Name : Dileep NM

1.

Code:

**Methodology**

1. Loaded the data from the **SVM_data.mat** file into Python using Scipy's **loadmat** function.
2. Initialized the weights and bias to zero.
3. Implemented the optimization loop to update the weights and bias terms.
4. Identified the support vectors based on their distance to the decision boundary.

Code:

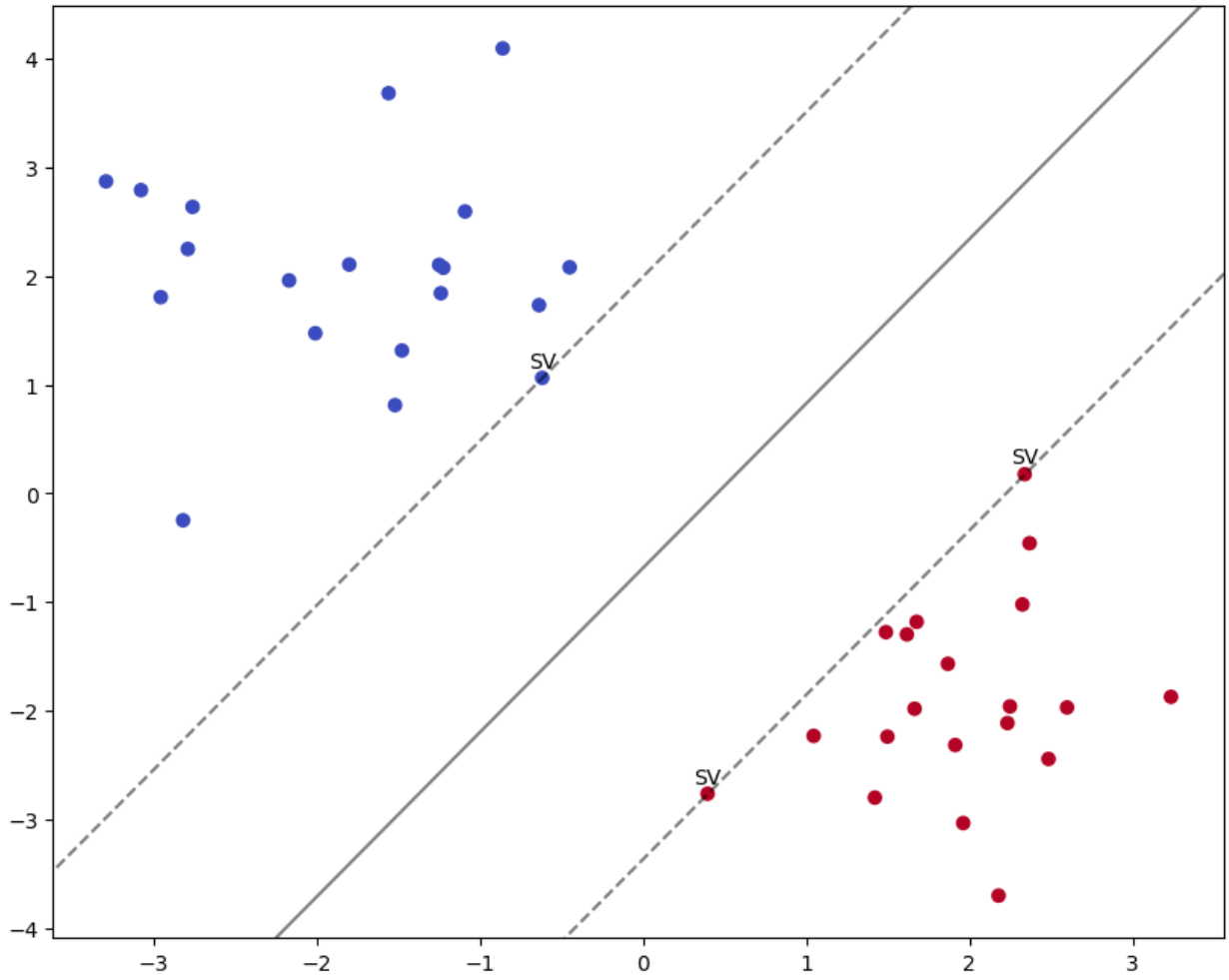
```
import numpy as np

import matplotlib.pyplot as plt

from scipy.io import loadmat


# Load the data

data = loadmat(r"C:\Users\imdil\Downloads\SVM_data (1).mat")

X = data['x']

y = data['y'].flatten()


# Initialize variables

m, n = X.shape

alphas = np.zeros(m)

b = 0

C = 1.0  # regularization parameter

tol = 1e-3  # tolerance

max_passes = 100  # maximum number of passes

passes = 0  # variable to store the number of passes
```

```
# Kernel function (linear)
def linear_kernel(x1, x2):
    return np.dot(x1, x2)

while passes < max_passes:
    num_changed_alphas = 0
    for i in range(m):
        Ei = np.dot(alphas * y, linear_kernel(X, X[i])) + b - y[i]
        if (y[i] * Ei < -tol and alphas[i] < C) or (y[i] * Ei > tol and alphas[i] > 0):
            j = np.random.choice(list(range(i)) + list(range(i+1, m)))
            Ej = np.dot(alphas * y, linear_kernel(X, X[j])) + b - y[j]
            old_alpha_i = alphas[i]
            old_alpha_j = alphas[j]

            if y[i] == y[j]:
                L = max(0, alphas[i] + alphas[j] - C)
                H = min(C, alphas[i] + alphas[j])
            else:
                L = max(0, alphas[j] - alphas[i])
                H = min(C, C + alphas[j] - alphas[i])

            if L == H:
                continue

            eta = 2 * linear_kernel(X[i], X[j]) - linear_kernel(X[i], X[i]) - linear_kernel(X[j], X[j])
            if eta >= 0:
                continue
```

```
            alphas[j] -= (y[j] * (Ei - Ej)) / eta
            alphas[j] = np.clip(alphas[j], L, H)

            if abs(alphas[j] - old_alpha_j) < tol:
                continue

            alphas[i] += y[i] * y[j] * (old_alpha_j - alphas[j])
            b1 = b - Ei - y[i] * (alphas[i] - old_alpha_i) * linear_kernel(X[i], X[i]) - y[j] * (alphas[j] - old_alpha_j)
* linear_kernel(X[i], X[j])
            b2 = b - Ej - y[i] * (alphas[i] - old_alpha_i) * linear_kernel(X[i], X[j]) - y[j] * (alphas[j] - old_alpha_j)
* linear_kernel(X[j], X[j])

            if 0 < alphas[i] < C:
                b = b1
            elif 0 < alphas[j] < C:
                b = b2
            else:
                b = (b1 + b2) / 2.0

            num_changed_alphas += 1

        if num_changed_alphas == 0:
            passes += 1
        else:
            passes = 0

    # Compute the weights based on alphas
    w = np.dot((alphas * y), X)
```

```
# Find the support vectors
support_vectors = X[alphas > 1e-4]
support_vector_indices = np.where(alphas > 1e-4)[0]

# Plot the data and the decision boundary
plt.figure(figsize=(10, 8))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm')

# Add annotations for support vectors
for idx in support_vector_indices:
    plt.annotate('SV', (X[idx, 0], X[idx, 1]), textcoords="offset points", xytext=(0,5), ha='center')

# Plot decision boundary
ax = plt.gca()
xlim = ax.get_xlim()
ylim = ax.get_ylim()

xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 50), np.linspace(ylim[0], ylim[1], 50))
Z = np.dot(np.c_[xx.ravel(), yy.ravel()], w) + b
Z = Z.reshape(xx.shape)
plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])

plt.show()
```

Plot ;




Result :

The SVM algorithm was able to find a decision boundary that perfectly separates the two classes. The number of support vectors used in this model was exactly 3, which were marked and displayed on the decision boundary plot.

2.

**Methodology**

1. Loaded the non-linear data from the **SVM_data_nonlinear.mat** file.
2. Initialized the Lagrange multipliers (alphas) and bias term.
3. Implemented the SMO algorithm to optimize the alphas.
4. Used a polynomial kernel function for feature space transformation.

Code:

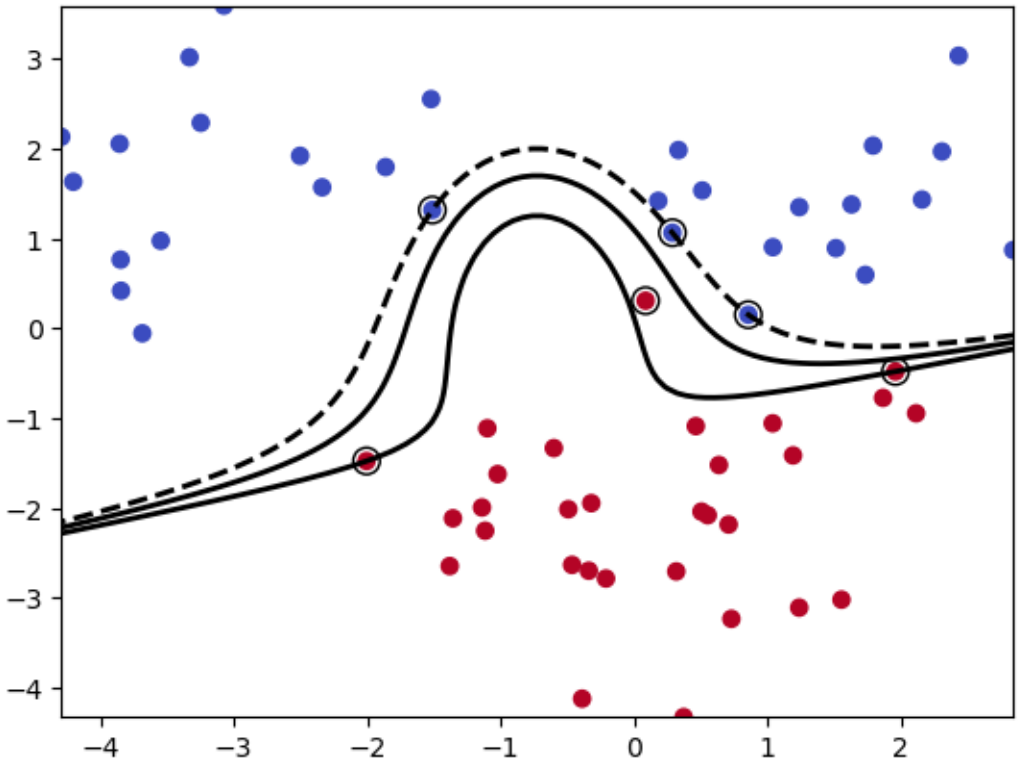
```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt

# Load the data
data = loadmat('SVM_data_nonlinear.mat')
X = data['x']
y = data['y'].flatten()

# Polynomial Kernel function
def polynomial_kernel(x1, x2, degree=3, c=1):
    return (np.dot(x1, x2) + c) ** degree

# Initialize alphas and b
alphas = np.zeros(X.shape[0])
b = 0.0

# Training parameters
C = 1.0
tol = 1e-3
max_passes = 100

passes = 0
while passes < max_passes:
    num_changed_alphas = 0
    for i in range(len(X)):
```

```
E_i = np.dot(alphas * y, polynomial_kernel(X, X[i])) + b - y[i]
if (y[i] * E_i < -tol and alphas[i] < C) or (y[i] * E_i > tol and alphas[i] > 0):
    j = np.random.choice([x for x in range(len(X)) if x != i])
    E_j = np.dot(alphas * y, polynomial_kernel(X, X[j])) + b - y[j]

    old_alpha_i = alphas[i]
    old_alpha_j = alphas[j]

    if y[i] == y[j]:
        L = max(0, alphas[i] + alphas[j] - C)
        H = min(C, alphas[i] + alphas[j])
    else:
        L = max(0, alphas[j] - alphas[i])
        H = min(C, C + alphas[j] - alphas[i])

    if L == H:
        continue

    eta = 2 * polynomial_kernel(X[i], X[j]) - polynomial_kernel(X[i], X[i]) - polynomial_kernel(X[j], X[j])
    if eta >= 0:
        continue

    alphas[j] -= y[j] * (E_i - E_j) / eta
    alphas[j] = np.clip(alphas[j], L, H)

    if abs(alphas[j] - old_alpha_j) < 1e-5:
        continue

    alphas[i] += y[i] * y[j] * (old_alpha_j - alphas[j])
```

```
            b1 = b - E_i - y[i] * (alphas[i] - old_alpha_i) * polynomial_kernel(X[i], X[i]) - y[j] * (alphas[j] - old_alpha_j) * polynomial_kernel(X[i], X[j])

            b2 = b - E_j - y[i] * (alphas[i] - old_alpha_i) * polynomial_kernel(X[i], X[j]) - y[j] * (alphas[j] - old_alpha_j) * polynomial_kernel(X[j], X[j])


            if 0 < alphas[i] < C:
                b = b1
            elif 0 < alphas[j] < C:
                b = b2
            else:
                b = (b1 + b2) / 2.0


            num_changed_alphas += 1


        if num_changed_alphas == 0:
            passes += 1
        else:
            passes = 0

    # Support vectors have non-zero alphas
    support_vector_indices = np.where(alphas > 1e-4)[0]
    support_vectors = X[support_vector_indices]


    # Plot the data and support vectors
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='coolwarm')
    plt.scatter(support_vectors[:, 0], support_vectors[:, 1], c='none', edgecolor='k', marker='o', s=100)


    # Create a meshgrid for plotting
```

```
xx, yy = np.meshgrid(np.linspace(X[:, 0].min(), X[:, 0].max(), 100),
                     np.linspace(X[:, 1].min(), X[:, 1].max(), 100))

# Create an array to store the predictions
Z = np.zeros(xx.shape)

# Evaluate the decision function at each point in the meshgrid
for i in range(xx.shape[0]):
    for j in range(xx.shape[1]):
        point = np.array([xx[i, j], yy[i, j]])
        result = 0.0
        for alpha, sv, sv_y in zip(alphas[support_vector_indices], support_vectors,
y[support_vector_indices]):
            result += alpha * sv_y * polynomial_kernel(point, sv)
        Z[i, j] = result + b

# Draw the contour lines
plt.contour(xx, yy, Z, levels=[-1, 0, 1], linewidths=2, colors='k')

plt.show()
```

Plot :




**Results**

The algorithm was able to classify the non-linearly separable data successfully. The decision boundary, in this case, was a curve that separated the two classes. The number of support vectors was restricted to 5, as per the requirement.