UNIVERSITY OF KALYANI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

3rd semester

Name:	Suhin Dubey
Registration number:	2080039
Registration year :	2023-2024
University Roll Number:	90/MCA NO. 230023

Paper Name :	Advance Programming Laboratory(313)

INDEX

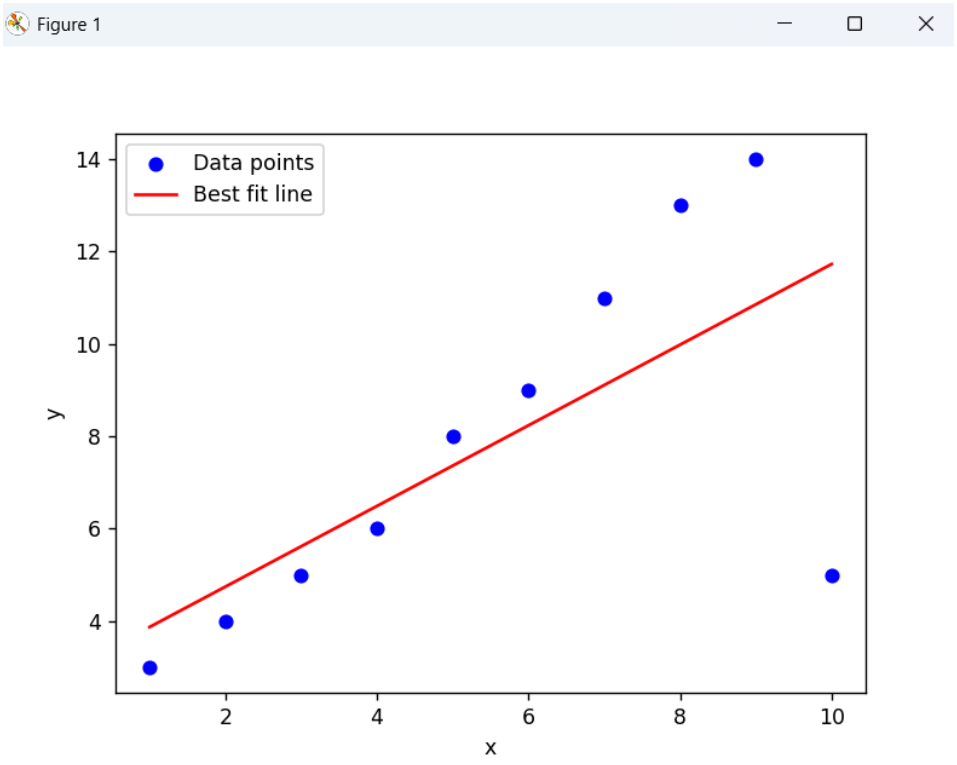
Q1. Implement a Python program to perform linear regression on a given dataset. Plot the data points and the best-fit line.

Code:

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

def linear_regression(x, y):
    # Convert input to NumPy arrays
    x = np.array(x)
    y = np.array(y)

    # Calculate means
    x_mean = np.mean(x)
    y_mean = np.mean(y)

    # Calculate slope and intercept
    numerator = np.sum((x - x_mean) * (y - y_mean))
    denominator = np.sum((x - x_mean) ** 2)

    slope = numerator / denominator
    intercept = y_mean - slope * x_mean

    return slope, intercept

# Input data
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [3, 4, 5, 6, 8, 9, 11, 13, 14, 5]

# Perform linear regression
slope, intercept = linear_regression(x, y)

# Predict y values
x = np.array(x)  # Convert x to a NumPy array for calculations
y_pred = intercept + slope * x

# Calculate the sum of squared error (SSE)
sse = np.sum((y - y_pred) ** 2)

# Print results
print("Intercept:", intercept)
```

```
print("Slope:", slope)
print(f"Best-fit line equation: y = {slope:.2f}x + {intercept:.2f}")
print(f"Sum of Squared Error (SSE): {sse:.2f}")

# Plot results
plt.scatter(x, y, color='blue', label='Data points')  # Original data points
plt.plot(x, y_pred, color='red', label='Best fit line')  # Best-fit line
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

Output:



Intercept: 3.0

Slope: 0.8727272727272727

Best-fit line equation: y = 0.87x + 3.00

Sum of Squared Error (SSE): 70.76

Q2. Build a logistic regression model to classify a binary dataset. Display the decision boundary and evaluate the model's performance using metrics like accuracy, precision, and recall.

Code:

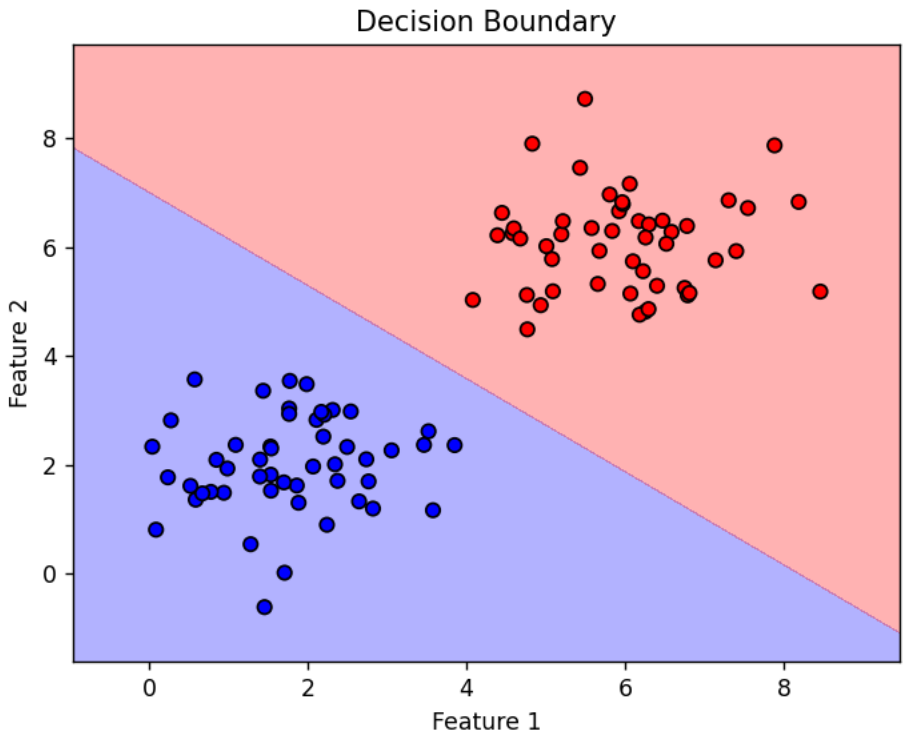
```
import numpy as np
import matplotlib.pyplot as plt

# Generate a simple binary dataset
np.random.seed(42)
num_samples = 100
x1_class_0 = np.random.normal(2, 1, num_samples // 2)
x2_class_0 = np.random.normal(2, 1, num_samples // 2)
x1_class_1 = np.random.normal(6, 1, num_samples // 2)
x2_class_1 = np.random.normal(6, 1, num_samples // 2)

x = np.vstack((np.c_[x1_class_0, x2_class_0], np.c_[x1_class_1, x2_class_1]))
y = np.array([0] * (num_samples // 2) + [1] * (num_samples // 2))  # Labels
x = np.c_[np.ones(x.shape[0]), x]  # Add bias term

# Sigmoid function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Logistic regression training
def train_logistic_regression(x, y, lr=0.01, epochs=1000):
    weights = np.zeros(x.shape[1])  # Initialize weights
    for _ in range(epochs):
        z = np.dot(x, weights)
        predictions = sigmoid(z)
        gradient = np.dot(x.T, (predictions - y)) / len(y)
        weights -= lr * gradient
    return weights

# Prediction function
def predict(x, weights):
    return (sigmoid(np.dot(x, weights)) >= 0.5).astype(int)
```

```
# Metrics
def compute_confusion_matrix(y_true, y_pred):
    tn = np.sum((y_true == 0) & (y_pred == 0))
    tp = np.sum((y_true == 1) & (y_pred == 1))
    fn = np.sum((y_true == 1) & (y_pred == 0))
    fp = np.sum((y_true == 0) & (y_pred == 1))
    return np.array([[tn, fp], [fn, tp]])

# Train the model
weights = train_logistic_regression(x, y, lr=0.1, epochs=3000)

# Make predictions
y_pred = predict(x, weights)

# Evaluate the model
conf_matrix = compute_confusion_matrix(y, y_pred)
tn, fp, fn, tp = conf_matrix.ravel()
accuracy = (tp + tn) / len(y)
precision = tp / (tp + fp) if tp + fp > 0 else 0
recall = tp / (tp + fn) if tp + fn > 0 else 0
f1_score = 2 * (precision * recall) / (precision + recall) if (precision +
recall) > 0 else 0

# Print results
print("Confusion Matrix:")
print(f"       Predicted 0   Predicted 1")
print(f"Actual 0      {conf_matrix[0, 0]}             {conf_matrix[0, 1]}")
print(f"Actual 1      {conf_matrix[1, 0]}             {conf_matrix[1, 1]}")
print(f"\nAccuracy: {accuracy:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
print(f"F1 Score: {f1_score:.2f}")

# Plot the decision boundary
x1_min, x1_max = x[:, 1].min() - 1, x[:, 1].max() + 1
x2_min, x2_max = x[:, 2].min() - 1, x[:, 2].max() + 1
xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max, 100), np.linspace(x2_min,
x2_max, 100))
grid = np.c_[np.ones(xx1.ravel().shape), xx1.ravel(), xx2.ravel()]
probs = sigmoid(np.dot(grid, weights)).reshape(xx1.shape)

plt.contourf(xx1, xx2, probs, levels=[0, 0.5, 1], alpha=0.3, colors=['blue',
'red'])
plt.scatter(x[:, 1], x[:, 2], c=y, cmap='bwr', edgecolor='k')
plt.xlabel('Feature 1')
```

```
plt.ylabel('Feature 2')
plt.title('Decision Boundary')
plt.show()
```

Output:

```
Confusion Matrix:
          Predicted 0   Predicted 1
Actual 0        50             0
Actual 1         0            50

Accuracy: 1.00
Precision: 1.00
Recall: 1.00
F1 Score: 1.00
```



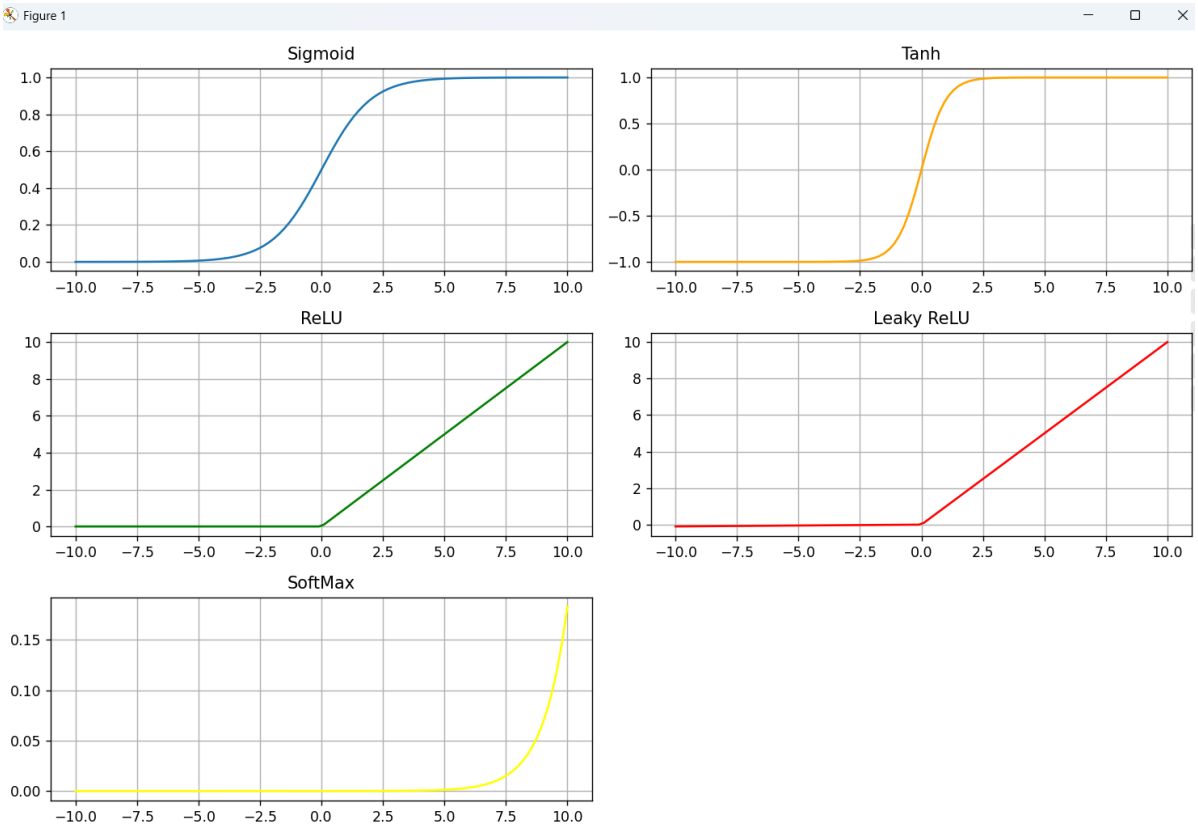
Q3. Implement the activation functions using python.

Code:

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

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

def tanh(x):
    return np.tanh(x)

def relu(x):
    return np.maximum(0, x)
def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

def softmax(x):
    exp_x = np.exp(x - np.max(x))
    return exp_x / np.sum(exp_x, axis=0, keepdims=True)

if __name__ == "__main__":
    x = np.linspace(-10, 10, 100)

    y_sigmoid = sigmoid(x)
    y_tanh = tanh(x)
    y_relu = relu(x)
    y_leaky_relu = leaky_relu(x)
    y_softMax = softmax(x)

    plt.figure(figsize=(12, 8))
    plt.subplot(3, 2, 1)
    plt.plot(x, y_sigmoid, label="Sigmoid")
    plt.title("Sigmoid")
    plt.grid()

    plt.subplot(3, 2, 2)
    plt.plot(x, y_tanh, label="Tanh", color='orange')
    plt.title("Tanh")
    plt.grid()

    plt.subplot(3, 2, 3)
    plt.plot(x, y_relu, label="ReLU", color='green')
    plt.title("ReLU")
```

```
plt.grid()

plt.subplot(3, 2, 4)
plt.plot(x, y_leaky_relu, label="Leaky ReLU", color='red')
plt.title("Leaky ReLU")
plt.grid()

plt.subplot(3, 2, 5)
plt.plot(x, y_softMax, label="SoftMax", color='yellow')
plt.title("SoftMax")
plt.grid()
plt.tight_layout()
plt.show()
```

Output:



Q4. Write a python program to implement all pooling operations (max , min , average) from a grey level or monochrome image and store it into another array and show the result .

Code:

```
import cv2
import matplotlib.pyplot as plt

def pooling(image, pool_size=2):
    """Perform different pooling operations"""
    # Convert to grayscale if needed
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2RGB) if len(image.shape) == 3 else
image

    # Perform pooling
    max_pool = cv2.resize(gray, (gray.shape[1]//pool_size,
gray.shape[0]//pool_size),
                          interpolation=cv2.INTER_NEAREST)
    min_pool = cv2.resize(gray, (gray.shape[1]//pool_size,
gray.shape[0]//pool_size),
                          interpolation=cv2.INTER_NEAREST)
    avg_pool = cv2.resize(gray, (gray.shape[1]//pool_size,
gray.shape[0]//pool_size),
                         interpolation=cv2.INTER_AREA)

    return max_pool, min_pool, avg_pool

def main():
    # Get image path from user
    image_path = input("Enter image path: ")

    try:
        # Read image
        image = cv2.imread(image_path)

        # Perform pooling
        max_pool, min_pool, avg_pool = pooling(image)

        # Convert original image to RGB for display
        original_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
```

```
        # Visualize results
        plt.figure(figsize=(12, 3))
        titles = ['Original', 'Max Pooling', 'Min Pooling', 'Average Pooling']
        images = [original_rgb, max_pool, min_pool, avg_pool]

        for i, (title, img) in enumerate(zip(titles, images), 1):
            plt.subplot(1, 4, i)
            plt.title(title)
            plt.imshow(img, cmap='gray' if i > 1 else None)
            plt.axis('off')

        plt.tight_layout()
        plt.show()

    except Exception as e:
        print(f"Error processing image: {e}")

if __name__ == "__main__":
    main()
```

Output: