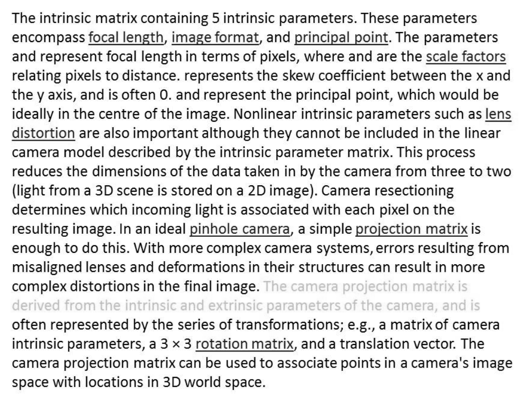
EEE 178 Test 2

Curtis Muntz

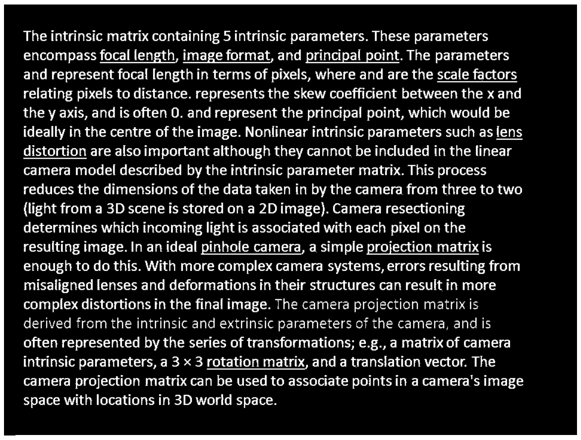
```
% pre-processing
clear all, close all, clc;
I1=imread('exam2letters.jpg');
I2=I1;
I1=rgb2gray(I1);
figure; imshow(I1);
```


The intrinsic matrix containing 5 intrinsic parameters. These parameters encompass focal length, image format, and principal point. The parameters and represent focal length in terms of pixels, where and are the scale factors relating pixels to distance. represents the skew coefficient between the x and the y axis, and is often 0. and represent the principal point, which would be ideally in the centre of the image. Nonlinear intrinsic parameters such as lens distortion are also important although they cannot be included in the linear camera model described by the intrinsic parameter matrix. This process reduces the dimensions of the data taken in by the camera from three to two (light from a 3D scene is stored on a 2D image). Camera resectioning determines which incoming light is associated with each pixel on the resulting image. In an ideal pinhole camera, a simple projection matrix is enough to do this. With more complex camera systems, errors resulting from misaligned lenses and deformations in their structures can result in more complex distortions in the final image. The camera projection matrix is derived from the intrinsic and extrinsic parameters of the camera, and is often represented by the series of transformations; e.g., a matrix of camera intrinsic parameters, a 3 × 3 rotation matrix, and a translation vector. The camera projection matrix can be used to associate points in a camera's image space with locations in 3D world space.

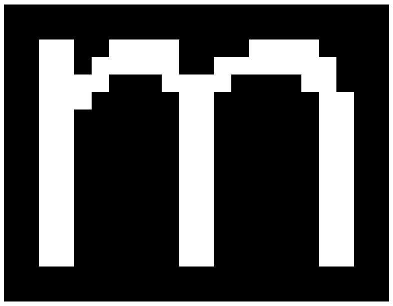
I. Problem 1

I was unable to use optimal thresholding due to the fact that the image contains yellow text, which falls below the Otsu's thresholding value. To compensate, a higher threshold was used. Unfortunately,using such a high threshold grabs a lot of noise from the image, and even combines some letter pairs into single objects. I worked around this issue by selecting one of the m's from the yellow text, and evaluating its regionprops. Because the m chosen was smaller in size than the majority of the other m's, I set a testing range for the regionprops. The values of the testing regionprops had to be within $90\% < m < 140\%$. Satisfying these conditions, there were 51 m's found in the image.

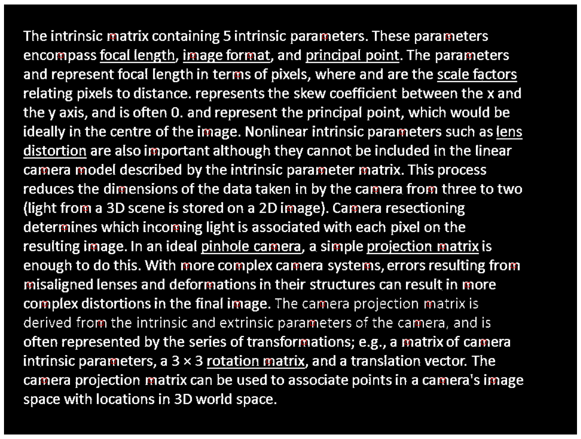
```
clc; close all;
clearvars -except I1 I2 I3; close all
%thresh=graythresh(I1);
thresh=0.83;
I = im2bw(I1,thresh); %convert to bw
I = imcomplement(I); %objects are white in
    matlab

%I=imerode(I,B);
%I=imclose(I,B);
imshow(I);

%example M
%bigM=imcrop(I, [173,47,(195-173),(63-47)]);
smallM=imcrop(I, [525,495,(546-525),(511-495)])
    ;
theM=smallM;
figure('name','the m'), imshow(theM), title('
    the m');
mSTATS=regionprops(theM,'all');

reconstructed = false(size(I));
stats = regionprops(I,'all');
figure('name', 'ms are highlighted');
imshow(I), title('ms are highlighted'); hold on
    ;

lowerBound=0.9;
upperBound=1.4;
for i=1:size(stats)
    if ( (stats(i).Solidity < (mSTATS.Solidity*
    upperBound)) ...
            && (stats(i).Solidity > (mSTATS.
    Solidity*lowerBound)) && ...
            (stats(i).FilledArea < (mSTATS.
    FilledArea*upperBound)) && ...
            (stats(i).FilledArea > (mSTATS.
    FilledArea*lowerBound)) && ...
            (stats(i).Area < (mSTATS.Area*
    upperBound)) && ...
            (stats(i).Area > (mSTATS.Area*
    lowerBound)) && ...
            (stats(i).EulerNumber == 1) && ...
            (stats(i).ConvexArea < (mSTATS.
    ConvexArea*upperBound)) && ...
            (stats(i).ConvexArea > (mSTATS.
    ConvexArea*lowerBound)) && ...
            (stats(i).Perimeter > (mSTATS.
    Perimeter*lowerBound)) && ...
            (stats(i).Perimeter < (mSTATS.
    Perimeter*upperBound)) && ...
            (stats(i).Extent > (mSTATS.Extent*
    lowerBound)) && ...
            (stats(i).Extent < (mSTATS.Extent*
    upperBound)) && ...
            (stats(i).MajorAxisLength < (mSTATS
    .MajorAxisLength*upperBound)) && ...
            (stats(i).MajorAxisLength > (mSTATS
    .MajorAxisLength*lowerBound)) && ...
            (stats(i).MinorAxisLength < (mSTATS
    .MinorAxisLength*upperBound)) && ...
            (stats(i).MinorAxisLength > (mSTATS
    .MinorAxisLength*lowerBound)) && ...
            (stats(i).Eccentricity < (mSTATS.
    Eccentricity*upperBound)) && ...
            (stats(i).Eccentricity > (mSTATS.
    Eccentricity*lowerBound)))
        N= floor(stats(i).Centroid(1));
        M= floor(stats(i).Centroid(2));
        reconstructed(M,N) = true;
        plot(N,M,'s', 'color','red')
    end
end
hold off;
```



The intrinsic matrix containing 5 intrinsic parameters. These parameters encompass focal length, image format, and principal point. The parameters and represent focal length in terms of pixels, where and are the scale factors relating pixels to distance. represents the skew coefficient between the x and the y axis, and is often 0. and represent the principal point, which would be ideally in the centre of the image. Nonlinear intrinsic parameters such as lens distortion are also important although they cannot be included in the linear camera model described by the intrinsic parameter matrix. This process reduces the dimensions of the data taken in by the camera from three to two (light from a 3D scene is stored on a 2D image). Camera resectioning determines which incoming light is associated with each pixel on the resulting image. In an ideal pinhole camera, a simple projection matrix is enough to do this. With more complex camera systems, errors resulting from misaligned lenses and deformations in their structures can result in more complex distortions in the final image. The camera projection matrix is derived from the intrinsic and extrinsic parameters of the camera, and is often represented by the series of transformations; e.g., a matrix of camera intrinsic parameters, a 3 × 3 rotation matrix, and a translation vector. The camera projection matrix can be used to associate points in a camera's image space with locations in 3D world space.



ms are highlighted



The intrinsic matrix containing 5 intrinsic parameters. These parameters encompass focal length, image format, and principal point. The parameters and represent focal length in terms of pixels, where and are the scale factors relating pixels to distance. represents the skew coefficient between the x and the y axis, and is often 0. and represent the principal point, which would be ideally in the centre of the image. Nonlinear intrinsic parameters such as lens distortion are also important although they cannot be included in the linear camera model described by the intrinsic parameter matrix. This process reduces the dimensions of the data taken in by the camera from three to two (light from a 3D scene is stored on a 2D image). Camera resectioning determines which incoming light is associated with each pixel on the resulting image. In an ideal pinhole camera, a simple projection matrix is enough to do this. With more complex camera systems, errors resulting from misaligned lenses and deformations in their structures can result in more complex distortions in the final image. The camera projection matrix is derived from the intrinsic and extrinsic parameters of the camera, and is often represented by the series of transformations; e.g., a matrix of camera intrinsic parameters, a 3 × 3 rotation matrix, and a translation vector. The camera projection matrix can be used to associate points in a camera's image space with locations in 3D world space.

A. *Part 2: Transform m's Into Pixels*

To transform the m's into single pixels, we can simply run a hit and miss operation using our m mask and the complement of the m mask. But because I elected to use region properties for this example, it was relatively easy to just include a centroid calculation and toggle the centroid of the detected m into a true pixel value. This code can be seen in the previous section's for loop. The total number of m's in the image was 51.

```
figure('name','m are now pixels'), imshow(
    reconstructed), title('m now pixels')
mCount = sum(sum(reconstructed))
```

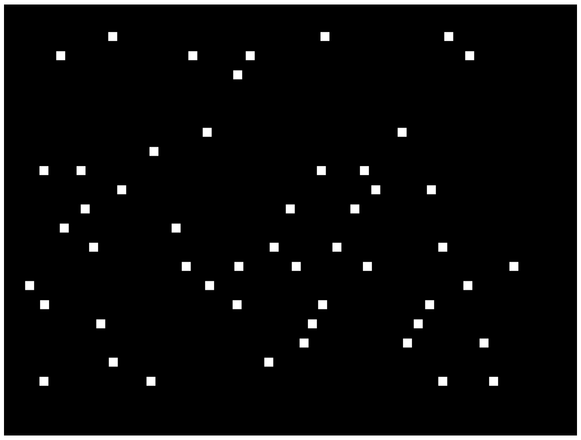
```
mCount =

    51
```

m now pixels

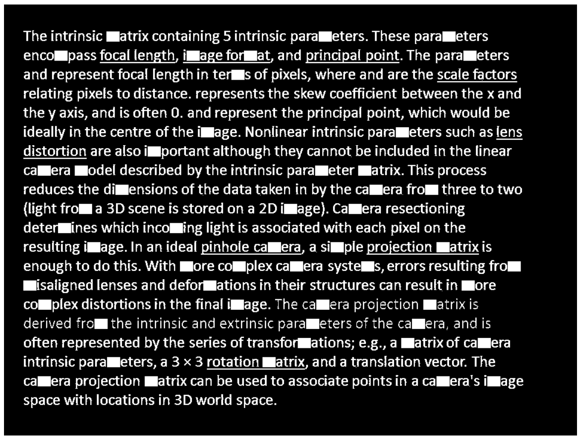
B. *Part 3: Transform m's into squares's*

I chose to transform the m's in the image into squares. By exploiting the functionality of dilation, I dilate the pixel image with a square, and add the resultant image to the original image.

```
remover=zeros(size(theM));
remover=imcomplement(remover);
Img = imdilate(reconstructed,remover);

square = strel('square',15);
adder = imdilate(reconstructed, square);
figure('name','adder'), imshow(adder)
Img2=floor(I+Img); %figure('name','img'),
    imshow(Img2);
Im3=Img2+adder;


figure('name','m are now squares'), imshow(Im3)
    , title('m now different')
```



m now different



The intrinsic ■atrix containing 5 intrinsic para■eters. These para■eters enco■pass focal length, i■age for■at, and principal point. The para■eters and represent focal length in ter■s of pixels, where and are the scale factors relating pixels to distance. represents the skew coefficient between the x and the y axis, and is often 0. and represent the principal point, which would be ideally in the centre of the i■age. Nonlinear intrinsic para■eters such as lens distortion are also i■portant although they cannot be included in the linear ca■era ■odel described by the intrinsic para■eter ■atrix. This process reduces the di■ensions of the data taken in by the ca■era fro■ three to two (light fro■ a 3D scene is stored on a 2D i■age). Ca■era resectioning deter■ines which inco■ing light is associated with each pixel on the resulting i■age. In an ideal pinhole ca■era, a si■ple projection ■atrix is enough to do this. With ■ore co■plex ca■era syste■s, errors resulting fro■ ■isaligned lenses and defor■ations in their structures can result in ■ore co■plex distortions in the final i■age. The ca■era projection ■atrix is derived fro■ the intrinsic and extrinsic para■eters of the ca■era, and is often represented by the series of transfor■ations; e.g., a ■atrix of ca■era intrinsic para■eters, a 3 × 3 rotation ■atrix, and a translation vector. The ca■era projection ■atrix can be used to associate points in a ca■era's i■age space with locations in 3D world space.

C. Part 4: Discussion of Results

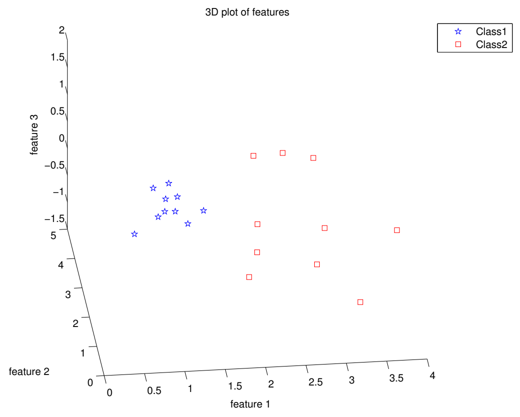
After counting the m's individually, I have verified that they were all found using my method.

II. Problem 2

In order to classify these two classes, they are first plotted to determine if they are separable. The plot reveals that they are linearly separable by a plane.

```
clear all; clc; close all;
class1=[0.8963, 1.4780, 0.9023;
        1.1023, 0.7144, 0.8438;
        0.7078, 1.0981, 1.3255;
        0.8516, 1.1091, 0.8739;
        1.0255, 1.2991, 1.0301;
        0.7408, 0.7857, 0.9575;
        0.9521, 0.7835, 1.0452;
        1.3166, 0.9372, 0.9489;
        0.4496, 0.8064, 0.6517;
        0.9034, 1.1314, 1.3808;]

class2=[3.7985, 1.8831, -0.1202;
        2.8175, 2.4902, 0.9653;
        2.7783, 0.5197, 0.7404;
        1.9085, 1.2397, -0.4936;
        3.2815, 1.1903, -1.0368;
        2.2147, 4.0108, 0.2256;
        2.3083, 1.0618, 1.8742;
        2.1250, 2.5301, -0.7521;
        2.8540, 2.3891, -0.9560;
        2.0397, 1.5494, 0.3092;]
figure('name','3dplot')
plot3(class1(:,1), class1(:,2), class1(:,3), 'p
    ','color','blue'), hold on
plot3(class2(:,1), class2(:,2), class2(:,3), 's
    ','color','red'), hold on
xlabel('feature 1')
ylabel('feature 2')
zlabel('feature 3')
title('3D plot of features');
legend('Class1','Class2'), hold off
```

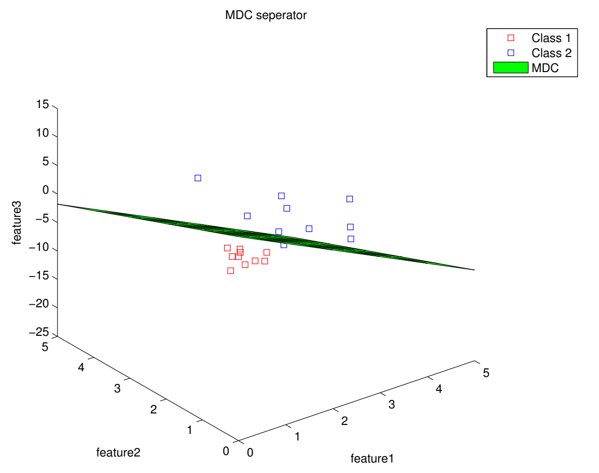
```
class1 =

    0.8963    1.4780    0.9023
    1.1023    0.7144    0.8438
    0.7078    1.0981    1.3255
    0.8516    1.1091    0.8739
    1.0255    1.2991    1.0301
    0.7408    0.7857    0.9575
    0.9521    0.7835    1.0452
    1.3166    0.9372    0.9489
    0.4496    0.8064    0.6517
    0.9034    1.1314    1.3808


class2 =

    3.7985    1.8831   -0.1202
    2.8175    2.4902    0.9653
    2.7783    0.5197    0.7404
    1.9085    1.2397   -0.4936
    3.2815    1.1903   -1.0368
    2.2147    4.0108    0.2256
    2.3083    1.0618    1.8742
    2.1250    2.5301   -0.7521
    2.8540    2.3891   -0.9560
    2.0397    1.5494    0.3092
```



A. Part 3: Minimum Distance Classifier

Using the equations defined in class, the following equation was derived in order to find the Minimum Distance Classifier.

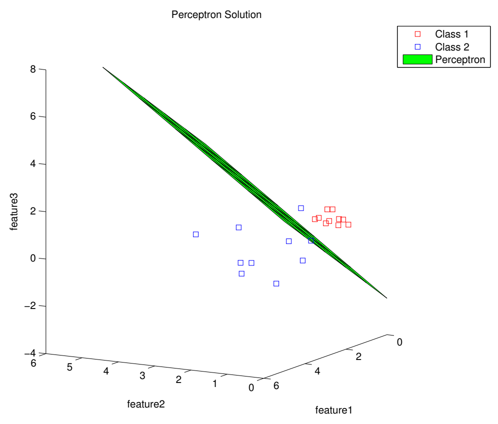
```
%$Z=\frac{(-2*X*\bar{X_{1}}-(\bar{X_{1}}^2)+(2*
    Y*\bar{Y_{1}})-(\bar{Y_{1}}^2)-(\bar{Z_
    {1}}^2)-(2*X*\bar{X_{2}})+(\bar{X_{2}}^2)
    -(2*Y*\bar{Y_{2}})+(\bar{Y_{2}}^2)+(\bar{Z_
    {2}}^2)}{2*(\bar{Z_{2}}-\bar{Z_{1}})}$
clc; close all;

%MDC C:
x1b=mean(class1(1,:));
x2b=mean(class2(1,:));

y1b=mean(class1(2,:));
y2b=mean(class2(2,:));

z1b=mean(class1(3,:));
z2b=mean(class2(3,:));

X=0:5;
Y=0:5;

Z=zeros(6,6);
for i=0:5
    for j=0:5
        Z(i+1,j+1)=((2*i*x1b)-(x1b^2)+(2*j*y1b)
    -(y1b^2)-(z1b^2)-(2*i*x2b)+(x2b^2)-(2*j*y2b
    )+(y2b^2)+(z2b^2))/(2*(z2b-z1b));
    end
end



figure('name','MDC wooooo');
%
%surf(X,Y,Z), hold on
plot3(class1(:,1), class1(:,2), class1(:,3), 's
    ','color','red'), hold on
plot3(class2(:,1), class2(:,2), class2(:,3), 's
    ','color','blue'), hold on
surf(X, Y, Z), colormap([0,1,0]);
%axis([0, 2, 0, 2, 0, 2]), title('MDC woooooooo
    ');
xlabel('feature1'); ylabel('feature2'); zlabel(
    'feature3');
title('MDC seperator'); legend('Class 1', '
    Class 2', 'MDC'), hold off;
```



B. Part 4: Perceptron

```
clc;
clear Z
%build training set
trainingSet = [class1;class2]
[m, n] = size(trainingSet);
resultSet=ones(1,m)';
resultSet(11:20)=0;
weightVector = ones(1,n)/5; %

threshold = 0;%threshold to decide if the
    output is good or bad. usually this is 0
error_count = 1;
bias = 0.1;
iterationNo = 1;
learningRate = 1;
% training phase
while (error_count > 0)
    error_count = 0;
    for i=1:m
        gx=dot(weightVector,trainingSet(i,:))+
    bias;
            if (gx > threshold)
                result = 1;
            else
                result = 0;
            end

            error = resultSet(i)-result;
            if (error ~= 0)
                error_count = error_count + 1;
                weightVector = weightVector + (
    learningRate*(error))*trainingSet(i,1:n);
                bias = bias + learningRate*
    error;
            end
    end

    if (iterationNo >= 100000)
        disp('Neuron input calculation couldn''
    t completed in timely fashion.');
        break
    end
    iterationNo = iterationNo +1;
end
disp(['answer converged in ' num2str(
    iterationNo) ' iterations']);
disp(['weights: ' num2str(weightVector)]);
disp(['bias: ' num2str(bias)]);


X=0:5;
Y=0:5;
Z=zeros(6,6);
for i=0:5
    for j=0:5
        Z(i+1,j+1)=(-(weightVector(1)*i)-(
    weightVector(2)*j)-bias)/weightVector(3);
    end
end



figure('name','Now with NN!');
plot3(class1(:,1), class1(:,2), class1(:,3), 's
    ','color','red'), hold on
plot3(class2(:,1), class2(:,2), class2(:,3), 's
    ','color','blue'), hold on
surf(X,Y,Z), colormap([0,1,0]), hold on
xlabel('feature1'); ylabel('feature2'); zlabel(
    'feature3');
legend('Class 1', 'Class 2', 'Perceptron'),
    title('Perceptron Solution'), hold off;
```

```
trainingSet =
    0.8963    1.4780    0.9023
    1.1023    0.7144    0.8438
    0.7078    1.0981    1.3255
    0.8516    1.1091    0.8739
    1.0255    1.2991    1.0301
    0.7408    0.7857    0.9575
    0.9521    0.7835    1.0452
    1.3166    0.9372    0.9489
    0.4496    0.8064    0.6517
    0.9034    1.1314    1.3808
    3.7985    1.8831   -0.1202
    2.8175    2.4902    0.9653
    2.7783    0.5197    0.7404
    1.9085    1.2397   -0.4936
    3.2815    1.1903   -1.0368
    2.2147    4.0108    0.2256
    2.3083    1.0618    1.8742
    2.1250    2.5301   -0.7521
    2.8540    2.3891   -0.9560
    2.0397    1.5494    0.3092

answer converged in 7 iterations
weights: -3.8326     -0.9679      3.5008
bias: 3.1
```

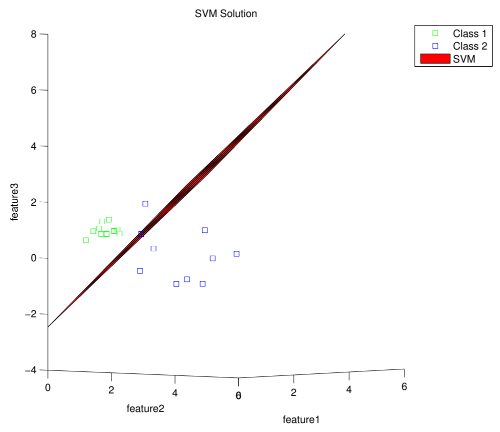


C. *Part 5 Support Vector Machine*

The equations derived in class were modified to produce the optimization problem. First, the data points were inspected and the points that were closest to the other groups were chosen. Next, a Q function was solved in terms of α and dot product of the closest classes. Setting the differential of this $Q = 0$, α was solved. This was plugged into the equations to produce a weight matrix and a bias, and the plane was plotted.

```
clc; close all;
clearvars -except class1 class2

x1=[1.3166, 0.9372, 0.9489]'
x2=[2.0397, 1.5494, 0.3092]'

x11=dot(x1,x1);
x12=dot(x1,x2);
x22=dot(x2,x2);

%syms a X Y Z
%Q=a+a-(1/2)*((a^2)*x11-(2*a*a*x12)+(a^2)*x22);
%alpha=double(solve(diff(Q)==0,a))

alpha = 1.5304 %FOUND BY USING THE ABOVE
    SYMBOLIC TOOLBOX CODE^^^

W=(alpha*(1)*x1) + (alpha*(-1)*x2)
b=1-(alpha*(1)*x11+alpha*(-1)*x12)

%SVM_solX=(-W(2)*Y-W(3)*Z-b)/W(1);
X=0:5;
Y=0:5;

Z=zeros(5,5);
for i=0:5
    for j=0:5
        Z(i+1,j+1)=(-(W(1)*i)-(W(2)*j)-b)/W(3);
    end
end

figure('name','Now with SVM!');
plot3(class1(:,1), class1(:,2), class1(:,3), 's
    ','color','green'), hold on
plot3(class2(:,1), class2(:,2), class2(:,3), 's
    ','color','blue'), hold on
surf(X,Y,Z), colormap([1,0,0]); hold on;
xlabel('feature1'); ylabel('feature2'); zlabel(
    'feature3');
legend('Class 1', 'Class 2', 'SVM'), title('SVM
     Solution'); hold off;
```

```
x1 =

    1.3166
    0.9372
    0.9489


x2 =

    2.0397
    1.5494
    0.3092


alpha =

    1.5304


W =

   -1.1066
   -0.9369
    0.9790


b =

    2.4061
```



D. Part 6 Discussion of Results

This problem was slightly harder to understand because the classes were defined to have three dimensions, so visualizing their distributions was a little more complicated than the two dimensional examples that I was used to. The separators all worked very effectively, but it seemed that the MDC method produced the best separator. This was surprising because the MDC separator worked better than the SVM method, which by theory should be the optimal solution. This is likely because I plotted the flat plane produced by the output of the SVM method, as opposed to the ideal kernel that would be produced through using the quadprog function.