    int64
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64
 8   Outcome                   768 non-null    int64
dtypes: float64(2), int64(7)
```

```
# Replace 0 values with NaN in relevant columns
missing_columns = ["Glucose", "BloodPressure", "SkinThickness", "Insulin", "BMI"]
data[missing_columns] = data[missing_columns].replace(0, np.nan)

# Impute missing values using median (robust to outliers)
data.fillna(data.median(), inplace=True)

# Check class distribution
sns.countplot(x="Outcome", data=data)
```

```
plt.title("Class Distribution")
plt.show()

# Split features and target
X = data.drop("Outcome", axis=1)
y = data["Outcome"]

# Apply SMOTE to balance the dataset
smote = SMOTE(random_state=42)
X_balanced, y_balanced = smote.fit_resample(X, y)

# Verify balanced class distribution
sns.countplot(x=y_balanced)
plt.title("Balanced Class Distribution (After SMOTE)")
plt.show()
```






```
# Scale features using StandardScaler
scaler = StandardScaler()
X_balanced_scaled = scaler.fit_transform(X_balanced)

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

X_train shape: (800, 8), X_test shape: (200, 8)
y_train shape: (800,), y_test shape: (200,)

2. Model Architecture & Implementation: Building and training a deep neural network for classification.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping

# Define the model architecture
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.3),  # Prevent overfitting
    Dense(32, activation='relu'),
    Dropout(0.3),
    Dense(1, activation='sigmoid')  # Binary classification output
])
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Train the model
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
history = model.fit(
    X_train, y_train,
    validation_data=(X_test, y_test),
    epochs=50,
    batch_size=32,
    callbacks=[early_stopping],
    verbose=1
)
```

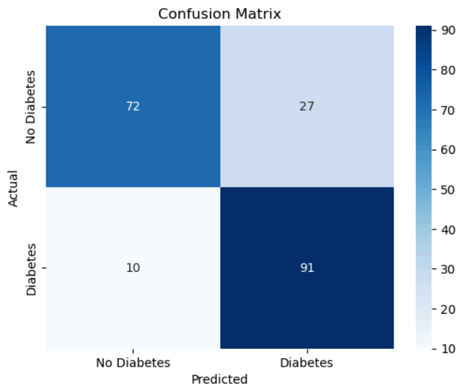
Epoch 1/50
25/25 ━━━━━━━━━━━━━━━━━━━━ **5s** 39ms/step - accuracy: 0.5143 - loss: 0.6976 - val_accuracy: 0.7150 - val_loss: 0.5842
Epoch 37/50
25/25 ━━━━━━━━━━━━━━━━━━━━ **0s** 13ms/step - accuracy: 0.7900 - loss: 0.4353 - val_accuracy: 0.8100 - val_loss: 0.4458

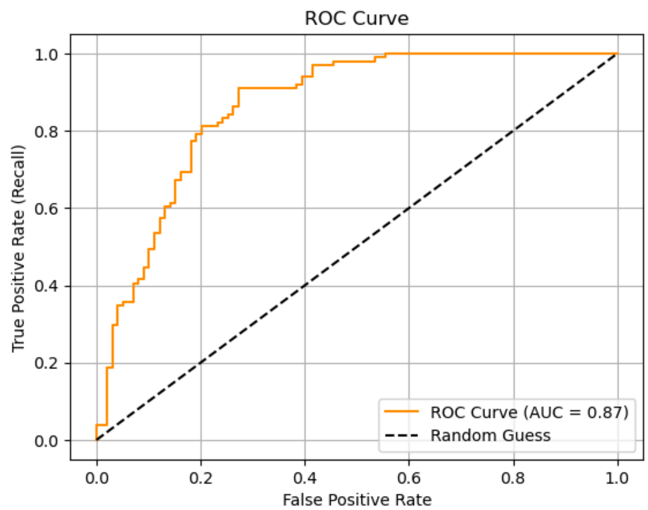
OUTPUT

```
# Generate predictions
y_pred_prob = model.predict(X_test).flatten()  # Probabilities
y_pred = (y_pred_prob > 0.5).astype(int)  # Threshold at 0.5 for binary classification
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)
roc_auc = roc_auc_score(y_test, y_pred_prob)
```

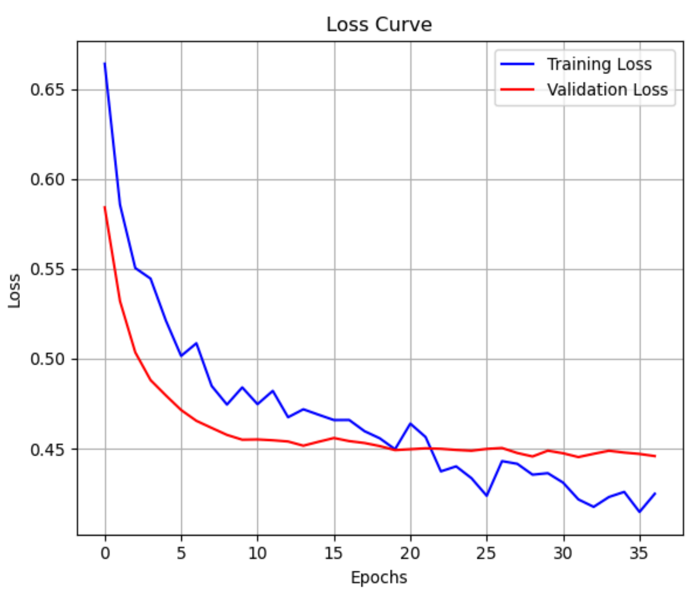
Model Performance Summary:
- Accuracy: 0.81
- Precision: 0.77
- Recall: 0.90
- F1-score: 0.83
- ROC-AUC: 0.87

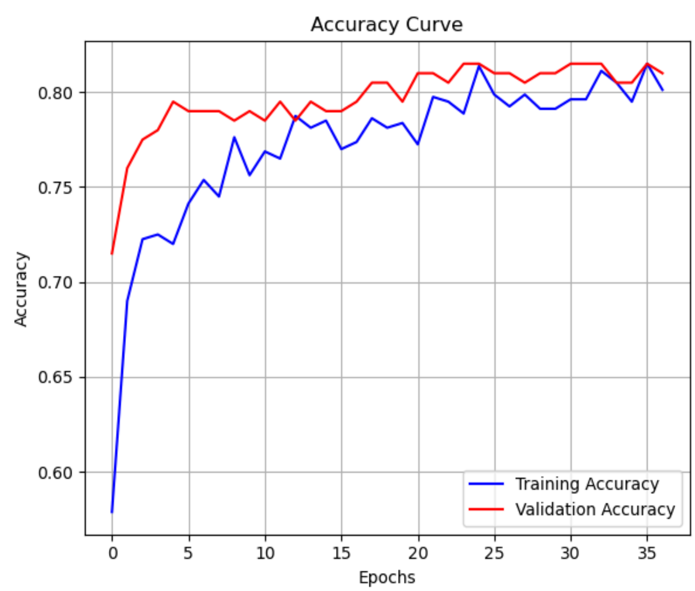
Handling Class Imbalance Impact:
1. SMOTE was used to address class imbalance by oversampling the minority class.
2. This improved the model's ability to predict both classes (diabetic and non-diabetic).
3. The F1-score and ROC-AUC score reflect balanced performance across classes.





Plot training and validation loss





Thank you sir