CuteSVM

By Yen-Ting Chen

Project research – Yen-Ting Chen

Implementation – Yen-Ting Chen

Report – Yen-Ting Chen

Project Goal:

Support Vector Machines (SVM) is the classification algorithm that is a type of supervised learning method that is often used for tasks such as pattern recognition. In my program, I used an implementation of SVM called the SMO (Sequential Minimal Optimization) to construct my classifier that has the potential to use various types of kernels and classify data in multiple dimensions.

The current implementation should allow the user to input two groups of data in real time, run SVM over the dataset, and output the classified results and a separating hyperplane that separates the two classes of data.

Techniques:

SMO divides up the large Quadratic Programming (QP) problem of SVM into many small parts; each part is itself a QP problem on the optimization of two Lagrange Multipliers. Since each small QP is solved analytically, it avoids numerical QP optimization as an inner loop and gives SMO a faster training time. Also the amount of memory required for SMO linear to the amount of training examples.

The overall QP problem to train an SVM is as following

(the big W as the objective function):



The point is an optimal point iff the KKT conditions:



are satisfied and Q(i,j):

yi \*yj\*k(xi,xj)

is positive semi-definite.

SMO divides the overall QP problem into parts of small QP problems. In each steps there are Three components to the SMO algorithm:

1. An analytic method to solve for the two Lagrange multipliers
2. A heuristic for choosing which multipliers to optimize
3. A method for computing b(the threshold value)

For the first task, SMO first computes the constraint on the two multipliers; the constraint bounds the Lagrange Multipliers in a box and the linear equality constraint places the Multipliers on a single diagonal line.

We first compute the 2nd multiplier a2 then computes the ends of the diagonal line in terms of a2:

If y1 != y2:



If y1 = y2:



Use the second derivative of objective function to find the maximum location of the objective Function:



Demo Instruction:

The GUI gives us a nice graphic canvas for real time inputting of 2D data points. The canvas will respond to the sequential mouse clicks by switching between blue and red; the colors blue and red correspond to the positive and negative class label relatively. After all data points has been drawn on the canvas and pressed done, the GUI uses these data points to train the SVM and outputs results on the GUI and python’s command line. At the moment it has a semi working classification capability and successfully applies soft margin (correctly classifying the outliers in the dataset) for the SVM.

Lesson earned:

As a first timer implementing a classic and slightly advanced AI technique, I have found SVM to be quite complicated in its steps, even though the main problem it tries to tackle is straight forward. The implementation of the entire algorithm however, is even more challenging. Since I’m learning the concepts of SVM on the fly while trying to make a fast implementation, I didn’t make minor tests before drafting out the whole program; and this gave me a major headache when I’m debugging.

While trying to understand the project, I found the online forums to be extremely helpful. By looking at other implementations of SVM and problems that other implementers faced, I was able to avoid creating things from scratch and stepping into easy traps.

Future development:

Currently my project has an incomplete graphical representation of SVM. My program is supposed to give a correct hyperplane to separate the classified groups. It should also be able to take in more complicated datasets such as an labeled image or a whole article.

Codes:

GUI.py

#!/usr/bin/env python

from tkinter import\*

from final import\*

radius = 2

isRed = TRUE

data\_pos = []

data\_neg = []

def process():

global canvas

global radius

f = open("coordinates.txt",'w')

for xy in data\_pos:

f.write(str(xy[0])+' '+str(xy[1])+' --> 0\n')

for xy in data\_neg:

f.write(str(xy[0])+' '+str(xy[1])+' --> 1\n')

f.close()

svm\_main(data\_pos, data\_neg)

## h = hyperplane()

## for xy in h:

## canvas.create\_oval(xy[0]-radius,xy[1]-radius,xy[0]+radius,xy[1]+radius, fill='black')

def click(event):

global radius

global isRed

global data

if isRed:

canvas.create\_oval(event.x-radius, event.y-radius,event.x+radius, event.y+radius, fill='red')

isRed = FALSE

data\_pos.append([event.x,event.y])

else:

canvas.create\_oval(event.x-radius, event.y-radius,event.x+radius, event.y+radius, fill='blue')

isRed = TRUE

data\_neg.append([event.x,event.y])

root = Tk()

button\_1 = Button(root,text = "Done", font=('Sims',12), command = process)

button\_2 = Button(root,text = "Quit", font=('Sims',12), command = root.destroy)

canvas = Canvas(root, width=1000, height=400)

canvas.grid(column=0, row=0, sticky=('N', 'W', 'E', 'S'))

canvas.bind("<Button-1>", click)

button\_2.pack()

button\_1.pack()

canvas.pack()

root.mainloop()

final.py

import math

import random

K\_matrix = []

omegaX = []

X = []

y = []

E = []

a = []

b = 0

n = 0

C = 1 #constant, weight for loss function. KKT boundary.

Tol = 0.0001 #tolerance.

rho = 100 #kernel width

epsilon = pow(10,-3) #KKT condition is fulfilled within epsilon.

def classify(omegaX,b):

global n

results = []

for i in range(0,n):

f = omegaX[i]-b

results.append(f)

return results

#Storing a Kernel matrix lookup table for fast future access.

#n: how many data examples are there.

def create\_K\_matrix(n):

global K\_matrix

global X

K\_matrix = [[1]\*n]\*n

for i in range(0,n):

for j in range(0,n):

K\_matrix[i][j] = kernel(X[i],X[j])

def dot\_product(x1,x2):

if len(x1) == len(x2):

dot = 0

for i in range(0,len(x1)):

dot+=x1[i]\*x2[i]

return dot

else:

"There's an error when doing dot product of: "+ x1 +" and "+ x2

def init\_alphas(n):

global a

global C

for i in range(0,n):

a.append(random.random()\*C) #random initialization of a, a in [0,C]

def init\_error\_cache(n):

global E

global X

global y

global a

global b

global K\_matrix

omegaX = [0]\*n

E = []

for k in range(0,n):

for i in range(0,n):

omegaX[k] += a[i]\*y[i]\*K\_matrix[i][k]

for i in range(0,n):

f = omegaX[i] - b

error = f - y[i]

E.append(error)

def kernel(X1,X2):

global rho

norm\_square = dot\_product(X1,X1) - 2\*dot\_product(X1,X2) + dot\_product(X2,X2)

return math.exp(-0.5 \* norm\_square/pow(rho,2))

def update\_obj(n):

global a

global y

global K\_matrix

tempSum = 0

W = 0

for i in range(0,n):

for j in range(0,n):

tempSum= tempSum + y[i]\*y[j]\*a[i]\*a[j]\*K\_matrix[i][j]

W= W + a[i]

W = W - 0.5 \* tempSum

return W

def update\_omegaX():

global n

global X

global y

global a

global K\_matrix

global omegaX

omegaX = [0]\*n

for k in range(0,n):

for i in range(0,n):

omegaX[k] += a[i]\*y[i]\*K\_matrix[i][k]

def takeStep(i1, i2):

global a

global b

global y

global epsilon

global K\_matrix

global n

global omegaX

if i1 == i2:

return 0

a1old = a[i1]; a2old = a[i2]

a1 = 0; a2 = 0

y1 = y[i1] ; y2 = y[i2]

bold = b

update\_omegaX()

f1 = omegaX[i1] -b ; f2 = omegaX[i2] -b

E1 = f1 - y1 ; E2 = f2 - y2

s = y1\*y2

L = 0 ; H=0;

if s == -1:

L = max(0, a2old - a1old)

H = min(C, C + a2old - a1old)

else:

L = max(0, a1old + a2old - C)

H = min(C, a1old + a2old)

if L==H:

return 0

k11 = K\_matrix[i1][i1]

k12 = K\_matrix[i1][i2]

k22 = K\_matrix[i2][i2]

eta = 2 \* k12 - k11 - k22

if eta < 0:

a2 = a2old - y2\*(E1-E2)/eta

if a2 < L:

a2 = L

if a2 > H:

a2 = H

else:

c1 = eta/2

c2 = y2 \* (E1-E2) - eta \* a2old

Lobj = c1 \* L \* L + c2 \* L

Hobj = c1 \* H \* H + c2 \* H

if Lobj > Hobj + epsilon:

a2 = L

elif Lobj < Hobj - epsilon:

a2 = H

else:

a2 = a2old

if a2 < 1e-8:

a2 = 0

elif a2 > C-1e-8:

a2 = C

if abs(a2 - a2old) < epsilon \* (a2 + a2old + epsilon):

return 0

a1 = a1old + s\*(a2old - a2)

b1 = E1 + y1 \* (a1 - a1old) \* k11 + y2 \* (a2 - a2old) \* k12 + bold

b2 = E1 + y1 \* (a1 - a1old) \* k12 + y2 \* (a2 - a2old) \* k22 + bold

if a1 > 0 and a1 < C:

b = b2

else:

if a2 > 0 and a2 < C:

b = b1

else:

b = (b1 + b2) / 2

#update error cache.

for k in range(0,n):

f = omegaX[k] - b

E[k] = f - y[k]

E[i1] = 0

E[i2] = 0

a[i1] = a1

a[i2] = a2

update\_omegaX()

return 1

def examineExample(i1):

global y

global C

global Tol

global n

global b

global omegaX

y1 = y[i1]

a1old = a[i1]

E1 = 0

#correctly iterating through the whole alphas array.

if a1old >0 and a1old<C:

E1 = E[i1]

else:

f = omegaX[i1] -b

E1 = f-y1

r1 = E1 \* y1

if (r1 < -Tol and a1old < C) or(r1 > Tol and a1old > 0):

a\_index = 0

a\_nonBound = []

#count the alphas that is non-zero or non-C.

for alpha in a:

if (alpha != 0 or alpha !=C): a\_nonBound.append(a\_index)

a\_index += 1

if len(a\_nonBound) > 0:

#using norm of error difference as step size is the second choice heuristic.

step\_max = 0

i2 = -1

for k in range(0,n):

E2 = E[k]

step\_size = abs(E1 - E2)

if step\_size > step\_max:

i2 = k

#print('2nd heuristic chose: '+str(i2))

if i2 >= 0:

if takeStep(i1,i2):

#print('after using 2nd heuristic to check')

return 1

#loop over all non-zero and non-C alpha, starting at random point

while len(a\_nonBound) > 0:

index = math.floor(random.random() \* len(a\_nonBound))

i2 = a\_nonBound[index]

a\_nonBound.remove(i2)

if takeStep(i1,i2):

return 1

#loop over all possible i2, starting at a random point

temp\_a = a[:]

while len(temp\_a) > 0:

i2 = math.floor(random.random() \* len(temp\_a))

temp\_a.remove(temp\_a[i2])

if takeStep(i1,i2):

return 1

return 0

##main

def svm\_main(data\_pos, data\_neg):

global X

global a

global n

global y

global C

#initializations

X = data\_pos + data\_neg

y = [1]\*len(data\_pos) + [-1]\*len(data\_neg)

n = len(X)

create\_K\_matrix(n)

init\_alphas(n)

init\_error\_cache(n)

b = 0

#save raw parameters before training

b\_raw = b

update\_omegaX()

omegaX\_raw = omegaX[:]

numChanged = 0

examineAll = 1

#first choice heuristic to choose i1.

while numChanged > 0 or examineAll:

numChanged = 0

if examineAll:

for i in range(0,n):

numChanged += examineExample(i)

#print('one run of all examples')

else:

a\_nonBoundI = []

for i in range(0,n):

if a[i] != 0 and a[i] != C:

a\_nonBoundI.append(i)

for i in a\_nonBoundI:

# alphas are not changing.

numChanged += examineExample(i)

#print('one round of non-bound examples')

if examineAll:

examineAll = 0

elif numChanged == 0:

examineAll = 1

## print(numChanged)

## print(a)

## print('W--->' + str(update\_obj(n)))

#results printing

print('error after training:')

print(E)

print("--------------------------------------------------")

print('original class labels:')

print(y)

print('trained results:')

print(classify(omegaX,b))

##d1 = [[500\*random.random()]] \* 6

##d2 = [[100\*random.random()]] \* 10

##svm\_main(d1,d2)