# FIND-S:

import csv

# s find algorithm

with open('data.csv','r') as f:

reader = csv.reader(f)

dlist = list(reader)

h = [['0','0','0','0','0','0']]

print("Data input is:")

for l in dlist:

print(l)

print("Training Data:")

for i in dlist:

if i[-1] == "True":

print(i)

j = 0

for x in i:

if x != "True":

if x != h[0][j] and h[0][j] == '0':

h[0][j] = x

elif x != h[0][j] and h[0][j] != '0':

h[0][j] = '?'

else:

pass

j = j+1

print("Most specific hypothesis is:")

print(h)

Candidate Elimination:

import csv

import random

def g\_0(n):

return ("?",)\*n

def s\_0(n):

return ("0",)\*n

def more\_general(h1, h2):

more\_general\_parts = []

for x, y in zip(h1, h2):

mg = x == "?" or (x != "0" and (x == y or y == "0"))

more\_general\_parts.append(mg)

return all(more\_general\_parts)

def fulfills(example, hypothesis):

return more\_general(hypothesis, example)

def min\_generalization(h, x):

h\_new = list(h)

for i in range(len(h)):

if not fulfills(x[i:i+1], h[i:i+1]):

h\_new[i] = '?' if h[i] != "0" else x[i]

return [tuple(h\_new)]

def min\_specialization(h, domains, x):

results = []

for i in range(len(h)):

if h[i] == "?":

for val in domains[i]:

if x[i] != val:

h\_new = h[:i] + (val,) + h[i+1:]

results.append(h\_new)

elif h[i] != "0":

h\_new = h[:i] + ("0",) + h[i+1:]

results.append(h\_new)

return results

with open("data.csv") as csv\_file:

examples = [tuple(line) for line in csv.reader(csv\_file)]

print(examples)

def get\_domains(examples):

d = [set() for i in examples[0]]

for x in examples:

for i,xi in enumerate(x):

d[i].add(xi)

print("domains =",[list(sorted(x)) for x in d])

return [list(sorted(x)) for x in d]

def candidate\_elimination(examples):

"""

:rtype:object

"""

domains = get\_domains(examples)[:-1]

G = set([g\_0(len(domains))])

S = set([s\_0(len(domains))])

i = 0

print("\n G[{0}]:".format(i), G)

print("\n S[{0}]:".format(i), S)

for xcx in examples:

i = i + 1

x, cx = xcx[:-1], xcx[-1] # split data into attributes and decisions

if cx == 'True': # x is a positive example

G = {g for g in G if fulfills(x, g)}

S = generalize\_S(x, G, S)

else: # x is a negative example

S = {s for s in S if not fulfills(x,s)}

G = specialize\_G(x,domains,G,S)

print("\n G[{0}]:".format(i),G)

print("\n S[{0}]:".format(i),S)

return

def generalize\_S(x,G,S):

S\_prev = list(S)

for s in S\_prev:

#if s not in S:

# continue

if not fulfills(x,s):

S.remove(s)

Splus = min\_generalization(s, x)

# keep only generalizations that have a counterpart in G

S.update([h for h in Splus if any([more\_general(g,h) for g in G])])

# remove hypothesis less specific than any other in S

S.difference\_update([h for h in S if any([more\_general(h,h1) for h1 in S if h != h1])])

return S

def specialize\_G(x,domains,G,S):

G\_prev = list(G)

for g in G\_prev:

# if g not in G:

# continue

if fulfills(x,g):

G.remove(g)

Gminus = min\_specialization(g,domains,x)

# keep only specializations that have a counterpart in S

G.update([h for h in Gminus if any([more\_general(h,s) for s in S])])

# remove hypothesis less general than any other in G

G.difference\_update([h for h in G if any([more\_general(g1,h) for g1 in G if h != g1])])

return G

# run the algorithm

candidate\_elimination(examples)

KMEANS:

import csv

import math

import copy

k=2

class cluster:

def \_\_init\_\_(self,cluster\_head\_data):

self.head = cluster\_head\_data

self.l = []

def display\_head(self):

print(self.head)

def add\_ele\_cluster(self,data):

self.l.append(data)

def display\_ele(self):

print('list contains',self.l)

def compare\_the\_values(first,second):

x = float(first[0])

x1 = float(second[0])

val = math.sqrt(math.pow(math.fabs(x-x1),2)+math.pow(math.fabs(x-x1),2))

return val

def compare\_the\_nearest\_cluster(cluster,data):

# initialize the nearest cluster to 0

dist\_measure = None

nearest = 0

for i in range(len(cluster)):

dist = compare\_the\_values(cluster[i].head,data)

if dist\_measure is None:

dist\_measure = dist

nearest = i

if dist < dist\_measure:

dist\_measure = dist

nearest = i

return nearest

def recal\_head(cluster):

for i in range(len(cluster)):

l1 = cluster[i].l

xval = 0.0

yval = 0.0

for j in l1:

xval += float(j[0])

yval += float(j[1])

xavg = xval/len(l1)

yavg = yval/len(l1)

avg1 = []

avg1.append(xavg)

avg1.append(yavg)

cluster[i].head = avg1

# read the contents of the csv file

with open("cluster.csv") as csvfile:

spamreader = csv.reader(csvfile,delimiter=",")

#insert elements into the list

db = []

for row in spamreader:

db.append(row)

# creating individual cluster heads

#display elements of the list

print("DB entries")

print(db)

#init cluster

c = []

for i in range(k):

new\_clust = cluster(db[i])

c.append(new\_clust)

print("Initial cluster head values")

for i in range(k):

print("-------cluster",i,"-------")

c[i].display\_head()

error\_rate = 1

# iteration and including element in cluster

while error\_rate > 0:

prevc = copy.deepcopy(c)

for ele in db:

r = compare\_the\_nearest\_cluster(c,ele)

c[r].add\_ele\_cluster(ele)

#display all elements

for clust in c:

clust.display\_ele()

#recalculate head

recal\_head(c)

for i in range(k):

print("--------cluster",i,"-----------")

c[i].display\_head()

# remove the ele of cluster head for next iteration

for i in range(k):

c[i].l = []

# calculate error

error\_rate = 0

for i in range(k):

if c[i].head != prevc[i].head:

error\_rate += 1

#final cluster ele

KNN:

import csv

import random

import math

import operator

def load\_dataset(filename, split, training\_set=[],test\_set=[]):

with open(filename) as csvfile:

lines = csv.reader(csvfile)

dataset = list(lines)

for x in range(len(dataset)-1):

for y in range(4):

dataset[x][y] = float(dataset[x][y])

if random.random() < split:

training\_set.append(dataset[x])

else:

test\_set.append(dataset[x])

def euclidean\_distance(instance1,instance2,length):

distance = 0

for x in range(length):

distance += pow((instance1[x] - instance2[x]),2)

return math.sqrt(distance)

def get\_neighbors(training\_set,test\_instance,k):

distances = []

length = len(test\_instance)-1

for x in range(len(training\_set)):

dist = euclidean\_distance(test\_instance,training\_set[x],length)

distances.append((training\_set[x], dist))

distances.sort(key=operator.itemgetter(1))

neighbors = []

for x in range(k):

neighbors.append(distances[x][0])

return neighbors

def get\_response(neighbors):

class\_votes = {}

for x in range(len(neighbors)):

response = neighbors[x][-1]

if response in class\_votes:

class\_votes[response] += 1

else:

class\_votes[response] = 1

sorted\_votes = sorted(class\_votes.items(),key=operator.itemgetter(1),reverse=True)

return sorted\_votes[0][0]

def get\_accuracy(test\_set,predictions):

correct = 0

for x in range(len(test\_set)):

if test\_set[x][-1] == predictions[x]:

correct += 1

return (correct/float((len(test\_set))) \* 100.0)

def main():

# prepare data

training\_set = []

test\_set = []

split = 0.67

load\_dataset('KNN-input.csv',split,training\_set,test\_set)

print('\n Number of training data:'+ (repr(len(training\_set))))

print('\n Number of test data:' + (repr(len(test\_set))))

# generate predictions

predictions = []

k = 3

print("\n the predictions are:")

for x in range(len(test\_set)):

neighbors = get\_neighbors(training\_set,test\_set[x],k)

result = get\_response(neighbors)

predictions.append(result)

print('predicted ='+repr(result)+', actual='+repr(test\_set[x][-1]))

accuracy = get\_accuracy(test\_set,predictions)

print('\n The accuracy is:'+repr(accuracy) + '%')

main()