#include <iostream>

#include <cstdio>

#include <algorithm>

#include <cmath>

using std::sort;

using std::fabs;

const int MAX\_DIMENSION = 2;//数据维度

const int MAX\_SAMPLES = 4;//数据个数

double x[MAX\_SAMPLES][MAX\_DIMENSION];//数据集

double y[MAX\_SAMPLES];//标签

double alpha[MAX\_SAMPLES];//拉格朗日系数

double w[MAX\_DIMENSION];//系数w

double b;//系数b

double c;//松弛量(假设是k，等式左边每个系数都除k，c还是1)

double eps = 1e-6;//误差

struct \_E{

double val;//g - y,用于更新alpha

int index;//序号

}E[MAX\_SAMPLES];

bool cmp(const \_E & a, const \_E & b)//sort排序时用的比较函数

{

return a.val < b.val;

}

int num\_dimension;

int num\_samples;

double max(double a,double b)//取最大

{

return a>b?a:b;

}

double min(double a,double b)//取最小

{

return a>b?b:a;

}

double kernal(double x1[], double x2[], double dimension)//求内积

{

double ans = 0 ;

for(int i = 0 ; i < dimension; i++)

{

ans += x1[i]\*x2[i];

}

return ans;

}

double target\_function()

{

//用新的alpha1,alpha2，带入W(a)里

//W(a)=1/2\*K11\*a1\*a1 + 1/2\*K22\*a2\*a2 + g1 + g2 + ans + W(a3~an)

//g1,g2是指带入下面的g函数，ans就是本函数的ans

double ans = 0;

for(int i = 0 ; i < num\_samples; i++)

{

for(int j = 0 ; j < num\_samples; j++)

{

ans += alpha[i]\*alpha[j]\*y[i]\*y[j]\*kernal(x[i],x[j],num\_dimension);

}

}

for(int i = 0 ; i < num\_samples; i++)

{

ans -= alpha[i];

}

return ans;

}

double g(double \_x[], int dimension)

{

//计算W(a)的一部分，与上面的target\_function函数,共同构成W(a)

//同时g也是alpha更新时的一部分，所以在这里单独列出

double ans = b;

for(int i = 0 ; i < num\_samples; i++)

{

ans += alpha[i]\*y[i]\*kernal(x[i],\_x,dimension);

}

return ans;

}

bool satisfy\_constrains(int i, int dimension)

{

//满足KKT条件就不用更新

//判断是否为满足KKT条件的以下三种情况，更新时实际上有三种点

if(alpha[i] == 0)//这一次已经分对了的点

{

if(y[i]\*g(x[i], dimension) >= 1)

return true;

else

return false;

}

else if( alpha[i] > 0 && alpha[i] < c)//边界上，支持向量

{

if(y[i] \* g(x[i], dimension) == 1)

return true;

else

return false;

}

else//alpha == C，边界之间的点

{

if(y[i] \* g(x[i], dimension) <= 1)

return true;

else

return false;

}

}

double calE(int i, int dimension)//计算E(更新alpha时推导出的，见alpha推导的函数)

{

return g(x[i], dimension) - y[i];

}

void calW()//根据W(a)计算w (带入新的alpha就行)

{

for(int i = 0 ; i < num\_dimension; i++)

{

w[i] = 0;

for(int j = 0 ; j < num\_samples; j++)

{

w[i] += alpha[j] \* y[j] \* x[j][i];

}

}

return ;

}

void calB()//利用最开始的公式计算b(带入新的alpha就行)

{

double ans = y[0];

for(int i = 0 ; i < num\_samples ; i++)

{

ans -= y[i]\*alpha[i]\*kernal(x[i], x[0], num\_dimension);

}

b = ans;

return;

}

void recalB(int alpha1index,int alpha2index, int dimension, double alpha1old, double alpha2old)

{

//根据Y(W^T \* X + b) - 1 == 0， 除了之前更新用的两个alpha，其余项实际上可被替换，替换后得下面两个公式

double alpha1new = alpha[alpha1index];

double alpha2new = alpha[alpha2index];

alpha[alpha1index] = alpha1old;

alpha[alpha2index] = alpha2old;

double e1 = calE(alpha1index, num\_dimension);

double e2 = calE(alpha2index, num\_dimension);

alpha[alpha1index] = alpha1new;

alpha[alpha2index] = alpha2new;

//b1 = 旧的b1 - E1 - y1(新的a1 - 旧的a1)\*K(x1, x1) - y2(新的a2 - 旧的a2)\*K(x1, x2)

//b2 = 旧的b2 - E2 - y1(新的a1 - 旧的a1)\*K(x1, x2) - y2(新的a2 - 旧的a2)\*K(x2, x2)

double b1new = -e1 - y[alpha1index]\*kernal(x[alpha1index], x[alpha1index], dimension)\*(alpha1new - alpha1old);

b1new -= y[alpha2index]\*kernal(x[alpha2index], x[alpha1index], dimension)\*(alpha2new - alpha2old) + b;

double b2new = -e2 - y[alpha1index]\*kernal(x[alpha1index], x[alpha2index], dimension)\*(alpha1new - alpha1old);

b1new -= y[alpha2index]\*kernal(x[alpha2index], x[alpha2index], dimension)\*(alpha2new - alpha2old) + b;

//更新出两个b，每次取平均值

b = (b1new + b2new)/2;

}

bool optimizehelp(int alpha1index,int alpha2index)

{

//由于只挑出两个且y==1或y==-1，根据约束：alpha[i]\*y[i]==alpha[j]\*y[j]==eps

double alpha1new = alpha[alpha1index];

double alpha2new = alpha[alpha2index];

double alpha1old = alpha[alpha1index];

double alpha2old = alpha[alpha2index];

double H,L;

//新的alpha的范围

if(fabs(y[alpha1index] - y[alpha2index]) > eps)//同类

{

L = max(0, alpha2old - alpha1old);

H = min(c, c + alpha2old - alpha1old);

}

else//y[i]!=y[j]，不同类

{

L = max(0, alpha2old + alpha1old - c);

H = min(c, alpha2old + alpha1old);

}

//求新的alpha

//把W(a)展开，把包含alpha1和alpha2的式子拿出来，其余项可不管

//根据alpha[i]\*y[i]==alpha[j]\*y[j]等式，把alpha2换成alpha1，整个W(a)现在就只剩alpha1了，这时再求导

double lena = kernal(x[alpha1index], x[alpha1index], num\_dimension) + kernal(x[alpha2index], x[alpha2index], num\_dimension) - 2\*kernal(x[alpha1index], x[alpha2index], num\_dimension);

//求导后，可推导出下一行的公式

alpha2new = alpha2old + y[alpha2index]\*(calE(alpha1index, num\_dimension) - calE(alpha2index, num\_dimension))/lena;

//如果超出L~H就等于L或H

if(alpha2new > H)

{

alpha2new = H;

}

else if( alpha2new < L)

{

alpha2new = L;

}

alpha1new = alpha1old + y[alpha1index]\*y[alpha2index]\*(alpha2old - alpha2new);

double energyold = target\_function();

alpha[alpha1index] = alpha1new;

alpha[alpha2index] = alpha2new;

double gap = 0.001;

//求更新的b

recalB(alpha1index, alpha2index, num\_dimension, alpha1old, alpha2old);

return true;

}

bool optimize()//更新

{

int alpha1index = -1;

int alpha2index = -1;

double alpha2new = 0;

double alpha1new = 0;

//构造E = g - y

for(int i = 0 ; i < num\_samples; i++)

{

E[i].val = calE(i, num\_dimension);

E[i].index = i;

}

//每次从不满足KKT条件的多个alpha中挑出两个进行调整

//对于每一个alpha1

for(int i = 0 ; i < num\_samples; i++)

{

alpha1new = alpha[i];

//如果是边界之间的点

if(alpha1new > 0 && alpha1new < c)

{

//判断是否满足KKT

if(satisfy\_constrains(i, num\_dimension))

continue;

//排序，以便下面选alpha

sort(E, E+num\_samples, cmp);

//如果大于0且不是本身，则用最小值和当前alpha进行调整

if(alpha1new > 0)

{

if(E[0].index == i)

{

;

}

else

{

alpha1index = i;

alpha2index = E[0].index;

if(optimizehelp(alpha1index, alpha2index))

{

return true;

}

}

}

//如果小于0且不是本身，则用最大值和当前alpha进行调整

else

{

if(E[num\_samples-1].index == i)

{

;

}

else

{

alpha1index = i;

alpha2index = E[num\_samples-1].index;

if(optimizehelp(alpha1index, alpha2index))

{

return true;

}

}

}

//对于每个alpha2，分别于当前的alpha1一起进行KKT调整

for(int j = 0 ; j < num\_samples; j++)

{

alpha2new = alpha[j];

if(alpha2new > 0 && alpha2new < c)

{

alpha1index = i;

alpha2index = j;

if(optimizehelp(alpha1index , alpha2index))

{

return true;

}

}

}

for(int j = 0 ; j < num\_samples; j++)

{

alpha2new = alpha[j];

if(!(alpha2new > 0 && alpha2new < c))

{

alpha1index = i;

alpha2index = j;

if(optimizehelp(alpha1index , alpha2index))

{

return true;

}

}

}

}

}

//对于每一个alpha1

for(int i = 0 ; i < num\_samples; i++)

{

alpha1new = alpha[i];

//如果不是边界之间的点

if(!(alpha1new > 0 && alpha1new < c))

{

//判断是否满足KKT

if(satisfy\_constrains(i, num\_dimension))

continue;

//排序，以便下面选alpha

sort(E, E+num\_samples, cmp);

//如果大于0且不是本身，则用最小值和当前alpha进行调整

if(alpha1new > 0)

{

if(E[0].index == i)

{

;

}

else

{

alpha1index = i;

alpha2index = E[0].index;

if(optimizehelp(alpha1index, alpha2index))

{

return true;

}

}

}

//如果小于0且不是本身，则用最大值和当前alpha进行调整

else

{

if(E[num\_samples-1].index == i)

{

;

}

else

{

alpha1index = i;

alpha2index = E[num\_samples-1].index;

if(optimizehelp(alpha1index, alpha2index))

{

return true;

}

}

}

//对于每个alpha2，分别于当前的alpha1一起进行KKT调整

for(int j = 0 ; j < num\_samples; j++)

{

alpha2new = alpha[j];

if(alpha2new > 0 && alpha2new < c)

{

alpha1index = i;

alpha2index = j;

if(optimizehelp(alpha1index , alpha2index))

{

return true;

}

}

}

for(int j = 0 ; j < num\_samples; j++)

{

alpha2new = alpha[j];

if(!(alpha2new > 0 && alpha2new < c))

{

alpha1index = i;

alpha2index = j;

if(optimizehelp(alpha1index , alpha2index))

{

return true;

}

}

}

}

}

return false;

}

bool check()

{

//检查每一个点都是不是合格了

double sum = 0;

for(int i = 0 ; i < num\_samples; i++)

{

sum += alpha[i] \* y[i];

if(!(0 <= alpha[i] && alpha[i] <= c))//判断各个点都是不是支持向量

{

printf("alpha[%d]: %lf wrong\n", i, alpha[i]);

return false;

}

if(!satisfy\_constrains(i, num\_dimension))//判断每个点是否都已经正确分类

{

printf("alpha[%d] not satisfy constrains\n", i);

return false;

}

}

//判断是否在误差内满足约束alpha[i]\*y[i] == 0

if(fabs(sum) > eps)

{

printf("Sum = %lf\n", sum);

return false;

}

return true;

}

/\*

min 1/2\*||w||^2

s.t. (w[i]\*x[i] + b[i] - y[i]) >= 0;

\*/

/\*

step 1: cal alpha[]

step 2: cal w,b

\*/

/\*

min(para alpha) 1/2\*sum(i)sum(j)(alpha[i]\*alpha[j]\*y[i]\*y[j]\*x[i]\*x[j]) - sum(alpha[i])

s.t. sum(alpha[i] \* y[i]) = 0

C>= alpha[i] >= 0

\*/

int main()

{

scanf("%d%d", &num\_samples, &num\_dimension);//输入数据个数、数据维度

//输入数据

for(int i = 0 ; i < num\_samples; i++)

{

for(int j = 0; j < num\_dimension; j++)

{

scanf("%lf",&x[i][j]);//第i个数据的各个维度坐标

}

scanf("%lf",&y[i]);//第i个数据的标签

}

c = 1;

//拉格朗日系数初值附为0

for(int i = 0 ; i < num\_samples; i++)

{

alpha[i] = 0;

}

int count = 0;

//进行更新

while(optimize()){

calB(); //计算b

count++;

}

printf("count = %d\n ",count);

calW();//计算W

calB();//计算b

printf("y = ");

//返回结果

for(int i = 0 ; i < num\_dimension; i++)

{

printf("%lf \* x[%d] + ", w[i], i);

}

printf("%lf\n", b);

//检查结果

if(!check())

printf("Not satisfy KKT.\n");

else

printf("Satisfy KKT\n");

}

//3 2

//3 3 1

//4 3 1

//1 1 -1

/\*

min 1/2\*||w||^2

s.t. (w[i]\*x[i] + b[i] - y[i]) >= 0;

\*/

/\*

min(para alpha) 1/2\*sum(i)sum(j)(alpha[i]\*alpha[j]\*y[i]\*y[j]\*x[i]\*x[j]) - sum(alpha[i])

s.t. sum(alpha[i] \* y[i]) = 0

C>= alpha[i] >= 0

\*/