**COURSE NAME: MACHINE LEARNING**

**COURSE CODE: CSC 323**

**ASSIGNMENT: MACHINE LEARNING ALGORITHMS**

**GROUP MEMBERS:**

1. COLLINS MESONGO P15/1335/2011
2. STEVE WAWERU P15/40614/2011
3. KIRUI JOAN P15/1334/2011

Contents

[GENETIC ALGORITHM 2](#_Toc386267843)

[BACK PROPAGATION 6](#_Toc386267844)

[**Steps :** 6](#_Toc386267845)

[**Implementation** 7](#_Toc386267846)

[Functions 7](#_Toc386267847)

[Output 9](#_Toc386267848)

[DECISION TREE 10](#_Toc386267849)

[K NEAREST NEIGHBOUR 15](#_Toc386267850)

[PERCEPTRON 19](#_Toc386267851)

[SELF ORGANISING MAPS 22](#_Toc386267852)

[K MEANS 25](#_Toc386267853)

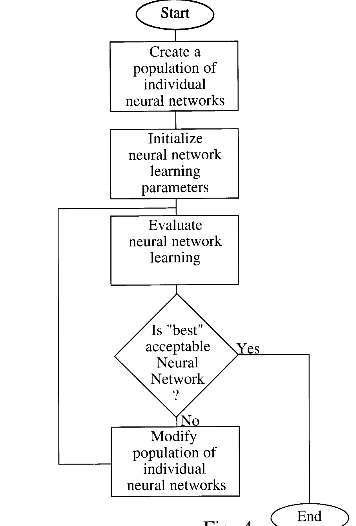
[ADALINE 28](#_Toc386267854)

[REINFORCEMENT LEARNING 31](#_Toc386267855)

# 

# GENETIC ALGORITHM

**Flowchart**



# 

**Algorithm**

1. **[Start]** Generate random population of n chromosomes (suitable solutions for the problem)
2. **[Fitness]** Evaluate the fitness f(x) of each chromosome x in the population
3. **[New population]** Create a new population by repeating following steps until the new population is complete
   1. **[Selection]** Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
   2. **[Crossover]** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
   3. **[Mutation]** With a mutation probability mutate new offspring at each locus (position in chromosome).
   4. **[Accepting]** Place new offspring in a new population
4. **[Replace]** Use new generated population for a further run of algorithm
5. **[Test]** If the end condition is satisfied, stop, and return the best solution in current population
6. **[Loop]** Go to step 2

**Problem statement**

Given the digits 0 through 9 and the operators +, -, \* and /, find a sequence that will represent a given target number. The operators will be applied sequentially from left to right as you read

Example, given the target number 23, the sequence 6+5\*4/2+1 would be one possible solution.

**Stage 1: Encoding**

First we need to encode a possible solution as a string of bits i,e a chromosome. So how do we do this? Well, first we need to represent all the different characters available to the solution... that is 0 through 9 and +, -, \* and /. This will represent a gene. Each chromosome will be made up of several genes.

Four bits are required to represent the range of characters used:

0: 0000

1: 0001

2: 0010

3: 0011

4: 0100

5: 0101

6: 0110

7: 0111

8: 1000

9: 1001

+: 1010

-: 1011

\*: 1100

/: 1101

The above show all the different genes required to encode the problem as described. The possible genes 1110 & 1111 will remain unused and will be ignored by the algorithm if encountered.

So now you can see that the solution mentioned above for 23, ' 6+5\*4/2+1' would be represented by nine genes like so:

0110 1010 0101 1100 0100 1101 0010 1010 0001

6 + 5 \* 4 / 2 + 1

These genes are all strung together to form the chromosome:

011010100101110001001101001010100001

**Stage 2: Deciding on a Fitness Function**

A fitness score can be assigned that's inversely proportional to the difference between the solution and the value a decoded chromosome represents.

If we assume the target number is 42, making use of the chromosome that is being used as an example

011010100101110001001101001010100001

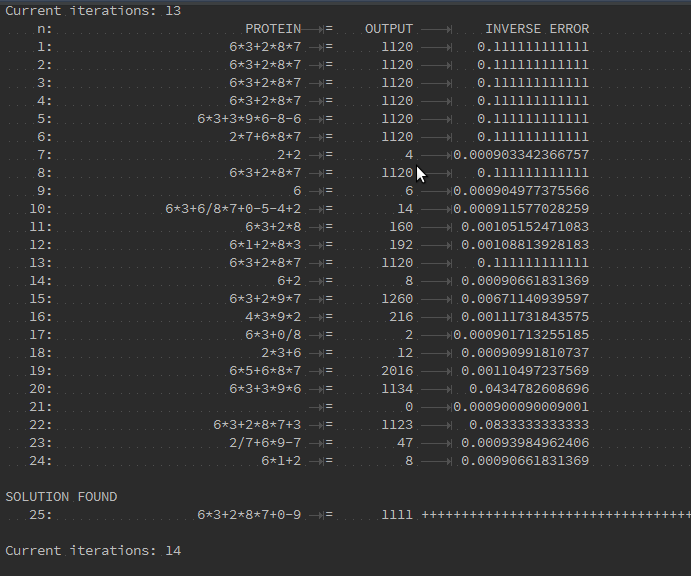
has a fitness score of 1/(42-23) or 1/19.

As it stands, if a solution is found, a divide by zero error would occur as the fitness would be 1/(42-42). This is not a problem however as we have found what we were looking for... a solution. Therefore a test can be made for this occurrence and the algorithm halted accordingly.

**Stage 3: Running the Code**

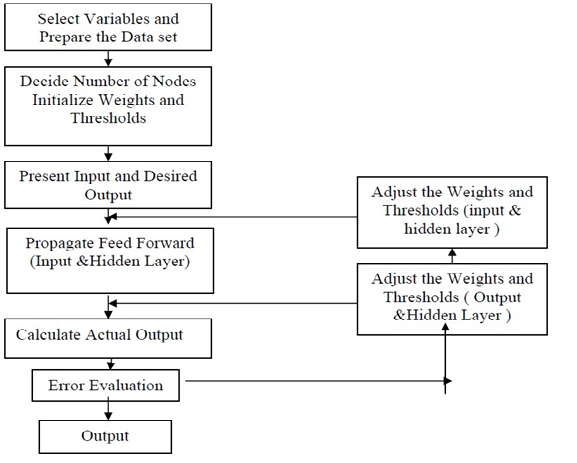
The code given will parse a chromosome bit string into the values we have discussed and it will attempt to find a solution which uses all the valid symbols it has found. Therefore if the target is 42, + 6 \* 7 / 2 would not give a positive result even though the first four symbols("+ 6 \* 7") do give a valid solution.

**Output**



# BACK PROPAGATION

**Flow chart**



**Steps :**

1. Initialize the network with small random weights
2. Present an input pattern to the input layer of the network
3. Feed the input pattern forward through the network to calculate its activation value
4. Take the difference between desired output and the activation value to calculate the network’s activation error
5. Adjust the weights feeding the output neuron to reduce its activation error for this input pattern
6. Propagate an error value back to each hidden neuron that is proportional to their contribution of the network’s activation error.
7. Adjust the weights feeding each hidden neuron to reduce their contribution of error for this input pattern.
8. Repeat steps 2 to 7 for each input pattern in the input collection.
9. Repeat step 8 until the network is suitably trained.

**Implementation**

First, let us create a data structure with our input data which will be analyzed later using a neural network.

**pat = [**

**[[0,0], [0]],**

**[[0,1], [1]],**

**[[1,0], [1]],**

**[[1,1], [0]]**

**]**

The goal of our neural net is to rediscover the relation between 2 inputs and the output, pretending that we know nothing about the such dependence.

A next step is to build a back propogation artificial neural network with 2 inputs, 2 hidden layers and one output.

**n = NN(2, 2, 1)**

Now we have prepared data and NN. Next step is to train the NN using the input data. It should be noted that it is desirable to rescale the input values so they will be between [-1,1].

**n.train(pat)**

And then the data is tested.

**n.test(pat)**

# Functions

calculating a random number where: a <= rand < b

**def rand(a, b):**

**return (b-a)\*random.random() + a**

The sigmoid function, tanh is easier to compute than the standard 1/(1+e^-x)

**def sigmoid(x):**

**return math.tanh(x)**

The activations for nodes

**self.ai = [1.0]\*self.ni**

**self.ah = [1.0]\*self.nh**

**self.ao = [1.0]\*self.no**

Creating the weights

**self.wi = makeMatrix(self.ni, self.nh)**

**self.wo = makeMatrix(self.nh, self.no)**

Setting the weights to random values

**for i in range(self.ni):**

**for j in range(self.nh):**

**self.wi[i][j] = rand(-0.2, 0.2)**

**for j in range(self.nh):**

**for k in range(self.no):**

**self.wo[j][k] = rand(-2.0, 2.0)**

Function that does the backpropagation

**def backPropagate(self, targets, N, M):**

**if len(targets) != self.no:**

**raise ValueError('wrong number of target values')**

calculating the error terms for output

**output\_deltas = [0.0] \* self.no**

**for k in range(self.no):**

**error = targets[k]-self.ao[k]**

**output\_deltas[k] = dsigmoid(self.ao[k]) \* error**

Calculating the error terms for hidden nodes

**hidden\_deltas = [0.0] \* self.nh**

**for j in range(self.nh):**

**error = 0.0**

**for k in range(self.no):**

**error = error + output\_deltas[k]\*self.wo[j][k]**

**hidden\_deltas[j] = dsigmoid(self.ah[j]) \* error**

updating the output weights

**for j in range(self.nh):**

**for k in range(self.no):**

**change = output\_deltas[k]\*self.ah[j]**

**self.wo[j][k] = self.wo[j][k] + N\*change + M\*self.co[j][k]**

**self.co[j][k] = change**

**#print N\*change, M\*self.co[j][k]**

updating the input weights

**for i in range(self.ni):**

**for j in range(self.nh):**

**change = hidden\_deltas[j]\*self.ai[i]**

**self.wi[i][j] = self.wi[i][j] + N\*change + M\*self.ci[i][j]**

**self.ci[i][j] = change**

calculation of the error

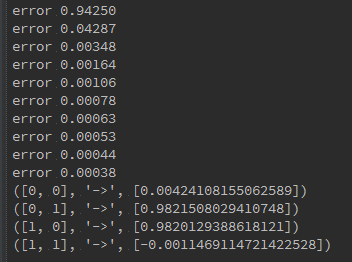
**error = 0.0**

**for k in range(len(targets)):**

**error = error + 0.5\*(targets[k]-self.ao[k])\*\*2**

**return error**

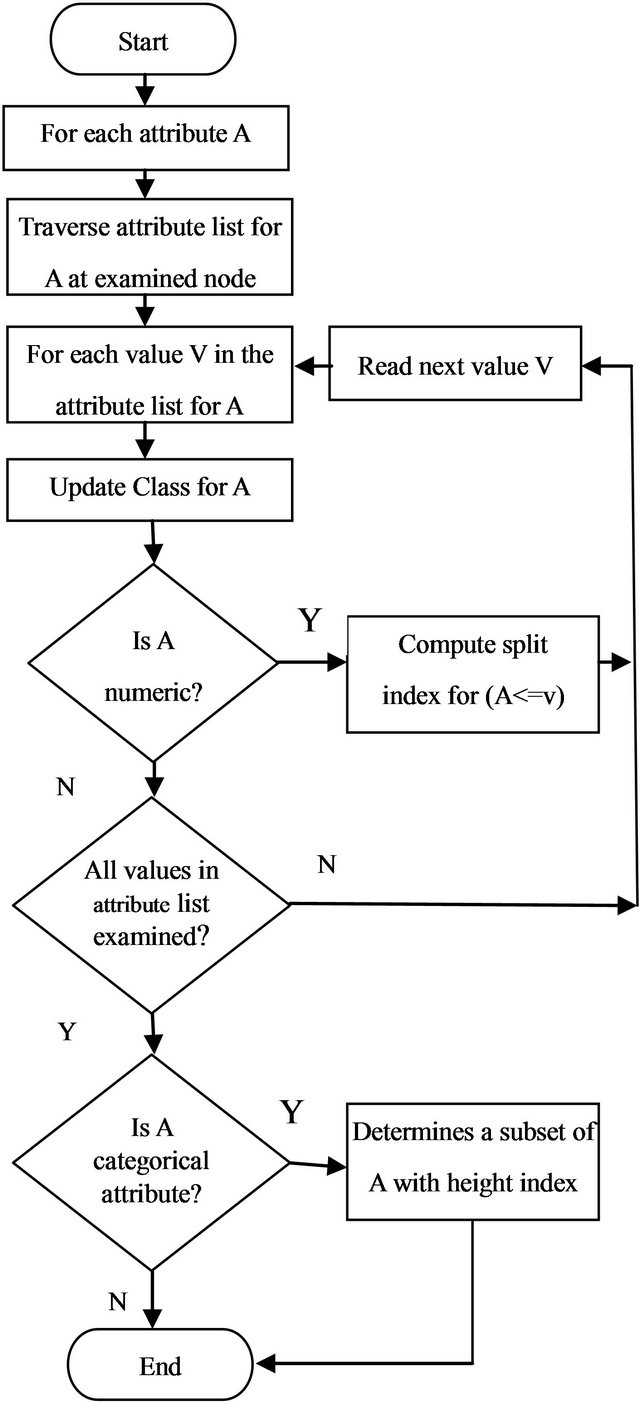
# Output

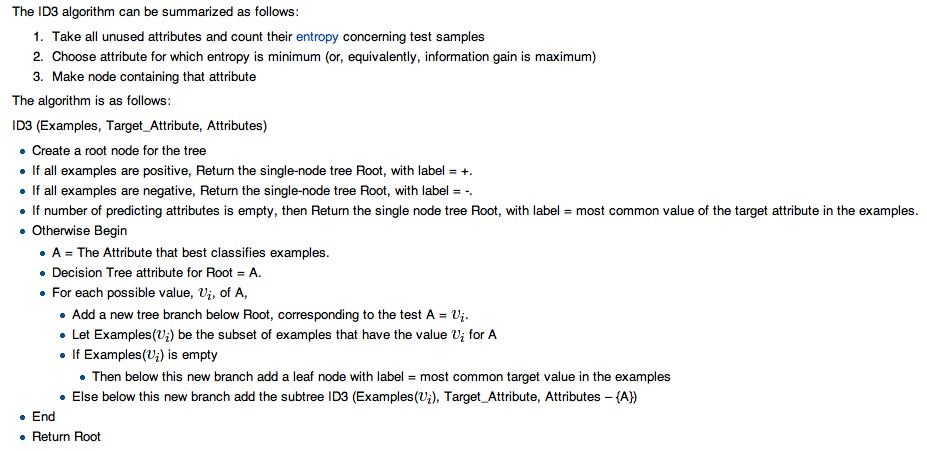


The output of this scripts gives the training error, and the last lines are the predictions after the training.

# DECISION TREE

**Flowchart**



**Pseudo code**

**Data structure to represent the tree**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Tree:  def \_\_init\_\_(self, parent=None):  self.parent = parent  self.children = []  self.splitFeature = None  self.splitFeatureValue = None  self.label = None |

In particular, now that features can have more than two possible values, we need to allow for an arbitrarily long list of child nodes. In addition, we add three pieces of data (with default values None): the splitFeature is the feature for which each of its children assumes a separate value; the splitFeatureValue is the feature assumed for its parent’s split; and the label (which is None for all interior nodes) is the final classification label for a leaf.

**Data representation**

Data is represented as a set list of pairs of the form (point, label), where the point is itself a list of the feature values, and the label is a string.

Now given a data set the first thing we need to do is compute its entropy. For that we can first convert it to a distribution (in the sense defined above, a list of probabilities which sum to 1):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def dataToDistribution(data):  ''' Turn a dataset which has n possible classification labels into a  probability distribution with n entries. '''  allLabels = [label for (point, label) in data]  numEntries = len(allLabels)  possibleLabels = set(allLabels)  dist = []  for aLabel in possibleLabels:  dist.append(float(allLabels.count(aLabel)) / numEntries)  return dist |

And we can compute the entropy of such a distribution:

|  |  |
| --- | --- |
| 1  2  3 | def entropy(dist):  ''' Compute the Shannon entropy of the given probability distribution. '''  return -sum([p \* math.log(p, 2) for p in dist]) |

In order to compute the gain of a data set by splitting on a particular value, we need to be able to split the data set. To do this, we identify features with their index in the list of feature values of a given data point, enumerate all possible values of that feature, and generate the needed subsets one at a time. In particular, we use a Python generator object:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def splitData(data, featureIndex):  ''' Iterate over the subsets of data corresponding to each value  of the feature at the index featureIndex. '''  # get possible values of the given feature  attrValues = [point[featureIndex] for (point, label) in data]  for aValue in set(attrValues):  dataSubset = [(point, label) for (point, label) in data  if point[featureIndex] == aValue]  yield dataSubset |

**Entropy Gain**

So to compute the gain, we simply need to iterate over the set of all splits, and compute the entropy of each split. In code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def gain(data, featureIndex):  ''' Compute the expected gain from splitting the data along all possible  values of feature. '''  entropyGain = entropy(dataToDistribution(data))  for dataSubset in splitData(data, featureIndex):  entropyGain -= entropy(dataToDistribution(dataSubset))  return entropyGain |

The best split (represented as the best feature to split on) is given by such a line of code as:

bestFeature = max(range(n), key=lambda index: gain(data, index))

Base case will be when we run out of data to split; that is, when our input data all have the same classification label. To check for this we implement a function called “homogeneous”

|  |  |
| --- | --- |
| 1  2  3 | def homogeneous(data):  ''' Return True if the data have the same label, and False otherwise. '''  return len(set([label for (point, label) in data])) <= 1 |

|  |  |
| --- | --- |
|  |  |

Here we see the base cases, and the selection of the best feature to split on. As a side remark, we observe this is not the most efficient implementation. We admittedly call the gain function and splitData functions more often than necessary, but we feel what is lost in runtime speed is gained in code legibility.

Once we bypass the three base cases, and we have determined the right split, we just do it:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def buildDecisionTree(data, root, remainingFeatures):  ''' Build a decision tree from the given data, appending the children  to the given root node (which may be the root of a subtree). '''  # add child nodes and process recursively  for dataSubset in splitData(data, bestFeature):  aChild = Tree(parent=root)  aChild.splitFeatureValue = dataSubset[0][0][bestFeature]  root.children.append(aChild)  buildDecisionTree(dataSubset, aChild, remainingFeatures - set([bestFeature]))  return root |

Here we iterate over the subsets of data after the split, and create a child node for each. We then assign the child its corresponding feature value in the splitFeatureValue variable, and append the child to the root’s list of children.

Now the first call to this function requires some initial parameter setup, so we define a convenience function that only requires a single argument: the data.

|  |  |
| --- | --- |
| 1  2 | def decisionTree(data):  return buildDecisionTree(data, Tree(), set(range(len(data[0][0])))) |

**Classifying New Data**

The last piece of the puzzle is to classify a new piece of data once we’ve constructed the decision tree. This is a considerably simpler recursive process. If the current node is a leaf, output its label. Otherwise, recursively search the subtree (the child of the current node) whose splitFeatureValue matches the new data’s choice of the feature being split. In code,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def classify(tree, point):  ''' Classify a data point by traversing the given decision tree. '''  if tree.children == []:  return tree.label  else:  matchingChildren = [child for child in tree.children  if child.splitFeatureValue == point[tree.splitFeature]]  return classify(matchingChildren[0], point) |

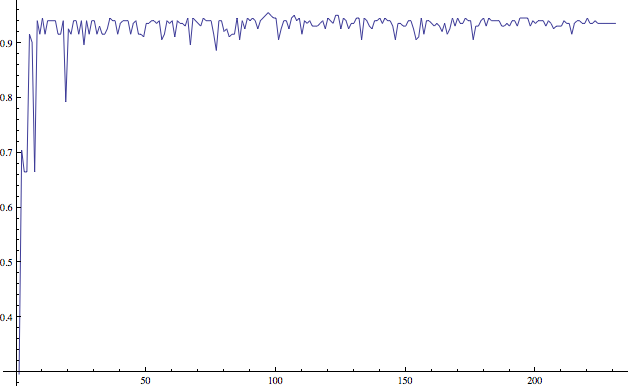
|  |  |
| --- | --- |
|  |  |

**Using the Data Set**

Our goal is to learn party membership based on the voting records. This data set is rife with missing values, roughly half of the members abstained from voting on some of these measures. So we constructed a decision tree from the clean portion of the data, and use that to classify the remainder of the data.

|  |  |
| --- | --- |
| 1  2  5  6  7  8  9 | with open('house-votes-1984.txt', 'r') as inputFile:  lines = inputFile.readlines()  data = [line.strip().split(',') for line in lines]  data = [(x[1:], x[0]) for x in data]  cleanData = [x for x in data if '?' not in x[0]]  noisyData = [x for x in data if '?' in x[0]]  tree = decisionTree(cleanData) |

**Output**

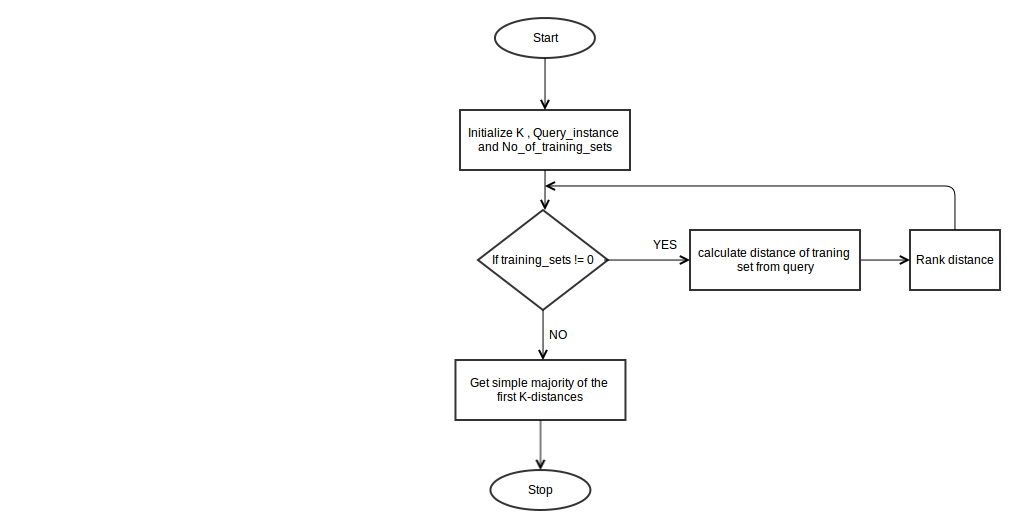


The size of the subset used to construct the tree versus its accuracy in classifying the remainder of the data. Note that the subsets were chosen uniformly at random without replacement. The x-axis is the number of points used to construct the tree, and the y-axis is the proportion of labels correctly classified.

Inspecting the trees generated in this process, it appears that the most prominent feature to split on is the adoption of a new budget resolution. Very few Democrats voted in favor of this, so for many of the random subsets of the data, a split on this feature left one side homogeneously Republican.

# K NEAREST NEIGHBOUR

**Flow chart**



**Algorithm and implementation**

def read\_data(fn):

""" read dataset and separate into input data

and label data

"""

**# read dataset file**

with open(fn) as f:

raw\_data = np.loadtxt(f, delimiter= ',', dtype="float",

skiprows=1, usecols=None)

**# initilize list**

data = []; label = []

**#assign input data and label data**

for row in raw\_data:

data.append(row[:-1])

label.append(int(row[-1]))

**# return input data and label data**

return np.array(data), np.array(label)

def knn(k, dtrain, dtest, dtr\_label, dist=1):

""" k-nearest neighbors """

**# initialize list to store predicted class**

pred\_class = []

**# for each instance in data testing,**

**# calculate distance in respect to data training**

for ii, di in enumerate(dtest):

distances = [] # initialize list to store distance

for ij, dj in enumerate(dtrain):

**# calculate distances**

distances.append((calc\_dist(di,dj,dist), ij))

**# k-neighbors**

k\_nn = sorted(distances)[:k]

**# predict the class for the instance**

pred\_class.append(classify(k\_nn, dtr\_label))

**# return prediction class**

return pred\_class

def calc\_dist(di,dj,i=1):

""" Distance calculation for every

distance functions in use"""

if i == 1:

return ssd.euclidean(di,dj) # built-in Euclidean fn

elif i == 2:

return ssd.cityblock(di,dj) # built-in Manhattan fn

elif i == 3:

return ssd.cosine(di,dj) # built-in Cosine fn

def classify(k\_nn, dtr\_label):

""" Classify instance data test into class"""

dlabel = []

for dist, idx in k\_nn:

**# retrieve label class and store into dlabel**

dlabel.append(dtr\_label[idx])

**# return prediction class**

return np.argmax(np.bincount(dlabel))

def evaluate(result):

""" Evaluate the prediction class"""

**# create eval result array to store evaluation result**

eval\_result = np.zeros(2,int)

for x in result:

**# increment the correct prediction by 1**

if x == 0:

eval\_result[0] += 1

# increment the wrong prediction by 1

else:

eval\_result[1] += 1

# return evaluation result

return eval\_result

def main():

**""" k-nearest neighbors classifier """**

# initialize runtime

start = time.clock()

# data tests, 1 = breast cancer data test,

# 2 = iris data test

data\_tests = [1,2]

for d in data\_tests:

if d == 1:

# read dataset of breast cancer

dtrain, dtr\_label = read\_data('breast-cancer-train.csv')

dtest, true\_class = read\_data('breast-cancer-test.csv')

else:

# read dataset of breast cancer

dtrain, dtr\_label = read\_data('iris-train.csv')

dtest, true\_class = read\_data('iris-test.csv')

# initialize K

K = [1,3,7,11]

# distance function for euclidean (1), manhattan (2),

# and cosine (3)

dist\_fn = [1,2,3]

if d == 1:

print "k-NN classification results for breast cancer data set:"

else:

print "k-NN classification results for iris data set:"

print

print " Number of correct / wrong classified test records"

print "k | Euclidean dist | Manhattan dist | Cosine dist"

# run knn classifier for each k and distance function

for i in range(len(K)):

# classification result for each distance function

results = []

for j in range(len(dist\_fn)):

# predict the data test into class

pred\_class = knn(K[i], dtrain, dtest, dtr\_label, dist\_fn[j])

# evaluate the predicted result

eval\_result = evaluate(pred\_class-true\_class)

# assign the evaluated result into classification result

results.append(eval\_result[0])

results.append(eval\_result[1])

# print the classification result into the screen

print K[i], " | ", results[0], "/", results[1], \

" | ", results[2], "/", results[3], \

" | ", results[4], "/", results[5]

results = []

print

**# retrieve**

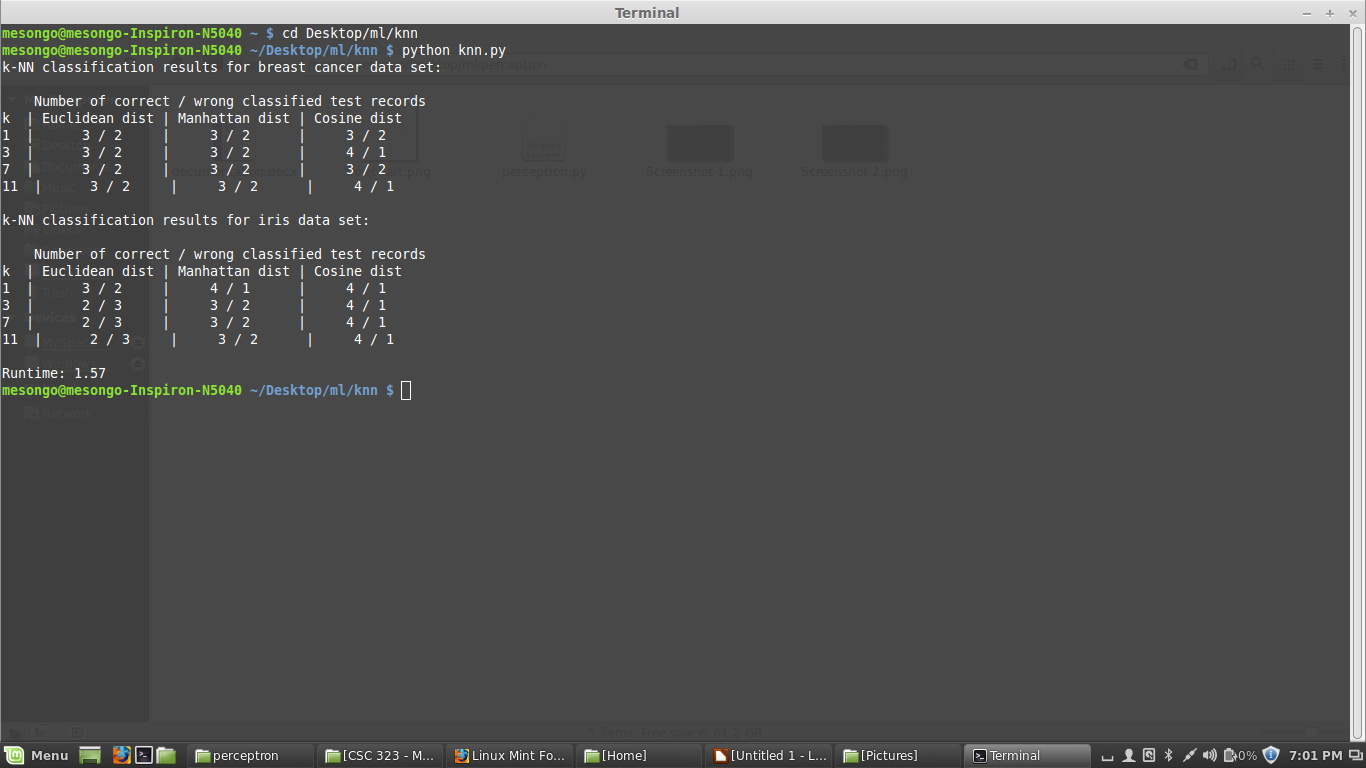
run\_time = time.clock() - start

print "Runtime:", run\_time

if \_\_name\_\_ == '\_\_main\_\_':

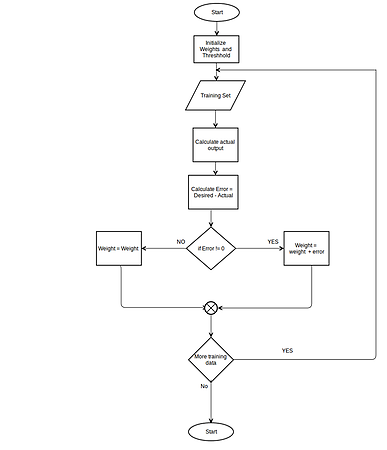
main()

**Screen shot**



# PERCEPTRON

**Flowchart**



**Algorithm and implementation**

threshold = 0.5

learning\_rate = 0.1

weights = [0, 0, 0]

training\_set = [((1, 0, 0), 1), ((1, 0, 1), 1), ((1, 1, 0), 1), ((1, 1, 1), 0)]

def dot\_product(values, weights):

return sum(value \* weight for value, weight in zip(values, weights))

while True:

#print the seperation line

print('-' \* 60)

error\_count = 0

for input\_vector, desired\_output in training\_set:

#printing the initial weights

print("Initial Weights: {}". format(weights))

result = dot\_product(input\_vector, weights) > threshold

#calculating error

error = desired\_output - result

print("Error: {}". format(error))

#if there is an error adjust the weights

if error != 0:

error\_count += 1

#for every input in the input vector calculate the new weights

for index, value in enumerate(input\_vector):

weights[index] += learning\_rate \* error \* value

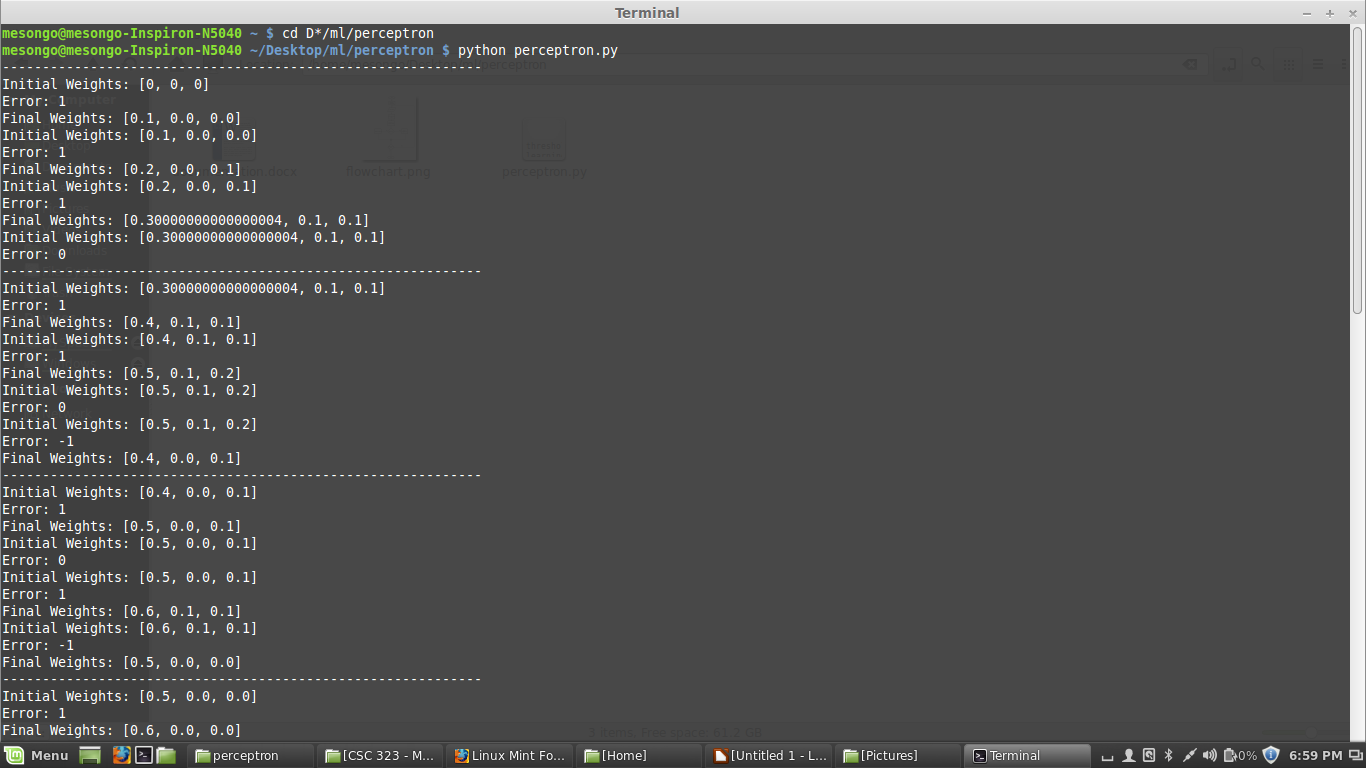
print("Final Weights: {}". format(weights))

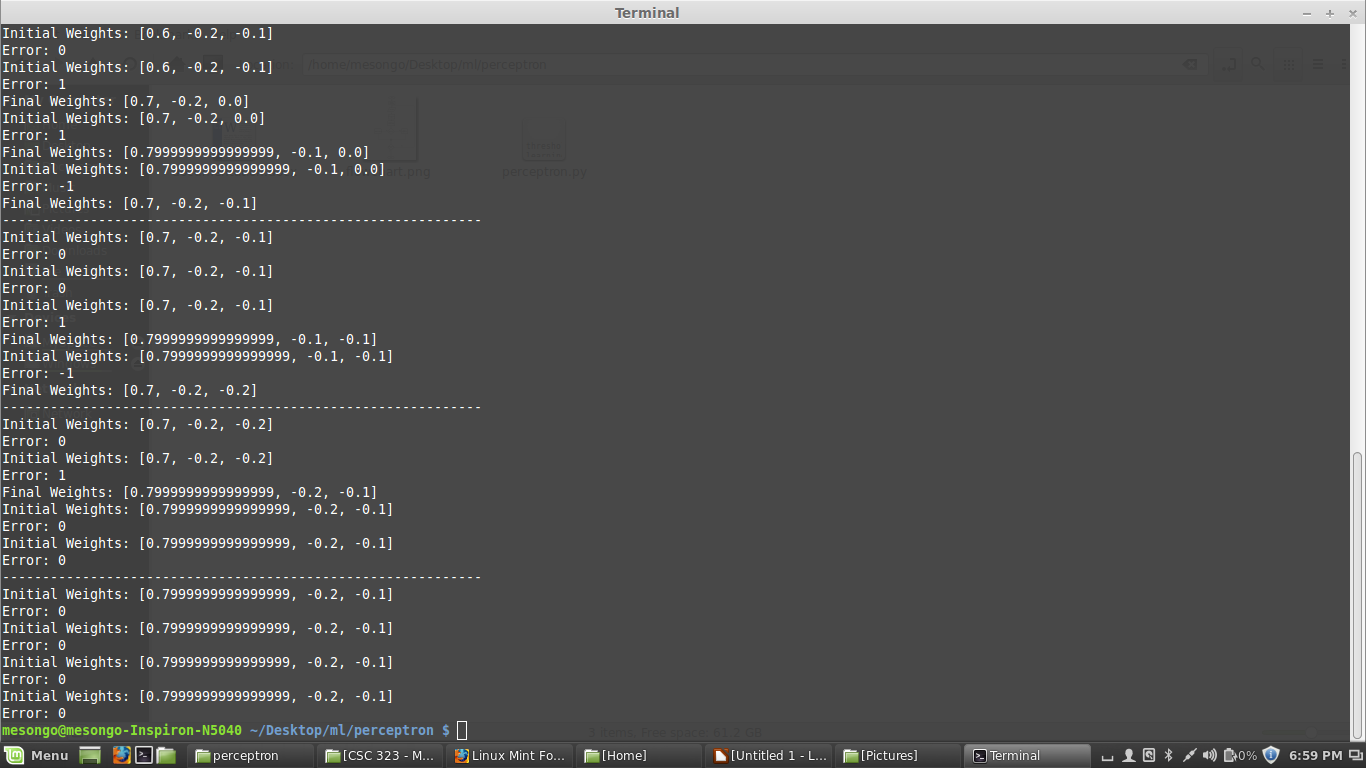
#if there was no input with an error terminate the loop

if error\_count == 0:

break

**Screen shots**

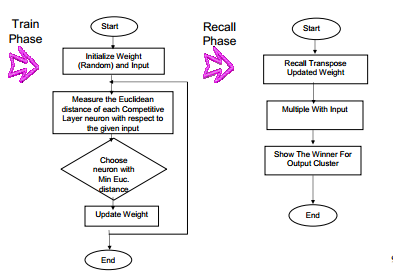




# 

# SELF ORGANISING MAPS

**Flow chart**



**Pseudo code**

Initialize weights  
 For 0 to N number of training epochs

Select a sample from the input data set  
 Find the "winning" neuron for the sample input  
 Adjust the weights of nearby neurons

End for loop

The stages of the SOM algorithm that achieves this can be summarised as follows:

1. Initialization – Choose random values for the initial weight vectors wj

2. Sampling – Draw a sample training input vector x from the input space.

3. Matching – Find the winning neuron I(x) that has weight vector closest to the input vector

4. Updating – Apply the weight update equation

5. Continuation – keep returning to step 2 until the feature map stops changing.

**Algorithm and implementation**

**Training input**

[[1, 1, 0, 0],

[0, 0, 0, 1],

[1, 0, 0, 0],

[0, 0, 1, 1]]

**Weight initialisation**

self.w = [[0.2, 0.6, 0.5, 0.9],

[0.8, 0.4, 0.7, 0.3]]

**Calculate the distance of input vector and weight**

for i in range(self.maxClusters):

for j in range(self.mVectors):

self.mD[i] += math.pow((self.w[i][j] - trainingTests[vectorNumber][j]), 2)

**See which is smaller, mD(0) or mD(1)?**

dMin = 1 if self.mD[0] > self.mD[1] else 0

**Update the weights on the winning unit.**

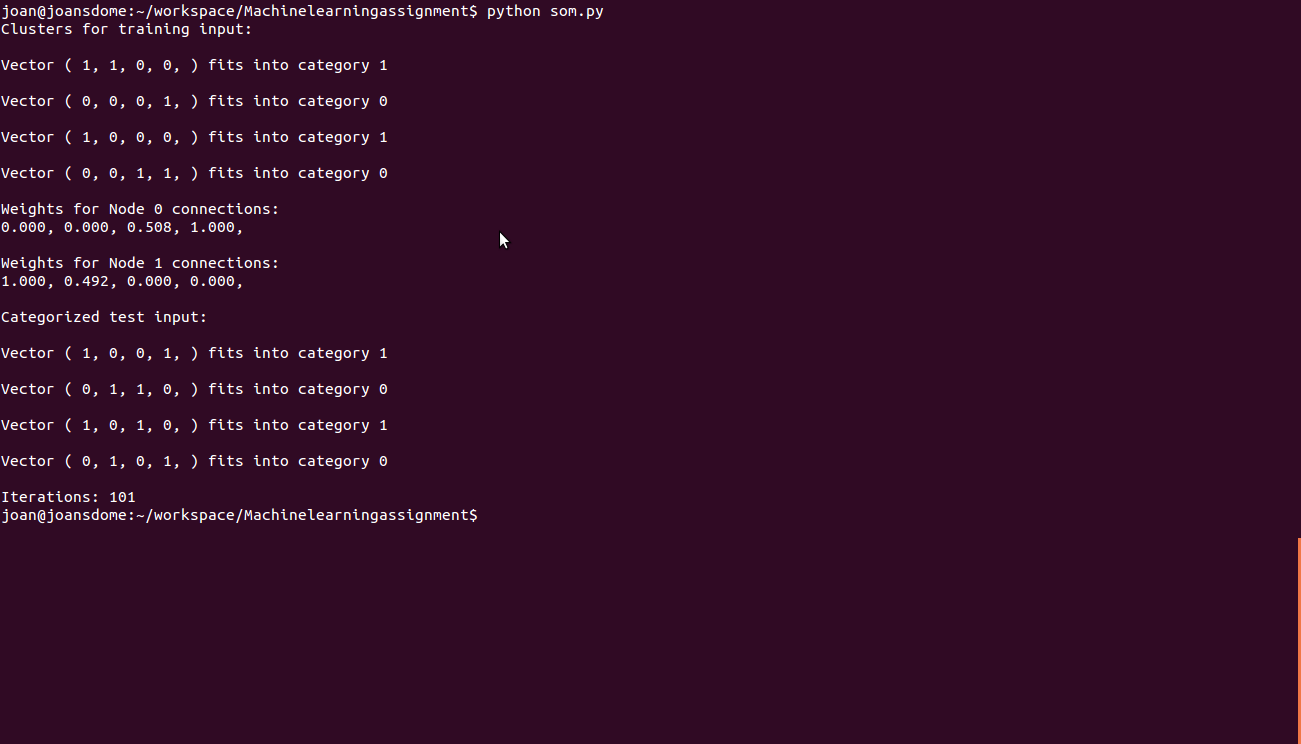
for j in range(self.mVectors):

self.w[dMin][j] = self.w[dMin][j] + (self.mAlpha \* (patterns[i][j] - self.w[dMin][j]))

**Reduce the learning rate.**

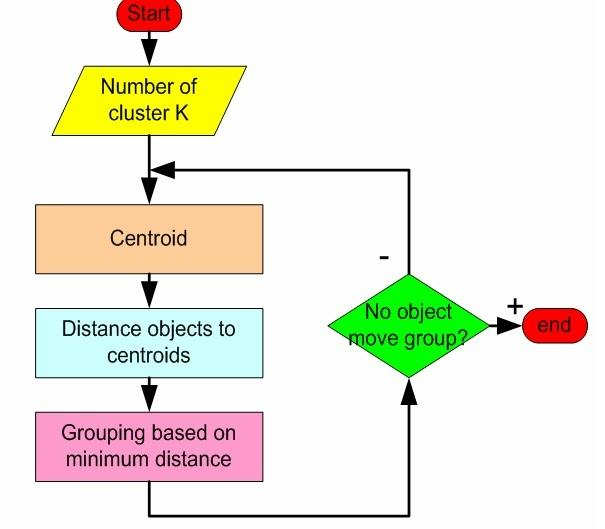
self.mAlpha = self.decayRate \* self.mAlpha

**SCREEN SHOT**



# K MEANS

**FLOW CHART**



**Pseudo code**

Begin

Do

1. Assign query to random clusters
2. Calculate the mean of the clusters using Euclidean distance
3. Calculate centroids and form new clusters
4. Calculate distance of the query from the new centroids
5. Assign query to cluster with least distance
6. If(query still in initial cluster)

Stop

Else

Repeat from step 2

End

**Algorithm and implementation**

**Centroid coordinates**

def calculateCentroid(self):

reduce\_coord = lambda i:reduce(lambda x,p : x + p.coords[i],self.points,0.0)

centroid\_coords = [reduce\_coord(i)/len(self.points) for i in range(self.n)]

return Point(centroid\_coords)

**Determine distance**

def getDistance(a, b):

if a.n != b.n: raise Exception("ILLEGAL: non comparable points")

ret = reduce(lambda x,y: x + pow((a.coords[y]-b.coords[y]), 2),range(a.n),0.0)

return math.sqrt(ret)

**Cluster creation**

for i in range(len(clusters)):

shift = clusters[i].update(lists[i])

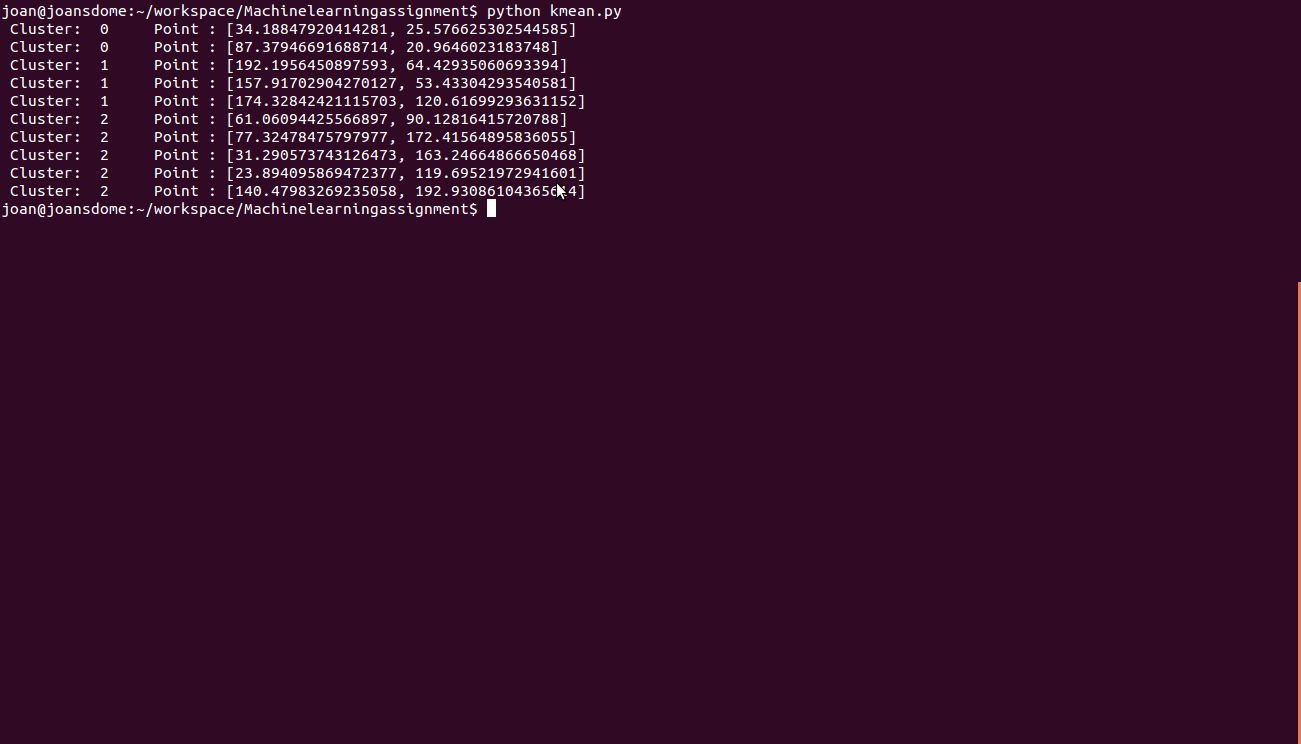
biggest\_shift = max(biggest\_shift, shift)

if biggest\_shift < cutoff:

break

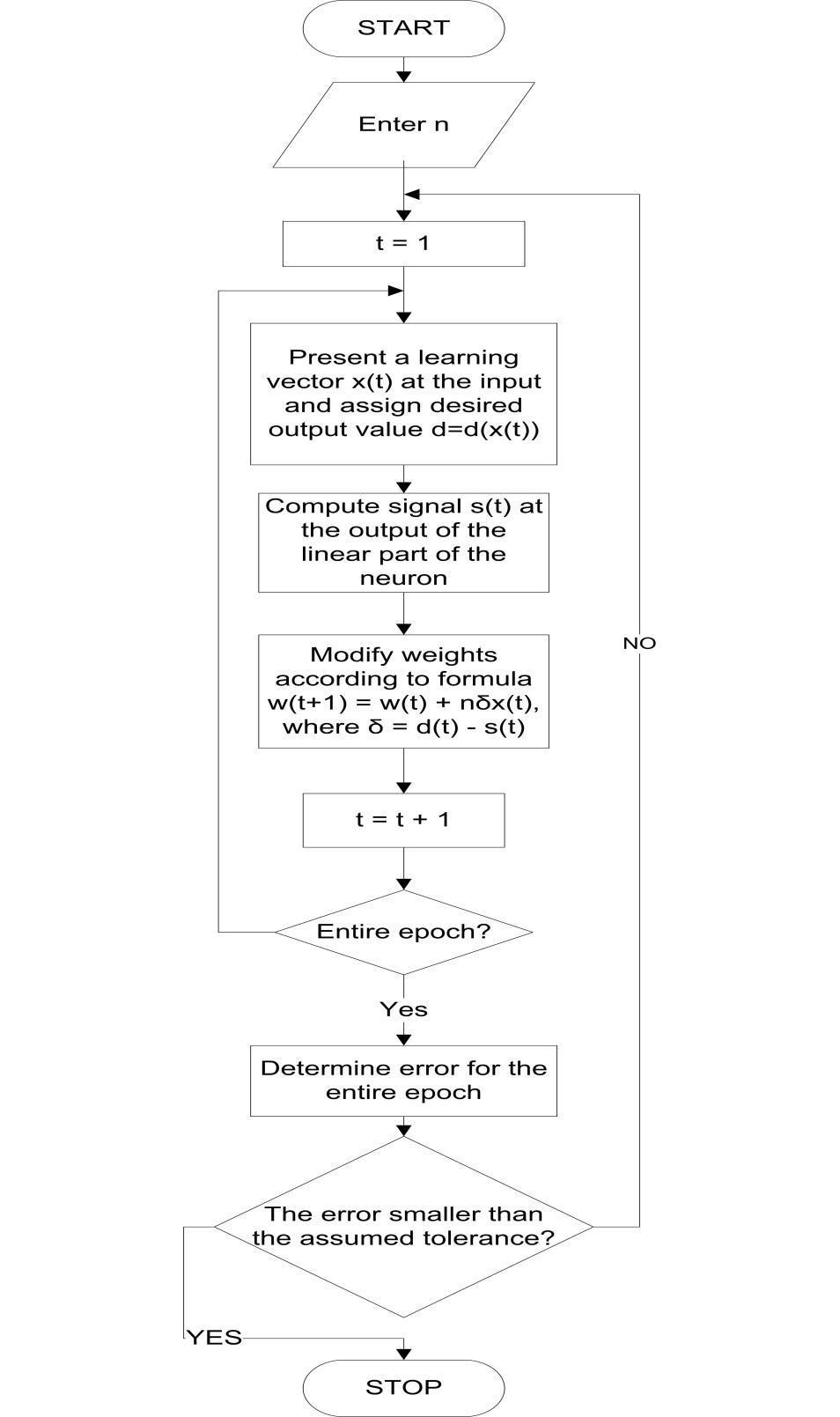
return clusters

**Screenshot**



# ADALINE

**Flowchart**



**Pseudo code**

1. Initialise weights (W1...Wn) and threshold (W0)
2. Set all weights and threshold to small random bipolar random values
3. Present new input and desired output
   1. present input vector x1,x2,...xn along with the desired output d(t)
4. Calculate the actual output[y(t)]
   1. y(t) = Fh[Σwi(t)\*xi(t) ]
5. Adapt weights

Wit+1()=Wi(t)+ηd(t)−{i=1n}∑Wi(t)Xi(t)Xi(t)

0 < i < n and η is the learning rate and usually is a small number ranging from 0 to 1.

6.Repeat step 2 to 4

Repeat until the desired outputs and the actual network outputs are all equal for all the input vectors of the training set.

**Algorithm and implementation**

This program decides if a given point is above the line y = (2x-1)/4

The adaline has two input nodes, receiving the coordinates of a point that it has to decide on.

**# defining a set of 500 patterns, type float, with 2 inputs and 1 output**

pat = Pattern("float", 2, 1, 500)

**# adaline network with two input nodes**

net = AdalineNet(2)

for i in range(500):

x = random.random()\*2.0 - 1.0 # adaline network need inputs -1 =< x =< 1

y = random.random()\*2.0 - 1.0

f = lambda a: (2\*a - 1)/4

res = (y < f(x)) and 1.0 or -1.0 # if y < f(x), the net should eval to 1.0

pat.setInputs([x, y], i)

pat.setOutputs([res], i)

**# Train the network with it patterns**

while good < 500 and itrn < 1000:

good = 0

for i in range(500):

net.loadInput(pat.getIn(i))  **# set the input values**

if net.getValue() != pat.getOut(i)[0]:

net.train()  **# network produced an error, train it**

**#output result**

print "x = %s, y = %s, result should be: %s" % (x,y,res)

print "The network produced: %s" % net\_res

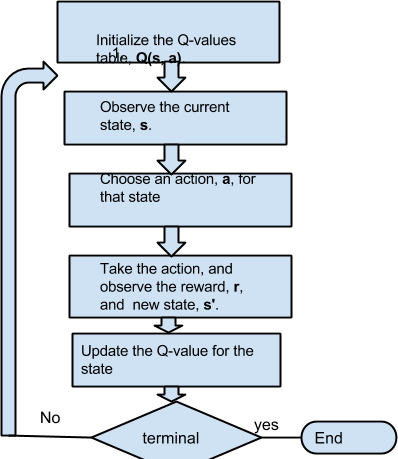
# 

# 

# 

# REINFORCEMENT LEARNING

**Flowchart**



**Pseudocode**

For each s, a, initialize table entry Q(s,a) <- 0

Observe current state s

Do forever:

Select an action a and execute it

Receive immediate reward r

Observe the new state s'

Update the table entry for Q(s, a) a

This procedural approach can be translated into steps as follows:

1. Initialize the Q-values table, **Q(s, a)**.
2. Observe the current state, **s**.
3. Choose an action, **a**, for that state
4. Take the action, and observe the reward, **r**, as well as the new state, **s'**.
5. Update the Q-value for the state using the observed reward and the maximum reward possible for the next state. The updating is done according to the formula and parameters described above.
6. Set the state to the new state, and repeat the process until a terminal state is reached.

**Algorithm and implementation**

**Initialize the Q-values table, Q(s, a).**

self.Q = [ [ initQ for a in range( noActions[s] ) ] for s in range( noStates ) ]

**Choose an action, a, for that state**

def explore( self ):

self.getAction = lambda st : self.exp.getAction( st, self.Q[st], self.maxQ[st], self.maxA[st], self.maxN[st] )

def exploit( self ):

self.getAction = lambda st : choice( self.maxA[st], self.maxN[st] )

**Update the Q-value for the state using the observed reward and the maximum reward**

if update:

V = self.maxQ[st\_]

self.Q[st][at] += alpha\*( rt + gamma\*V - self.Q[st][at] )

if updatenext == "always":

self.maxQ[st] = max( self.Q[st] )

self.V[st] = self.maxQ[st]

self.maxA[st] = [ a for a in range( self.A[st] ) if self.Q[st][a] == self.maxQ[st] ]

self.maxN[st] = len( self.maxA[st] )

if updatenext == "now":

for s in range( self.S ):

self.maxQ[s] = max( self.Q[s] )

self.V[s] = self.maxQ[s]

self.maxA[s] = [ a for a in range( self.A[s] ) if self.Q[s][a] == self.maxQ[s] ]

self.maxN[s] = len( self.maxA[s] )