Practical 7A

 # A. Aim: Write a program for Linear separation in python.

```
import numpy as np
# Perceptron class for linear separation
class Perceptron:
    def __init__(self, num_features):
        self.weights = np.zeros(num_features)
        self.bias = 0.0
    def predict(self, inputs):
        weighted_sum = np.dot(self.weights, inputs) + self.bias
        return 1 if weighted_sum >= 0 else 0
    def train(self, training_data, labels, learning_rate, epochs):
        for epoch in range(epochs):
            for i in range(len(training_data)):
                inputs = training_data[i]
                label = labels[i]
                prediction = self.predict(inputs)
                if prediction != label:
                    update = (label - prediction) * learning_rate
                    self.weights += update * inputs
                    self.bias += update
# Example usage for linear separation
if __name__ == "__main__":
    # Define training data and labels for a simple linearly separable problem
    training_data = np.array([[1, 2], [2, 3], [3, 1], [4, 4], [5, 5], [6, 4]])
    labels = np.array([0, 0, 0, 1, 1, 1])
    # Create a Perceptron instance
    num_features = training_data.shape[1]
    perceptron = Perceptron(num_features)
    # Train the Perceptron with the training data
    learning_rate = 0.1
    epochs = 100
    perceptron.train(training_data, labels, learning_rate, epochs)
    # Test the trained Perceptron with new data points
    test_data = np.array([[2, 2], [4, 3], [5, 6]])
    for data_point in test_data:
        prediction = perceptron.predict(data_point)
        print(f"Prediction for {data_point}: Class {prediction}")
```

Output:

```
Prediction for [2 2]: Class 0
Prediction for [4 3]: Class 1
Prediction for [5 6]: Class 1
```

Practical 7B

 # B. Aim: Write a program for Hopfield network model for associative memory

```
import numpy as np
class HopfieldNetwork:
   def __init__(self, num_neurons):
       self.num_neurons = num_neurons
       self.weights = np.zeros((num_neurons, num_neurons))
   def train(self, patterns):
       num_patterns = len(patterns)
       for pattern in patterns:
           pattern = np.array(pattern)
           self.weights += np.outer(pattern, pattern)
       np.fill_diagonal(self.weights, 0)
       self.weights /= num_patterns
   def energy(self, state):
       return -0.5 * np.dot(state, np.dot(self.weights, state))
   def update_rule(self, state):
       h = np.dot(self.weights, state)
       return np.where(h >= 0, 1, -1)
if __name__ == "__main__":
    patterns = [[1, 1, -1, -1], [-1, -1, 1, 1]]
    num_neurons = len(patterns[0])
    hopfield_net = HopfieldNetwork(num_neurons)
    hopfield_net.train(patterns)
    test_patterns = [[1, 1, 1, -1], [-1, 1, 1, 1]]
    for pattern in test_patterns:
        initial_state = np.array(pattern)
        iterations = 5
        print(f"Initial state: {initial_state}")
        for i in range(iterations):
            new_state = hopfield_net.update_rule(initial_state)
            print(f"Iteration {i + 1}: {new_state}")
            if np.array_equal(new_state, initial_state):
                print("Converged to a stable state.")
                break
            initial_state = new_state
```

Output:

```
Initial state: [ 1  1  1 -1]
Iteration 1: [ 1  1 -1 -1]
Iteration 2: [ 1  1 -1 -1]
Converged to a stable state.
Initial state: [-1  1  1  1]
Iteration 1: [-1 -1  1  1]
Iteration 2: [-1 -1  1  1]
Converged to a stable state.
```