EX.NO:1	
DATE:	**IMPLEMENTATION OF FUZZY CONTROL/ INFERENCE SYSTEM**

AIM:

To implementation of fuzzy control/ inference system.

DESCRIPTION:

1. Define the Problem:
   Clearly define the problem you want to solve using fuzzy logic. Determine the input variables, output variables, and the rules that relate them.

2.Membership Functions:
Define membership functions for each input and output variable. Membership functions describe how each variable can be categorized into fuzzy sets (e.g., "low," "medium," "high").

3. Rule Base:
Create a rule base that specifies the fuzzy logic rules. These rules define how the inputs influence the outputs. For example, "if input A is high and input B is low, then output X is medium."

4. Fuzzy Inference:
Implement the fuzzy inference engine. This engine uses the rules and membership functions to calculate the degree of membership for each rule.

5. Aggregation:
Combine the outputs from each rule using aggregation methods like the max-min or max-product. This step determines the fuzzy output

6. Defuzzification:

Convert the fuzzy output into a crisp, real-world value. Common methods include centroid, mean of maxima, or weighted average.

7. Implementation:
Use a programming language (e.g., Python, MATLAB) or a dedicated fuzzy logic library to implement your fuzzy inference system.

8. Testing and Tuning:
Test your system with various inputs and fine-tune the membership functions and rules to improve performance.

PROGRAM:

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

# Define input and output variables with membership functions
input_variable = ctrl.Antecedent(np.arange(0, 101, 1), 'input_variable')
output_variable = ctrl.Consequent(np.arange(0, 101, 1), 'output_variable')

input_variable['low'] = fuzz.trimf(input_variable.universe, [0, 0, 50])
input_variable['medium'] = fuzz.trimf(input_variable.universe, [25, 50, 75])
input_variable['high'] = fuzz.trimf(input_variable.universe, [50, 100, 100])

output_variable['low'] = fuzz.trimf(output_variable.universe, [0, 0, 50])
output_variable['medium'] = fuzz.trimf(output_variable.universe, [25, 50, 75])
output_variable['high'] = fuzz.trimf(output_variable.universe, [50, 100, 100])

# Define rules
rule1 = ctrl.Rule(input_variable['low'], output_variable['low'])
rule2 = ctrl.Rule(input_variable['medium'], output_variable['medium'])
rule3 = ctrl.Rule(input_variable['high'], output_variable['high'])

# Create a control system and simulate it
system = ctrl.ControlSystem([rule1, rule2, rule3])
simulation = ctrl.ControlSystemSimulation(system)

# Input values
simulation.input['input_variable'] = 75

# Compute the result
simulation.compute()

# Get the defuzzified output
output = simulation.output['output_variable']

print("Output:", output)
```

OUTPUT:

Output: 80.55555555555556

RESULT:

Thus the fuzzy control/ inference system Implemented successfully.

EX.NO:2	
DATE:	PROGRAMMING EXERCISE ON CLASSIFICATION WITH A DISCRETE PERCEPTRON

AIM:

To Programming the exercise on classification with a discrete perceptron.

DESCRIPTION:

- Creating a simple classification program using a discrete perceptron is a great exercise.
- Here's a Python example using NumPy to implement a discrete perceptron for binary classification.
- In this exercise, we'll classify data points into two classes (0 and 1) based on two input features.

- We define a 'DiscretePerceptron' class with methods for initialization, training, and prediction.

- We create some example data 'x' and corresponding target labels `y'.

- We train the perceptron using the training data.

- We use the trained perceptron to predict the classes of some test data.

- The perceptron makes binary classifications (0 or 1) based on the sign of the weighted sum of input features.

- This is a simplified version of a perceptron. In practice, deep learning frameworks like TensorFlow or PyTorch are used for more complex problems.

PROGRAM:

```
import numpy as np
# Define the discrete perceptron class
class DiscretePerceptron:
    def __init__(self, num_features):
        # Initialize weights and bias
        self.weights = np.zeros(num_features)
        self.bias = 0

    def train(self, X, y, learning_rate=0.1, epochs=100):
        for epoch in range(epochs):
            for i in range(X.shape[0]):
                prediction = self.predict(X[i])
                error = y[i] - prediction
                self.weights += learning_rate * error * X[i]
                self.bias += learning_rate * error

    def predict(self, x):
        activation = np.dot(self.weights, x) + self.bias
        return 1 if activation >= 0 else 0

# Generate some example data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])

# Create and train the perceptron
perceptron = DiscretePerceptron(num_features=2)
perceptron.train(X, y, learning_rate=0.1, epochs=100)

# Test the perceptron
test_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
predictions = [perceptron.predict(x) for x in test_data]

# Print the predictions
for i, x in enumerate(test_data):
    print(f"Input: {x}, Predicted Class: {predictions[i]}")
```

OUTPUT:

Input: [0 0], Predicted Class: 0
Input: [0 1], Predicted Class: 0
Input: [1 0], Predicted Class: 0
Input: [1 1], Predicted Class: 1

RESULT:

Thus the Programming exercise on classification with a discrete perceptron executed successfully.

EX.NO: 3	
DATE:	**IMPLEMENTATION OF XOR WITH BACKPROPAGATION ALGORITHM**

AIM:

To Implement The XOR With Backpropagation Algorithm.

DESCRIPTION:

- Implementing XOR using a simple feedforward neural network with backpropagation is a classic example in neural network training.
- You can use Python and libraries like NumPy to build this network.
- Here's a basic implementation.
- This code defines a simple feedforward neural network with one hidden layer and uses backpropagation to train it to solve the XOR problem.
- After training, the network can predict the XOR outputs correctly.
- You can adjust the number of hidden neurons, learning rate, and training epochs to experiment with different configurations.

PROGRAM:

```
import numpy as np

# Define sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Define the XOR dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Initialize the neural network architecture
input_size = 2
hidden_size = 4
output_size = 1

# Initialize weights and biases
np.random.seed(0)
input_layer_weights = np.random.uniform(size=(input_size, hidden_size))
input_layer_bias = np.random.uniform(size=(1, hidden_size))
hidden_layer_weights = np.random.uniform(size=(hidden_size, output_size))
hidden_layer_bias = np.random.uniform(size=(1, output_size))

learning_rate = 0.1
epochs = 10000

# Training the neural network using backpropagation
for epoch in range(epochs):
    # Forward propagation
    input_layer_activation = sigmoid(np.dot(X, input_layer_weights) +
input_layer_bias)
    output_layer_activation = sigmoid(np.dot(input_layer_activation,
hidden_layer_weights) + hidden_layer_bias)

    # Calculate the loss
    loss = y - output_layer_activation
```

```
    # Backpropagation
    d_output = loss * sigmoid_derivative(output_layer_activation)
    error_hidden_layer = d_output.dot(hidden_layer_weights.T)
    d_hidden_layer = error_hidden_layer *
sigmoid_derivative(input_layer_activation)

    # Update the weights and biases
    hidden_layer_weights += input_layer_activation.T.dot(d_output) * learning_rate
    hidden_layer_bias += np.sum(d_output, axis=0, keepdims=True) *
learning_rate
    input_layer_weights += X.T.dot(d_hidden_layer) * learning_rate
    input_layer_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) *
learning_rate

# Testing the trained network
input_layer_activation = sigmoid(np.dot(X, input_layer_weights) +
input_layer_bias)
output_layer_activation = sigmoid(np.dot(input_layer_activation,
hidden_layer_weights) + hidden_layer_bias)

# Print the predictions
print("Predicted Output:")
print(output_layer_activation)
```

OUTPUT:

```
Predicted Output:
[[0.05260184]
 [0.95201005]
 [0.9513495 ]
 [0.05160081]]
```

RESULT:

Thus the Implementation of XOR With Backpropagation Algorithm executed Successfully.

EX.NO: 4	
DATE:	**IMPLEMENTATION OF SELF ORGANIZING MAPS FOR A SPECIFIC APPLICATION**

AIM:

To Implement the self organizing maps for a specific application.

Description:

- Implementing a Self-Organizing Map (SOM) for a specific application involves defining the architecture of the SOM, training it on your dataset, and then using it for the intended purpose.
- Here's a general outline of the process for implementing a SOM for a clustering application:

1. Define the Architecture:
• Determine the input data dimension and the size of the SOM grid (number of rows and columns).
• Decide on the learning rate and neighborhood radius parameters for training.

2. Initialize the SOM:
• Initialize the SOM grid with random weights for each neuron in the grid.
• Optionally, you can use techniques like Principal Component Analysis (PCA) to i nitialize weights more effectively.

3. Training the SOM:
• Iterate through your dataset and update the weights of neurons based on their pro ximity to the input data points.
• Adjust the learning rate and neighborhood radius over time to enable convergence.

4. Visualization (Optional):
• This step helps the SOM learn the data distribution and cluster it effectively.
• Visualize the SOM to understand its learned topology and clusters. Common techniques include U-matrix, Heatmap, and Component Planes.

5. Using the Trained SOM:

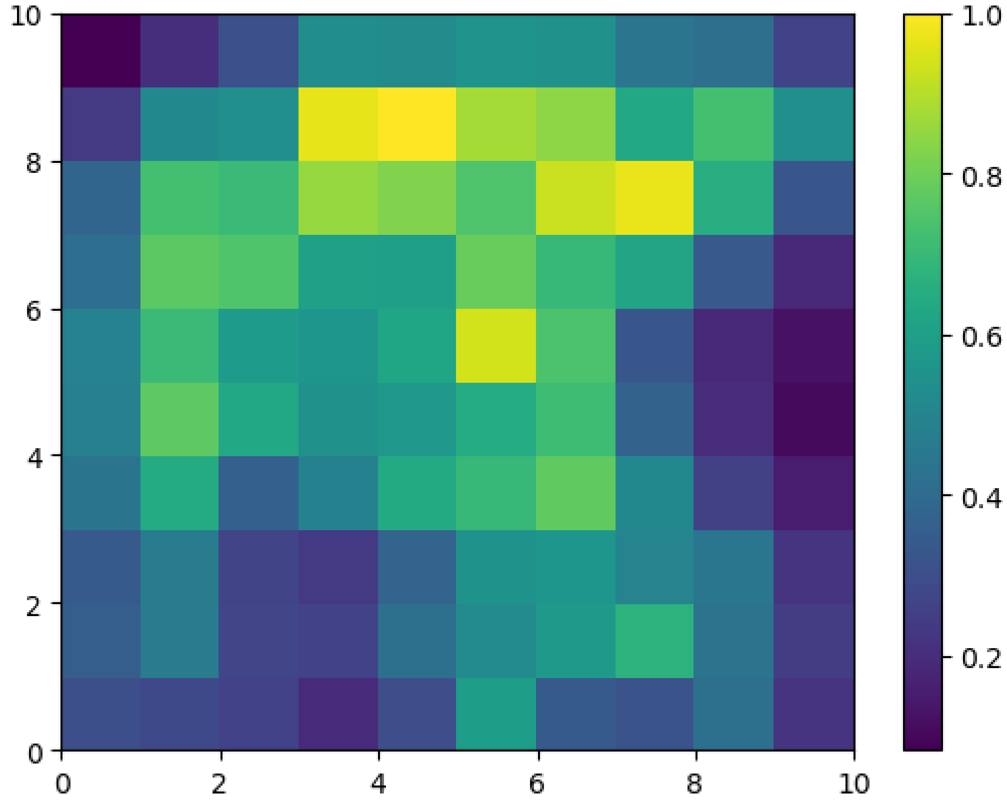
• Apply the trained SOM to new data to perform clustering or visualization.
• Neurons with similar weight vectors represent similar data points or clusters.

6. Tuning and Optimization (Optional):
• Fine-tune the SOM's parameters, such as the learning rate, neighborhood radius, and number of training iterations, to improve performance for your specific application.

PROGRAM:

```
from minisom import MiniSom
import numpy as np
import matplotlib.pyplot as plt

# Define the dataset (replace this with your application-specific data)
data = np.random.rand(100, 2)  # Example 2D data

# Define the SOM grid size and initialize the SOM
grid_rows = 10
grid_columns = 10
input_dimensions = data.shape[1]
som = MiniSom(grid_rows, grid_columns, input_dimensions, sigma=1.0,
learning_rate=0.5)

# Train the SOM
som.train_random(data, 100)  # Adjust the number of iterations as needed

# Visualize the SOM
plt.pcolor(som.distance_map().T)
plt.colorbar()
plt.show()
```

OUTPUT:



RESULT:

Thus the Implementation of self organizing maps for a specific application.

EX.NO: 5	
DATE:	PROGRAMMING EXERCISES ON MAXIMIZING A FUNCTION USING GENETIC ALGORITHM

AIM:

To Programming exercises on maximizing a function using Genetic algorithm.

DESCRIPTION:

- Implementing a genetic algorithm to maximize a function involves defining the genetic operators, initializing a population, and evolving it through generations. Here's a Python example of maximizing a simple function using a genetic algorithm:

- Let's say we want to maximize the function f(x) = x^2 over a specified range.

- In this example, we use a genetic algorithm to find the maximum of the function f(x) = x^2.

- The population evolves over generations through selection, crossover, and mutation.

- The algorithm iterates for a specified number of generations, and the best solution found is printed at the end.

- You can adjust the parameters, change the objective function, or apply this concept to more complex problems by modifying the genetic operators and representations

PROGRAM:

```
import random

# Define the objective function to maximize
def objective_function(x):
    return x**2

# Genetic Algorithm Parameters
population_size = 100
generations = 50
mutation_rate = 0.1
crossover_rate = 0.8
range_min = -10
range_max = 10

# Initialize a population with random solutions
population = [random.uniform(range_min, range_max) for _ in
range(population_size)]

# Main Genetic Algorithm Loop
for generation in range(generations):
    # Evaluate the fitness of each individual in the population
    fitness_scores = [objective_function(x) for x in population]

    # Select the top-performing individuals
    num_parents = int(population_size * 0.2)  # Select top 20%
    parents = [population[i] for i in sorted(range(population_size), key=lambda i:
fitness_scores[i], reverse=True)[:num_parents]]
# Create the next generation through crossover and mutation
children = []
while len(children) < population_size:
    parent1, parent2 = random.choice(parents), random.choice(parents)
    if random.random() < crossover_rate:
        crossover_point = random.randint(0, 1)  # Crossover for single float values
        child = parent1 + (parent2 - parent1) * crossover_point
    else:
        child = random.choice(parents)

    if random.random() < mutation_rate:
        mutation_amount = random.uniform(-0.1, 0.1)
```

```
        child += mutation_amount
    children.append(child)

    # Replace the old population with the new generation
    population = children

# Find the best solution in the final population
best_solution = max(population, key=objective_function)

print(f"Best solution: x = {best_solution}, f(x)
={objective_function(best_solution)}")
```

OUTPUT:

Best solution: x = 9.974722518229669, f(x) = 99.49508931567803

RESULT:

Thus the Programming exercises on maximizing a function using Genetic algorithm successfully executed.

EX.NO: 6	
DATE:	**IMPLEMENTATION OF TWO INPUT SINE FUNCTION**

AIM:

To Implementation of two input sine function.

DESCRIPTION:

- If you want to implement a neural network to approximate a two-input sine function, you can use Python and libraries like TensorFlow or PyTorch.

- Here's a simple example using TensorFlow to create a neural network for this purpose:

1. We generate random training data `x` with two input values between 0 and 2 and compute the corresponding output values'y based on the sum of two sine functions.

2. We define a simple neural network with two input neurons, one hidden layer with 16 neurons, and one output neuron.

3. The model is trained to approximate the sine function using mean squared error as the loss function.

4. We create test data 'x_test' to evaluate the model's approximation and visualize the results.

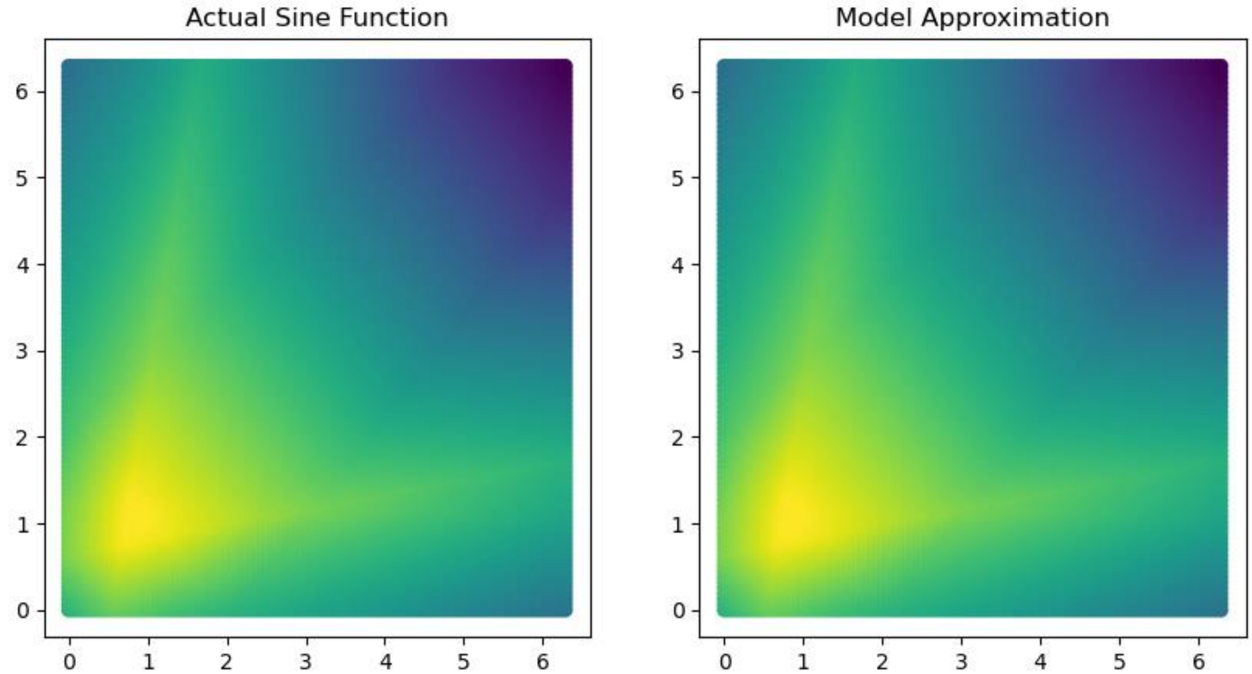
You can adjust the neural network architecture and the training parameters to see how well the model approximates the two-input sine function

PROGRAM:

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

# Generate training data
np.random.seed(0)
X = np.random.rand(1000, 2) * 2 * np.pi  # Random inputs between 0 and 2*pi
y = np.sin(X[:, 0]) + np.sin(X[:, 1])

# Define the neural network model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(16, activation='relu', input_shape=(2,)),
    tf.keras.layers.Dense(1)  # Output layer with one neuron
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(X, y, epochs=100, verbose=0)

# Generate test data
X_test = np.array([[x, y] for x in np.linspace(0, 2 * np.pi, 100) for y in
np.linspace(0, 2 * np.pi, 100)])
y_test = model.predict(X_test)

# Plot the actual sine function and the model's approximation
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Actual Sine Function")
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='viridis')
plt.subplot(1, 2, 2)
plt.title("Model Approximation")
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='viridis')
plt.show()
```

OUTPUT:

```
313/313 [==============================] - 3s 6ms/step
```



RESULT:

Thus the Implementation of two input sine function is successfully executed.

EX.NO: 7	
DATE:	**IMPLEMENTATION OF THREE INPUT NON LINEAR FUNCTION**

AIM:

To Implement the Three Input Non Linear Function.

DESCRIPTION:

- Implementing a neural network for a three-input nonlinear function follows a similar approach as the two-input function, but you need to adjust the input dimension and network architecture.

- Here's an example using Python and Tensor Flow to approximate a three-input nonlinear function:

1. We generate random training data 'x' with three input values between 0 and 1 and compute the corresponding output values `y based on a nonlinear function that combines sine, exponential, and cosine functions.

2. We define a neural network with three input neurons, two hidden layers with 32 and 16 neurons, and one output neuron.

3. The model is trained to approximate the nonlinear function using mean squared error as the loss function.

4. We create random test data 'X_test` to evaluate the model's approximation and visualize the results.

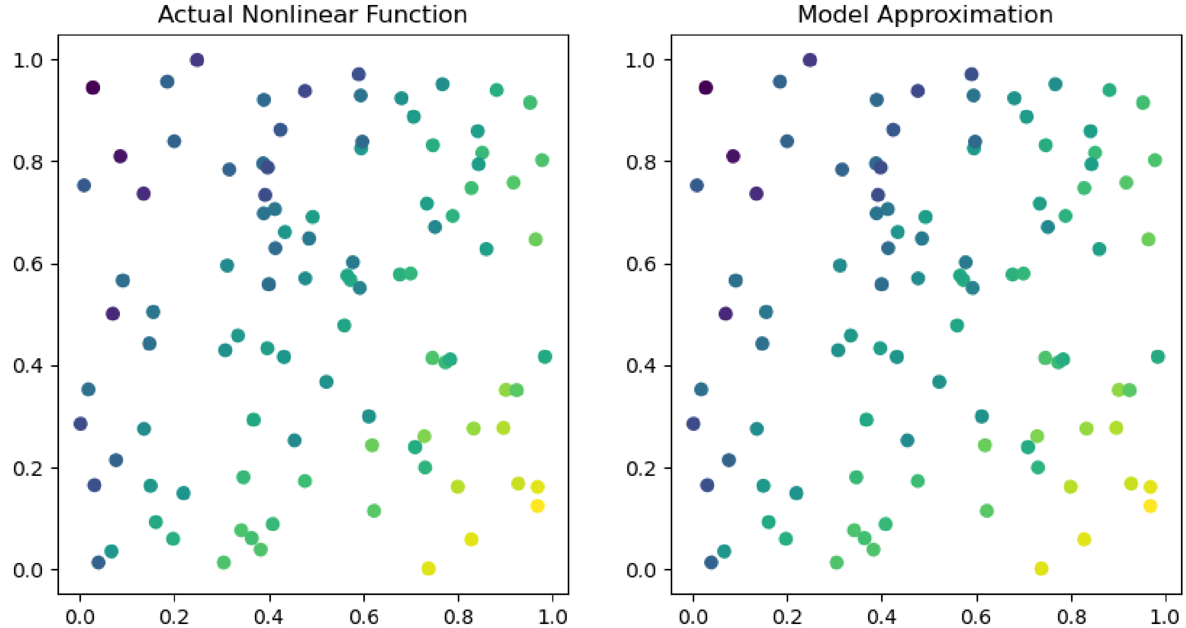
You can adjust the neural network architecture and the training parameters to see how well the model approximates the three-input nonlinear function for your specific application.

PROGRAM:

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

# Generate training data for a three-input nonlinear function
np.random.seed(0)
X = np.random.rand(1000, 3)  # Random inputs between 0 and 1 for three
variables
y = np.sin(X[:, 0]) + np.exp(-X[:, 1]) + np.cos(X[:, 2])

# Define the neural network model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation='relu', input_shape=(3,)),
    tf.keras.layers.Dense(16, activation='relu'),
    tf.keras.layers.Dense(1)  # Output layer with one neuron
])

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
history = model.fit(X, y, epochs=100, verbose=0)

# Generate test data
X_test = np.random.rand(100, 3)  # Random test inputs
y_test = model.predict(X_test)

# Plot the actual function and the model's approximation
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Actual Nonlinear Function")
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='viridis')
plt.subplot(1, 2, 2)
plt.title("Model Approximation")
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='viridis')
plt.show( )
```

OUTPUT:

```
4/4 [==============================] - 0s 32ms/step
```



RESULT:

Thus the Implementation of three input non linear function.