SOFT COMPUTING LAB

(Course Code: 22UPCSC1E12)

A programming laboratory record submitted to Periyar University, Salem
In partial fulfillment of the requirements for the degree of

MASTER OF COMPUTER APPLICATIONS

By

ELANCHEZHIAN M.
[Reg. No.: U22PG507CAP006]

DEPARTMENT OF COMPUTER SCIENCE

PERIYAR UNIVERSITY

(NAAC `A++` Grade with CGPA 3.61) – NIRF RANK 59 – ARIIA RANK 10

PERIYAR PALKALAI NAGAR,

SALEM – 636 011.

(APRIL - 2023)

CERTIFICATE

This is to certify that the Programming Laboratory entitled "**SOFT COMPUTING LAB (22UPCSC2E12)**" is a bonafide record work done by Mr. /Ms. ____ Register No: _______________________as partial fulfillment of the requirements for the degree of Master of Computer Application, in the Department of Computer Science, Periyar University, Salem, during the Academic Year 2022-2023.

Staff In-charge   Head of the Department

Submitted for the practical examination held on……...………………

Internal Examiner   External Examiner

CONTENTS

1. Implementation of Logic gates using Artificial Neural Network

SOURCE CODE:

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

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

def inPa(inFe,neInHi,ouFe):
    w1=np.random.randn(neInHi,inFe)
    w2=np.random.randn(ouFe,neInHi)
    b1=np.zeros((neInHi,1))
    b2=np.zeros((ouFe,1))
    parameters={"w1" : w1, "b1": b1,"w2" : w2, "b2": b2}
    return parameters

def forPro(X,Y,parameters):
    m=X.shape[1]
    w1=parameters["w1"]
    w2=parameters["w2"]
    b1=parameters["b1"]
    b2=parameters["b2"]
    z1=np.dot(w1,X)+b1
    a1=sigmoid(z1)
    z2=np.dot(w2,a1)+b2
    a2=sigmoid(z2)
    cache=(z1,a1,w1,b1,z2,a2,w2,b2)
    logprobs = np.multiply(np.log(a2),Y)+np.multiply(np.log(1-a2),(1-Y))
```

```
    cost = -np.sum(logprobs)/m
    return cost,cache,a2

def baPro(X,Y,cache):
    m=X.shape[1]
    (z1,a1,w1,b1,z2,a2,w2,b2)=cache
    dz2 =a2-Y
    dw2=np.dot(dz2,a1.T)/m
    db2 = np.sum(dz2,axis=1,keepdims =True)
    da1=np.dot(w2.T,dz2)
    dz1=np.multiply(da1,a1*(1-a1))
    dw1=np.dot(dz1,X.T)/m
    db1=np.sum(dz1,axis= 1, keepdims =True)/m
    gradients ={"dz2": dz2, "dw2": dw2, "db2": db2,"dz1": dz1,
    "dw1": dw1,"db1": db1}
    return gradients

def upPar(parameters, gradients, learningRate):
    parameters["w1"]=parameters["w1"] - learningRate * gradients["dw1"]
    parameters["w2"]=parameters["w2"] - learningRate * gradients["dw2"]
    parameters["b1"]=parameters["b1"] - learningRate * gradients["db1"]
    parameters["b2"]=parameters["b2"] - learningRate * gradients["db2"]
    return parameters

X =np.array([[0, 0, 1, 1], [0, 1, 0, 1]])
Y = np.array([[0, 0, 0, 1]])
neInHi=2
inFe=X.shape[0]
ouFe=Y.shape[0]
parameters = inPa(inFe,neInHi,ouFe)
epoch =100000
```

```
learningRate=0.01
losses =np.zeros((epoch,1))
for i in range(epoch):
        losses[i,0],cache,a2=forPro(X,Y,parameters)
        gradients=baPro(X,Y,cache)
        parameters=upPar(parameters, gradients, learningRate)

plt.figure()
plt.plot(losses)
plt.xlabel("EPOCHS")
plt.ylabel("LOSS VALUE")
plt.show()

X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]])
cost, _, a1=forPro(X,Y,parameters)
prediction =(a2>0.5)*1.0

print(prediction)
```

OUTPUT:

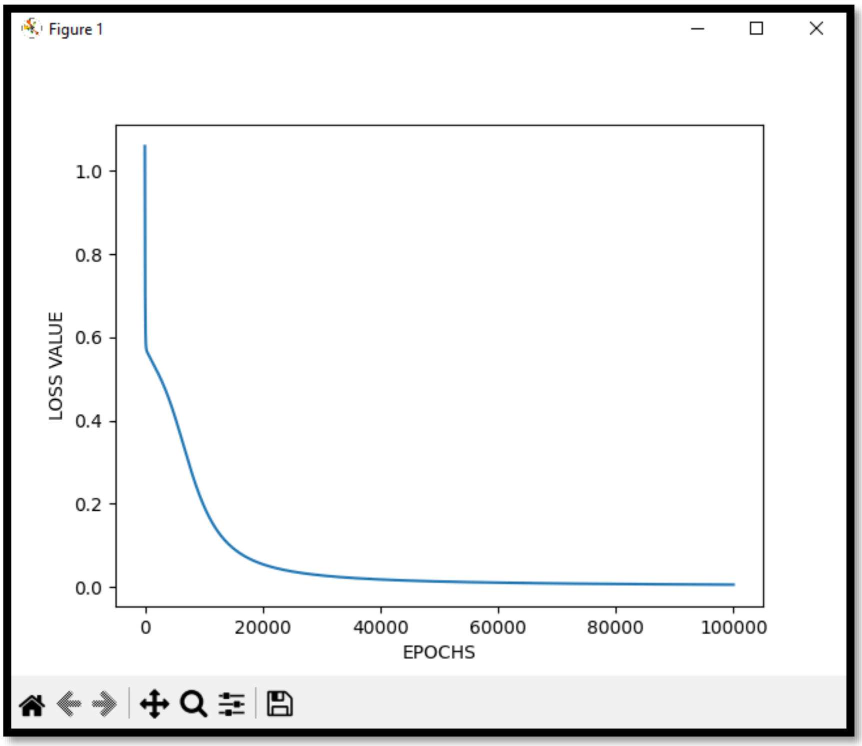


Figure 1
1.0
0.8
0.6
0.4
0.2
0.0
LOSS VALUE
0
20000
40000
60000
80000
100000
EPOCHS

[[0. 0. 0. 1.]]

3. Implementation of Back Propagation Algorithm

SOURCE CODE:

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

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

def initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures):
    W1 = np.random.randn(neuronsInHiddenLayers, inputFeatures)
    W2 = np.random.randn(outputFeatures, neuronsInHiddenLayers)
    b1 = np.zeros((neuronsInHiddenLayers, 1))
    b2 = np.zeros((outputFeatures, 1))
    parameters = {"W1": W1, "b1": b1,"W2":W2, "b2": b2}
    return parameters

def forwardPropagation(X, Y, parameters):
    m = X.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]
    Z1 = np.dot(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
```

```
    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)
    logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))
    cost = -np.sum(logprobs) / m
    return cost, cache, A2

def backwardPropagation(X, Y, cache):
    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2) = cache
    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis = 1, keepdims = True)
    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, A1 * (1- A1))
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis = 1, keepdims = True) / m
    gradients = {"dZ2": dZ2, "dW2": dW2, "db2": db2,
    "dZ1": dZ1, "dW1": dW1, "db1": db1}
    return gradients

def updateParameters(parameters, gradients, learningRate):
    parameters["W1"] = parameters["W1"] - learningRate * gradients["dW1"]
    parameters["W2"] = parameters["W2"] - learningRate * gradients["dW2"]
    parameters["b1"] = parameters["b1"] - learningRate * gradients["db1"]
    parameters["b2"] = parameters["b2"] - learningRate * gradients["db2"]
    return parameters
```

```
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])
Y = np.array([[0, 1, 1, 0]])
neuronsInHiddenLayers = 2
inputFeatures = X.shape[0]
outputFeatures = Y.shape[0]
parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers,
outputFeatures)
epoch = 100
learningRate = 0.01
losses = np.zeros((epoch, 1))
for i in range(epoch):
    losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)
    print("Epoch:", i, "Loss", losses[i])
    gradients = backwardPropagation(X, Y, cache)
    parameters = updateParameters(parameters, gradients, learningRate)

plt.figure()
plt.plot(losses)
plt.xlabel("EPOCHS")
plt.ylabel("LOSS VALUE")
plt.show()


X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]])
cost, _, A2 = forwardPropagation(X, Y, parameters)
prediction = (A2 > 0.5) * 1.0

print(prediction)
```

OUTPUT:

Epoch: 0 Loss [0.76693298]

Epoch: 1 Loss [0.76543201]

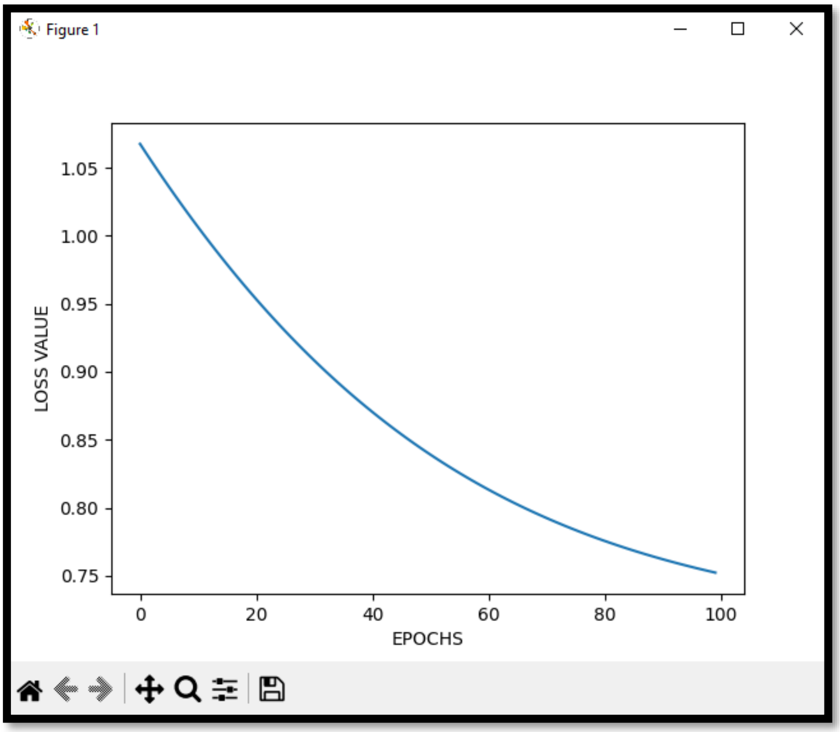
.

.

.

Epoch: 98 Loss [0.70380534]

Epoch: 99 Loss [0.70362287]



[[0. 0. 0. 0.]]

2. Implementation of Perception Algorithm

SOURCE CODE:

```
import numpy as np
def unitStep(v):
    if v >= 0:
        return 1
    else:
        return 0
def perceptronModel(x, w, b):
    v = np.dot(w, x) + b
    y = unitStep(v)
    return y
def OR_logicFunction(x):
    w = np.array([1,1])
    b = 0.5
    return perceptronModel(x, w, b)
test1 = np.array([0,1])
test2 = np.array([1,1])
test3 = np.array([0,0])
test4 = np.array([1,0])

print("OR({},{}) = {}".format(0,1, OR_logicFunction(test1)))
print("OR({},{}) = {}".format(1,1, OR_logicFunction(test2)))
print("OR({},{}) = {}".format(0,0, OR_logicFunction(test3)))
print("OR({},{}) = {}".format(0,1, OR_logicFunction(test4)))
```

OUTPUT:

```
OR(0,1) = 1
OR(1,1) = 1
OR(0,0) = 1
OR(0,1) = 1
```

4. Implementation of Self Organizing Maps

SOURCE CODE:

```
import math
class SOM:
    def winner(self, weights, sample):
        D0 = 0
        D1 = 0
        for i in range(len(sample)):
            D0 = D0 + math.pow((sample[i] - weights[0][1]),2)
            D1 = D1 + math.pow((sample[i] - weights[1][i]),2)
            if D0 > D1:
                return 0
            else:
                return 1

    def update(self, weights, sample, J, alpha):
        for i in range(len(weights)):
            weights[J][i] = weights[J][i] + alpha *(sample[i] -weights[J][i])
        return weights

def main():
        T = [[1, 1, 0, 0],[0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]
        m, n = len(T), len(T[0])
        weights = [[0.2, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]
        ob = SOM()
```

```
    epochs = 3000
    alpha = 0.5

    for i in range(epochs):
        for j in range(m):
            sample = T[j]
            J = ob.winner(weights, sample)
            weights = ob.update(weights, sample, J, alpha)

    s = [0, 0, 0, 1]
    J = ob.winner(weights, s)
    print("Test sample s belongs to Cluster : ", J)
    print("Trained weights : ",weights)

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

OUTPUT:

```
Test sample s belongs to Cluster :  0
Trained weights :  [[0.6666666666666666, 0.6666666666666667, 0.5, 0.9], [0.33333333333333337, 0.06666666666666667, 0.7, 0.3]]
```

5. Implementation of Radial Basis Function Network

SOURCE CODE:

```
import numpy as np

class RBFNN:
    def __init__(self, kernels, centers, beta=1, lr=0.1, epochs=80):
        self.kernels = kernels
        self.centers = centers
        self.beta = beta
        self.lr = lr
        self.epochs = epochs
        self.w = np.random.randn(kernels, 1)
        self.b = np.random.randn(1, 1)
        self.errors = []  # Changed 'error' to 'errors' for consistency
        self.gradients = []

    def rbf_activation(self, x, center):
        return np.exp(-self.beta * np.linalg.norm(x - center)**2)

    def linear_activation(self, A):
        return np.dot(self.w.T, A) + self.b

    def least_square_error(self, pred, y):
        return (y - pred)**2

    def _forward_propagation(self, x):
        A1 = np.array([[self.rbf_activation(x, center)] for center in self.centers])
        A2 = self.linear_activation(A1)
        return A2, A1
```

```
    def _backward_propagation(self, y, pred, A1):
        error = self.least_square_error(pred, y)
        dw = -2 * np.dot(A1, (y - pred).T)
        db = -2 * (y - pred)
        self.w = self.w - self.lr * dw
        self.b = self.b - self.lr * db
        self.errors.append(error)

    def fit(self, x, y):
        for _ in range(self.epochs):
            for xi, yi in zip(x, y):
                pred, A1 = self._forward_propagation(xi)
                self._backward_propagation(yi, pred, A1)

    def predict(self, x):
        A2, _ = self._forward_propagation(x)
        return 1 if np.squeeze(A2) >= 0.5 else 0

def main():
    x = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0], [1], [1], [0]])

    rbf = RBFNN(kernels=2, centers=np.array([[0, 1], [1, 0]]), beta=1,
    lr=0.1,epochs=1000)
    rbf.fit(x, y)

    print(f"RBF weights: {rbf.w}")
    print(f"RBF bias: {rbf.b}")
    print()
    print("---XOR GATE---")
    print(f"| 1 xor 1: {rbf.predict(x[3])} |")
```

```
    print(f"| 0 xor 0: {rbf.predict(x[0])} |")
    print(f"| 1 xor 0: {rbf.predict(x[2])} |")
    print(f"| 0 xor 1: {rbf.predict(x[1])} |")

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

OUTPUT:

```
RBF weights: [[2.50264625]
 [2.50264595]]
RBF bias: [[-1.8413432]]

---XOR GATE---
| 1 xor 1: 0 |
| 0 xor 0: 0 |
| 1 xor 0: 1 |
| 0 xor 1: 1 |
```

6. Implementation of De-Morgan‘s Law

SOURCE CODE:

```
def demorgan_law(statement):
    if 'not (' in statement and ' and ' in statement:
        # not (A and B) = (not A) or (not B)
        A, B = statement.split('not (')[1].split(' and ')
        return f'(not {A}) or (not {B})'
    elif 'not (' in statement and ' or ' in statement:
        # not (A or B) = (not A) and (not B)
        A, B = statement.split('not (')[1].split(' or ')
        return f'(not {A}) and (not {B})'
    else:
        return statement
```

OUTPUT:

```
statement1 = 'not (A and B)'
demorgan_law(statement1)
'(not A) or (not B))'
statement2 = 'not (A or B)'
demorgan_law(statement2)
'(not A) and (not B))'
statement3 = 'C and D'
demorgan_law(statement3)
'C and D'
```

8. Implementation of Simple genetic algorithm

SOURCE CODE:

```
from numpy.random import randint
from numpy.random import rand

def onemax(x):
    return -sum(x)

def selection(pop, scores, k=3):
    selection_ix = randint(len(pop))
    for ix in randint(0, len(pop), k-1):
        if scores[ix] < scores[selection_ix]:
            selection_ix = ix
    return pop[selection_ix]

def crossover(p1, p2, r_cross):
    c1, c2 = p1.copy(), p2.copy()
    if rand() < r_cross:
        pt = randint(1, len(p1)-2)
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    return [c1, c2]

def mutation(bitstring, r_mut):
    for i in range(len(bitstring)):
        if rand() < r_mut:
            bitstring[i] = 1 - bitstring[i]
```

```
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross, r_mut):
    pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
    best, best_eval = 0, objective(pop[0])
    for gen in range(n_iter):
        scores = [objective(c) for c in pop]

        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                print(">%d, new best f(%s) = %.3f" % (gen, pop[i], scores[i]))
        selected = [selection(pop, scores) for _ in range(n_pop)]
        children = list()
        for i in range(0, n_pop, 2):
            p1, p2 = selected[i], selected[i+1]
            for c in crossover(p1, p2, r_cross):
                mutation(c, r_mut)
                children.append(c)
        pop = children
    return [best, best_eval]

n_iter = 100
n_bits = 20
n_pop = 100
r_cross = 0.3
r_mut = 1.0 / float(n_bits)
best, score = genetic_algorithm(onemax, n_bits, n_iter, n_pop, r_cross, r_mut)
print('Done!')
print('f(%s) = %f' % (best, score))
```

OUTPUT:

```
>0, new best f([1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0]) = -15.000
>1, new best f([1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]) = -16.000
>2, new best f([1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0]) = -17.000
>5, new best f([1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]) = -18.000
>9, new best f([1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -19.000
>10, new best f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000
Done!
f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) = -20.000000
```

9. Implementation of fuzzy based Logical operations

SOURCE CODE:

```
A = dict()
B = dict()
Y = dict()

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}

print('The First Fuzzy Set is :', A)
print('The Second Fuzzy Set is :', B)

for A_key, B_key in zip(A, B):
    A_value = A[A_key]
    B_value = B[B_key]
    if A_value > B_value:
        Y[A_key] = A_value
    else:
        Y[B_key] = B_value

print('Fuzzy Set Union is :', Y)

A = dict()
B = dict()
Y = dict()
A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}
print('The First Fuzzy Set is :', A)
print('The Second Fuzzy Set is :', B)
```

```
for A_key, B_key in zip(A, B):
    A_value = A[A_key]
    B_value = B[B_key]

    if A_value < B_value:
        Y[A_key] = A_value
    else:
        Y[B_key] = B_value

print('Fuzzy Set Intersection is :', Y)
Y = dict()
A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
print('The Fuzzy Set is :', A)

for A_key in A:
    Y[A_key] = 1 - A[A_key]
print('Fuzzy Set Complement is :', Y)
```

OUTPUT:

```
The First Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
The Second Fuzzy Set is : {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}
Fuzzy Set Union is : {'a': 0.9, 'b': 0.9, 'c': 0.6, 'd': 0.6}
The First Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
The Second Fuzzy Set is : {'a': 0.9, 'b': 0.9, 'c': 0.4, 'd': 0.5}
Fuzzy Set Intersection is : {'a': 0.2, 'b': 0.3, 'c': 0.4, 'd': 0.5}
The Fuzzy Set is : {'a': 0.2, 'b': 0.3, 'c': 0.6, 'd': 0.6}
Fuzzy Set Complement is : {'a': 0.8, 'b': 0.7, 'c': 0.4, 'd': 0.4}
```

10. Implementation of fuzzy based arithmetic operations

SOURCE CODE:

```
def fuzzy_addition(a, b):
    result = {}
    for key_a, value_a in a.items():
        for key_b, value_b in b.items():
            if key_a == key_b:
                result[key_a] = max(value_a, value_b)
                break
    return result

def fuzzy_subtraction(a, b):
    result = {}
    for key_a, value_a in a.items():
        for key_b, value_b in b.items():
            if key_a == key_b:
                result[key_a] = max(value_a - value_b, 0)
                break
    return result

def fuzzy_multiplication(a, b):
    result = {}
    for key_a, value_a in a.items():
        for key_b, value_b in b.items():
            if key_a == key_b:
                result[key_a] = value_a * value_b
                break
    return result
```

```python
def fuzzy_division(a, b):
    result = {}
    for key_a, value_a in a.items():
        for key_b, value_b in b.items():
            if key_a == key_b and value_b != 0:
                result[key_a] = value_a / value_b
                break
    return result

A = {"a": 0.2, "b": 0.3, "c": 0.6, "d": 0.6}
B = {"a": 0.9, "b": 0.9, "c": 0.4, "d": 0.5}
C = fuzzy_addition(A, B)
print("Fuzzy set addition: ", C)

D = fuzzy_subtraction(A, B)
print("Fuzzy set subtraction: ", D)

E = fuzzy_multiplication(A, B)
print("Fuzzy set multiplication: ", E)

F = fuzzy_division(A, B)
print("Fuzzy set division: ", F)
```

OUTPUT:

```
Fuzzy set addition:  {'a': 0.9, 'b': 0.9, 'c': 0.6, 'd': 0.6}
Fuzzy set subtraction:  {'a': 0, 'b': 0, 'c': 0.19999999999999996, 'd': 0.09999999999999998}
Fuzzy set multiplication:  {'a': 0.18000000000000002, 'b': 0.27, 'c': 0.24, 'd': 0.3}
Fuzzy set division:  {'a': 0.22222222222222224, 'b': 0.3333333333333333, 'c': 1.4999999999999998, 'd': 1.2}
```

7. Implementation of McCulloch Pits Artificial Neuron model

SOURCE CODE:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import sklearn.datasets

cancer = sklearn.datasets.load_breast_cancer()
data = pd.DataFrame(cancer.data, columns=cancer.feature_names)
data["class"] = cancer.target
data.head()

plt.plot(data.T, "*")
plt.xticks(rotation="vertical", c="red", size=5)
plt.yticks(c="blue")
plt.show()

x = data.drop("class", axis=1)
y = data["class"]
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
x_train_binarized = x_train.apply(pd.cut, bins=2, labels=[1, 0]).values
x_test_binarized = x_test.apply(pd.cut, bins=2, labels=[1, 0]).values
plt.figure(figsize=(15, 5))
plt.plot(x_train_binarized.T, "*")
plt.xticks(c="red", size=13)
plt.yticks(c="blue", size=13)
plt.show()
```

```
class MP_Neuron:
    def __init__(self):
        self.b = 0

    def model(self, x):
        return np.sum(x) >= self.b

    def fit(self, x, y):
        accuracy = {}
        for b in range(x.shape[1] + 1):
            self.b = b
            yhat = []
            for row in x:
                yhat.append(self.model(row))
            accuracy[b] = accuracy_score(yhat, y)
        best_b = max(accuracy, key=accuracy.get)
        self.b = best_b
        return accuracy, best_b, accuracy[best_b]

    def predict(self, x, y):
        yhat = []
        for row in x:
            yhat.append(self.model(row))
        accuracy = accuracy_score(y, yhat)
        return accuracy
neuron = MP_Neuron()

accuracy, best_b, accuracy_model = neuron.fit(x_train_binarized, y_train)
print("The optimal value of b:", best_b)
print("Accuracy of model on training data:", accuracy_model * 100)
```

```
accuracies = list(accuracy.values())
plt.plot(accuracies)
plt.xticks(c="red")
plt.yticks(c="blue")
plt.show()
accuracy = neuron.predict(x_test_binarized, y_test)
print("Accuracy of model on test data:", accuracy * 100)
```

OUTPUT:

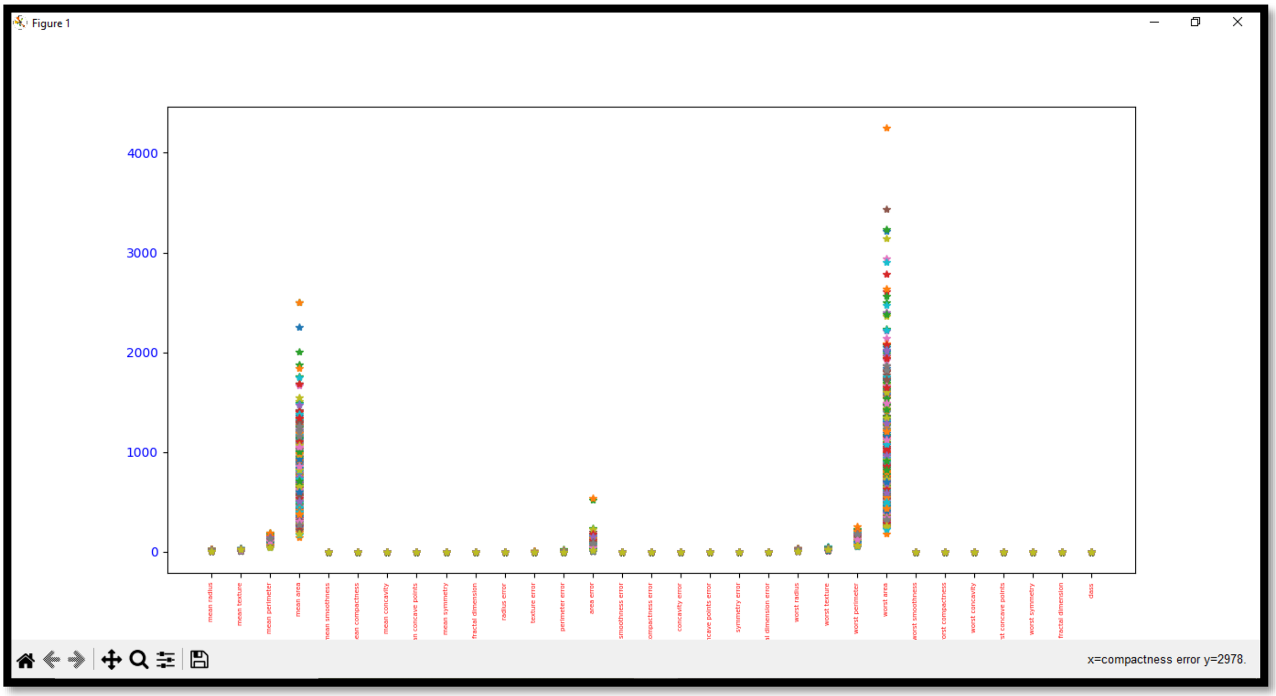


Figure 1
4000
3000
2000
1000
0
mean radius
mean texture
mean perimeter
mean area
mean smoothness
ean compactness
mean concavity
n concave points
mean symmetry
fractal dimension
radius error
texture error
perimeter error
area error
smoothness error
ompactness error
concavity error
ncave points error
symmetry error
l dimension error
worst radius
worst texture
worst perimeter
worst area
worst smoothness
orst compactness
worst concavity
st concave points
worst symmetry
fractal dimension
class
x=compactness error y=2978.

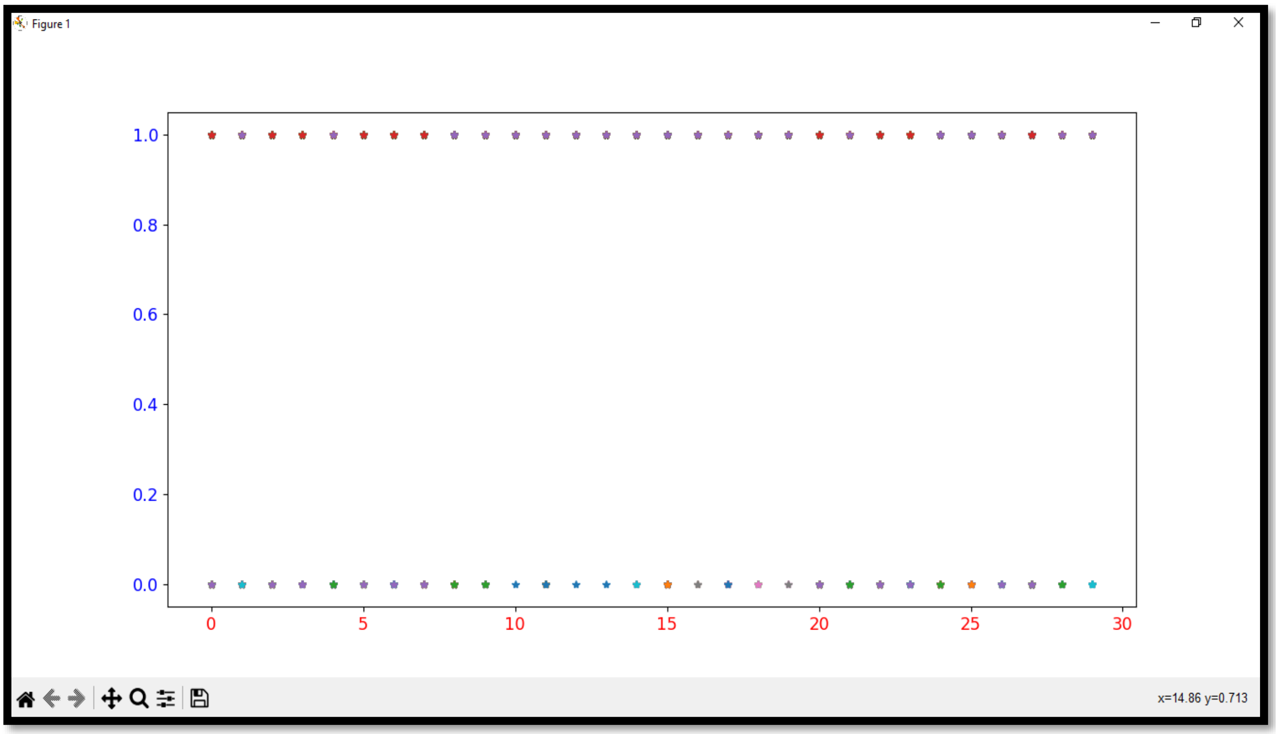
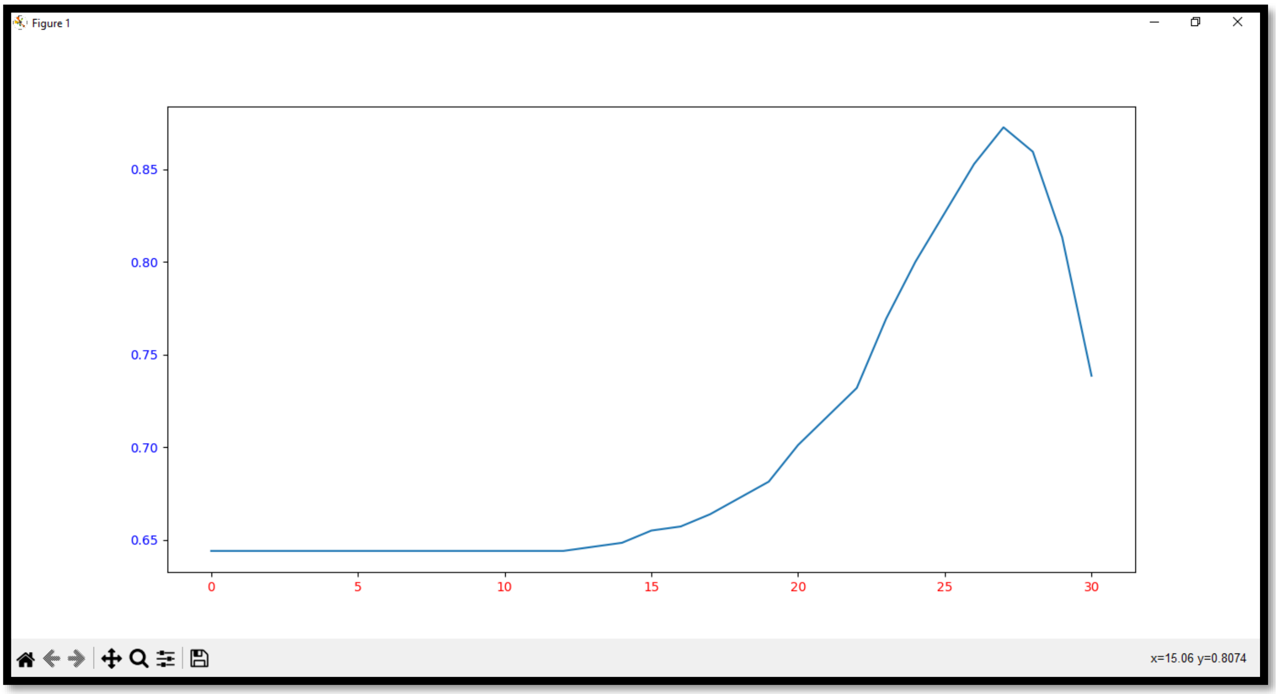


Figure 1
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
30
x=14.86 y=0.713



The optimal value of b: 27
Accuracy of model on training data: 86.37362637362638
Accuracy of model on test data: 80.7017543859649