Introduction to Vision and Robotics
Vision Assignment

Georgi Hristov s1446364
Martina Kotseva s1449090

Introduction

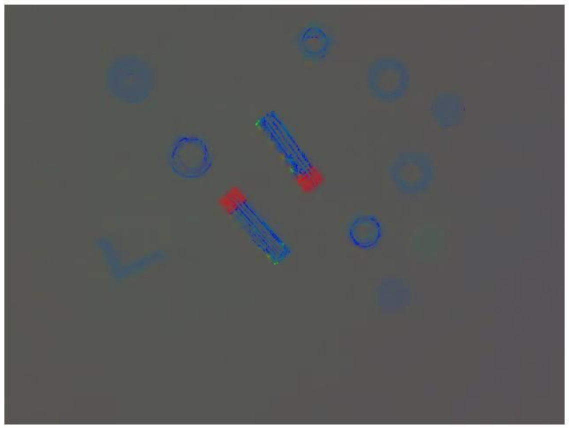
The purpose of this assignment is the development of a Matlab program capable of identifying, segmenting and classifying coins and objects from an image - effectively a coin counter. To summarise our approach at the problem we divided our program into the following stages - background creation using median colours of all images, image adjusting and processing, segmenting objects, creating a data set of all objects and their properties, labelling their true classes, dividing the data set into a training and test set, learning a classifier using the training set and testing it on the test set.

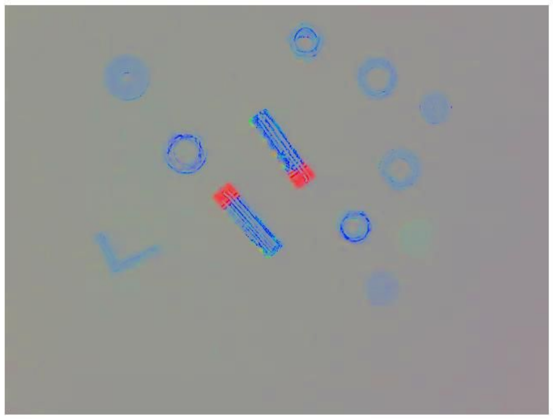
We experimented with different approaches regarding each stage and chose the ones that gave the best results.

Methods

Background

Since the camera angle and illumination are stationary, an obvious approach is to subtract the background from the image which would only leave us with the objects we are classifying. To create this background, we took the median values of every colour channel of all the pixels across all available images, as suggested in the assignment. Combining these median values effectively synthesise an estimate of the background. To deal with the different illumination of the objects we normalized the RGB colour values by dividing each channel by *sqrt(R^2 + G^2 + B^2)* since it gave a brighter and more contrasting image then simply dividing by the sum *R+G+B*.





In our initial approach we first normalized the images and then synthesized the background, however that still produced some noise in the created background image. A better solution was to first use the raw RGB images to estimate the background and then RGB normalize it and the images.

The created background still had some noise due to some objects not changing position across many images. Thus to make sure less noise made its way later on in the processed images we further filtered the background using the method *medfilt2()*. This affected the computational speed of our program however produced an almost monotonous, noiseless background.

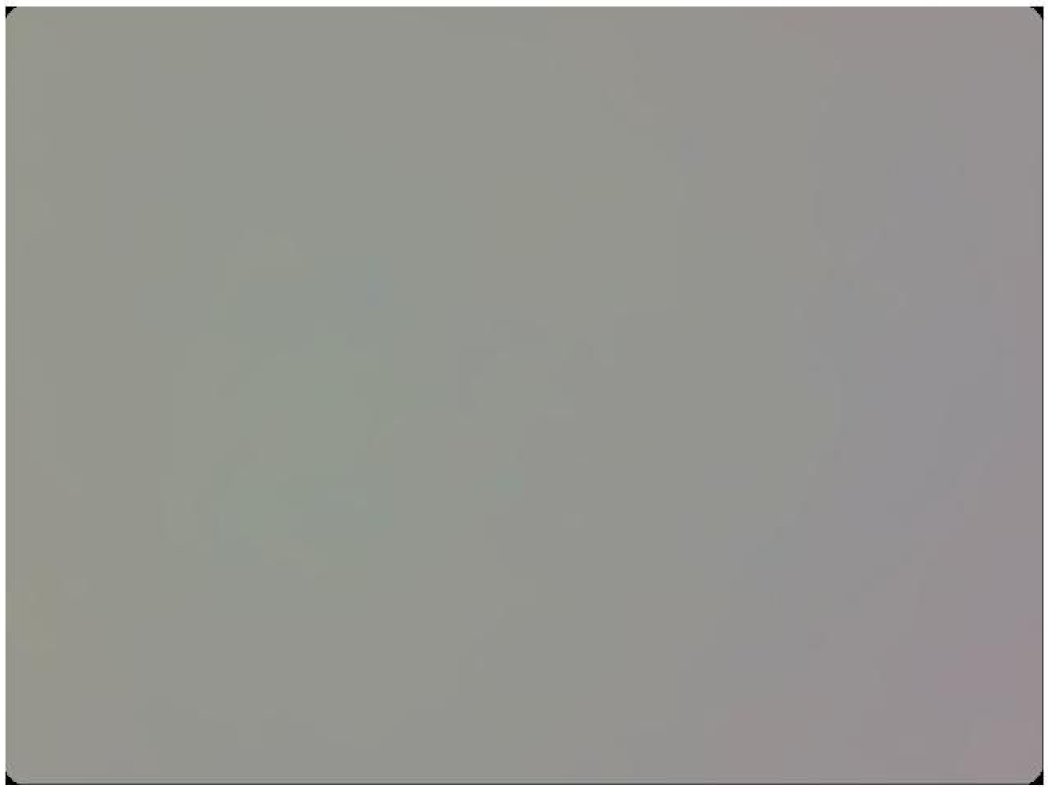
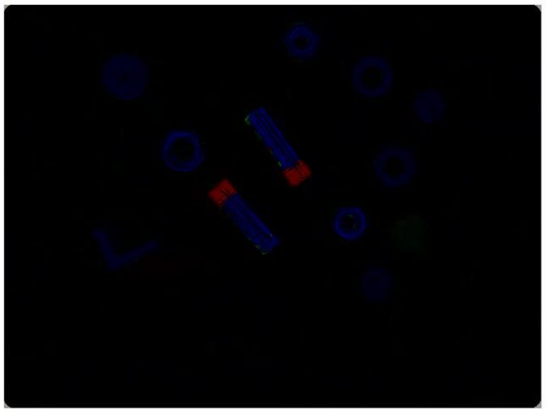
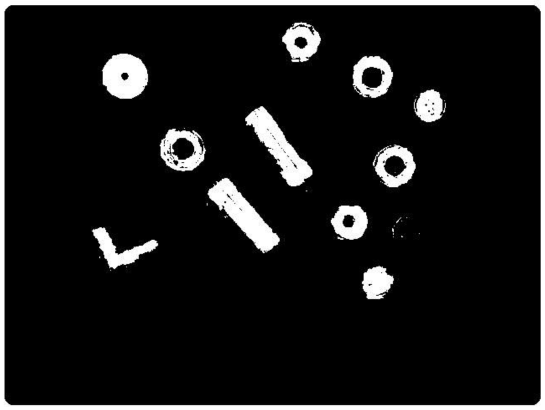
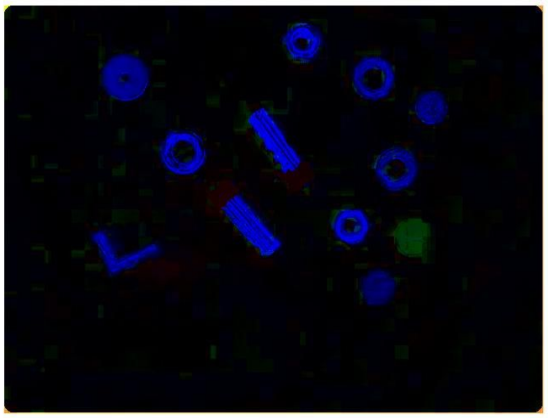
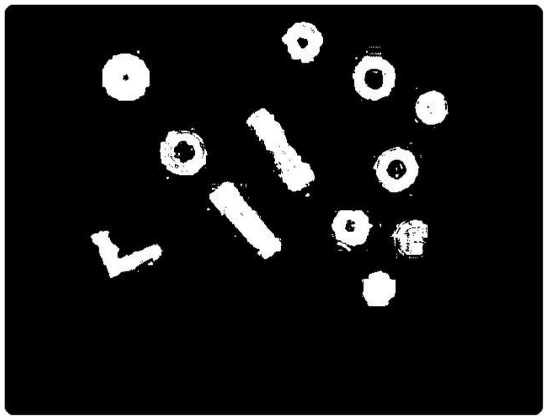


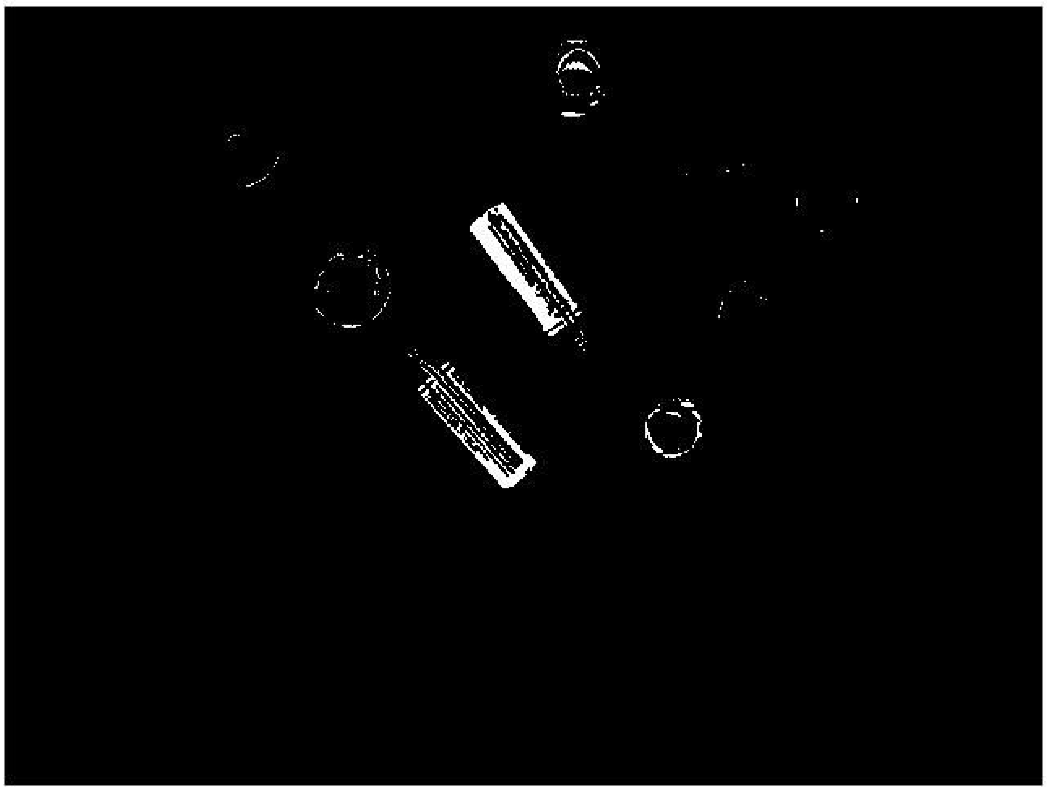
Image Adjusting

The next step was to subtract the background from the image. A problem we faced at that stage was that the background and some of the images (like the one-pound coin for instance) were very similar in colour, which led to losing parts of them when subtracting the background. The solution that we came up with was to boost the contrast in the normalized image and background. We did that by using the function *imadjust*. After experimenting with its parameters for a while, we found values that gave us good contrast and worked well on the image data we were given to test our program. We then realised this solution was not good enough, however, because the values we had found were dependent on the background and the fact that it worked for those specific images did not mean that it was going to do well in the general case. We needed something more flexible and an alternative that we found was the *stretchlim* function, which calculates the lower and upper limits that can be used for contrast stretching an image.

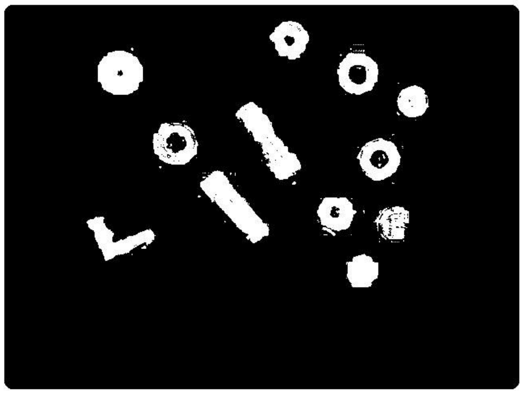





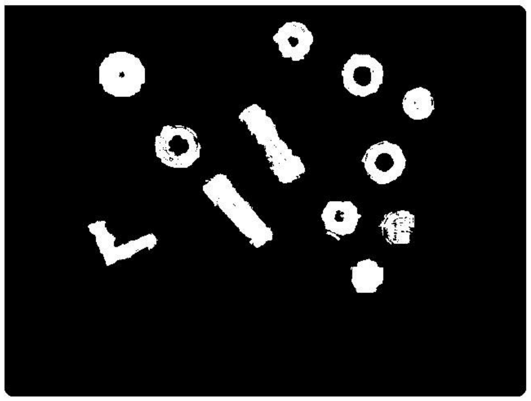
Object Segmentation

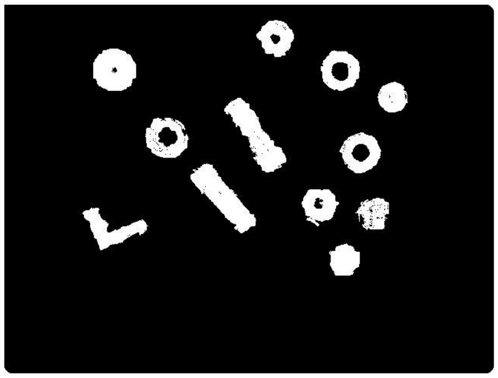
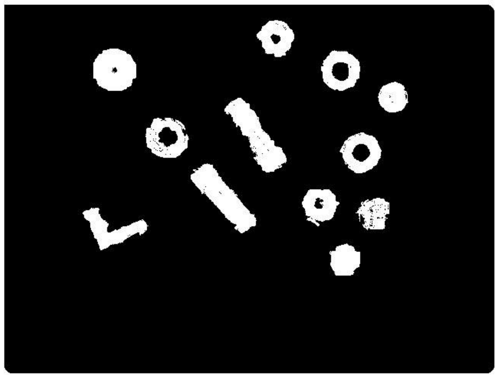
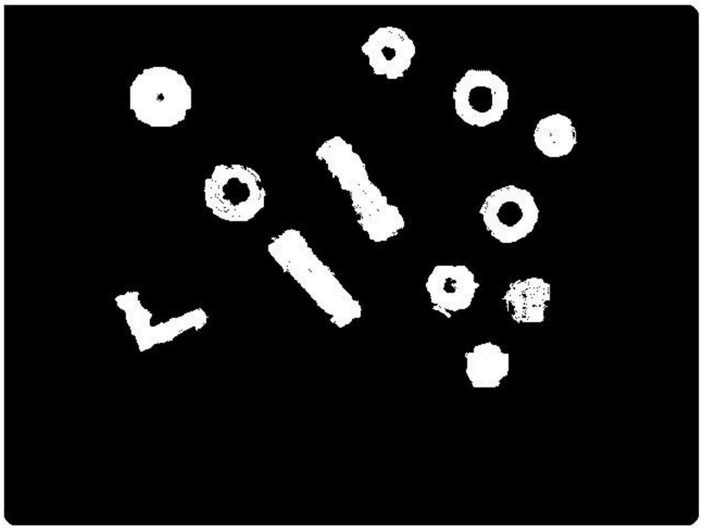
Once the background was removed, we had to separate the individual objects in each picture. First, we needed to obtain a binary image of the objects that we were left with. Our initial approach was to grayscale the image with the *rgb2gray* function that Matlab provides, then threshold the grey scaled image and binarize it using the *myim2bw.m* code we were given. All of the colours on the images were very similar and the results were not satisfactory.



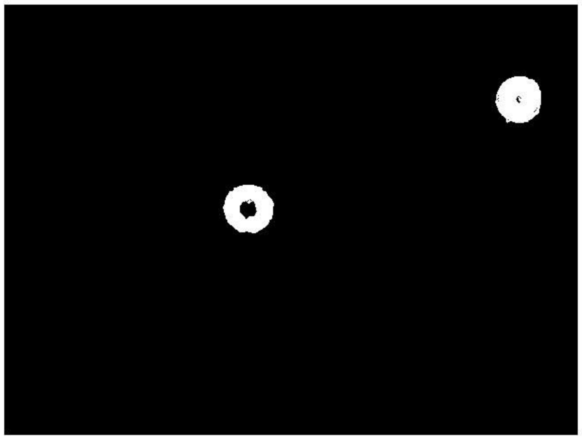
That made us switch to another way of thresholding. We separated the image (with normalized colours and subtracted background) into its three channels – red, green and blue. We then calculated the mean intensity and the standard deviation of each channel, set to one the values that were at no more than 1 or 2 standard deviations from the mean for that channel and merged the three together to create a binary image of the objects. There was still a certain amount of noise and imperfections, which we further smoothened with the functions *bwareaopen* and *bwmorph*.





The transformed image was essentially a couple of connected components. Using the provided function *getlargest* we could extract the objects one by one, removing them from the image until it changed no more. After some testing we noticed a bug. Whenever two components had the same size, *getlargest* returned them simultaneously as if they were one object even if they were completely separated from each other.



A useful function to get around this problem was *bwlabel* which turns each of the 1's in the matrix that represents the binary image into a number corresponding to the label of that component. In this way we could easily extract each of the objects.
The objects, obtained from all of the images were then stored in a list, which we went through and hardcoded the true classes. Whenever some noise was detected as an object or an object was deemed too poorly detected, we took account of that and later cleaned the data so as to have a better training data set.

Extracting Object Properties

The next thing on the agenda was to decide which features of the objects characterized them best and would help us distinguish them from one another. This was no easy task and required a lot of experimenting.

The area and compactness helped telling the big objects (such as the angle bracket and the battery) from the small ones (like the nut or the 5p coin).

The object in the images could be oriented in all directions, which is why it was a good idea to include two of their moments, which were rotation invariant.

The four features listed above we extracted using the *getproperties* function provided for us.

At first we thought adding some representation of the object colour would be beneficial. We tried adding the mean value for each of the three colour channels. Then we repeated the experiment with the median and the maximum intensity, but neither of the three improved the classification rate of our program. Although we achieved better classification results for the 1 and 2 pound coins, which are of a slightly different color than the rest of the objects, the overall classification rate fell as classification of the rest of the objects became worse. This was due to the fact that the objects were all metallic and produced very similar colour values. Furthermore, the training data we were working with was limited and we had to be careful not to use too many features.

Another thing we noticed was that most of the objects were round and not very different in size, but some of them had holes which made them stand out a bit more. The size of the hole could easily be calculated by subtracting the filled area of the image from its total area. To obtain those two, we used Matlab's *regionprops,* which returns measurements for a set of properties for each connected component in a binary image.

Classification and Testing

To classify the now gathered data, we use a multivariate Gaussian distribution model. We fit the training data using *buildmodel* and make the predictions for the test data using *classify* (both functions from the course web page).

In the beginning we were training the classifier with thirteen pictures and had one left for testing, but the accuracy we got was different every time due to the small size of the data set. This made us switch to K-fold cross-validation. With this method, every point of the data set gets to be both a training and a testing one over different iterations. This significantly reduces the chances of getting a biased training set and gives a better evaluation of our classifier.

At the end the accuracy of our algorithm hit 70%.

Discussion

Apart from the processing speed which needs improvement, our programs works with a relatively good success rate. Although we know which parts of the process need improving, due to time constraints we settled on this final code. To improve the overall performance, a better method to filter the synthesized background is needed, which is the main reason for the slower processing speed. Furthermore, creating more training data and adding more features would improve the classifier. As colour is an obvious feature, we would need to pinpoint why exactly it did not improve the overall performance of the classifier and fix the problem. Another limitation of our program is that our classifier does not treat harder cases differently and still classifies them to the best option. We experimented with adding this option to our program but although it worked for badly detected cases it also denied classifying objects which it previously classified correctly.

Assignment credit: 50% Georgi Hristov, 50% Martina Kotseva

Appendix

Function for turning a colour image into a binary one:

```
function [ bw ] = binarizeRGB( image )
%UNTITLED2 Summary of this function goes here
%   Detailed explanation goes here

redChannel = image(:, :, 1);
greenChannel = image(:, :, 2);
blueChannel = image(:, :, 3);

% imshow(blueChannel); figure;

[n, m] = size(redChannel);
redmean = 0;
for i = 1:n
    for j = 1:m
        redmean = redmean + (redChannel(i,j) / (n*m));
    end
end
redsd = 0;
for i = 1:n
    for j = 1:m
        redsd = redsd + (((redChannel(i,j) - redmean)*(redChannel(i,j) - redmean)) / (n*m));
    end
end
redsd = sqrt(redsd);

bwred = double(abs(redChannel - redmean) > 1*redsd);
%imshow(bwred);

greenmean = 0;
for i = 1:n
    for j = 1:m
        greenmean = greenmean + (greenChannel(i,j) / (n*m));
    end
end
greensd = 0;
for i = 1:n
    for j = 1:m
        greensd = greensd + (((greenChannel(i,j) - greenmean)*(greenChannel(i,j) - greenmean)) / (n*m));
    end
end
greensd = sqrt(greensd);

bwgreen = double(abs(greenChannel - greenmean) > 2*greensd);
%imshow(bwgreen);

bluemean = 0;
for i = 1:n
    for j = 1:m
        bluemean = bluemean + (blueChannel(i,j) / (n*m));
    end
end
bluesd = 0;
```

```
for i = 1:n
    for j = 1:m
        bluesd = bluesd + (((blueChannel(i,j) - bluemean)*(blueChannel(i,j) - bluemean)) / (n*m));
    end
end
bluesd = sqrt(bluesd);

bwblue = double(abs(blueChannel - bluemean) > 1*bluesd);
%imshow(bwblue);

bw = bwred + bwgreen + bwblue;

end
```

Function for normalizing an RGB image:

```
function [ norm_image ] = normalizeRGB( image )
%Normalizes RGB of an image
%   Detailed explanation goes here
        imageR = image(:,:,1);
        imageG = image(:,:,2);
        imageB = image(:,:,3);

        NormalizedRed = imageR./ sqrt(imageR.^2 + imageG.^2 + imageB.^2);
        NormalizedGreen = imageG./sqrt(imageR.^2 + imageG.^2 + imageB.^2);
        NormalizedBlue = imageB./sqrt(imageR.^2 + imageG.^2 + imageB.^2);

        norm_image = cat(3,NormalizedRed,NormalizedGreen,NormalizedBlue);

end
```

Function for extracting the background from a set of images:

```
function [ background ] = synthesize_background( images_number, image_name)
%UNTITLED Summary of this function goes here
%   Detailed explanation goes here

red = zeros(480,640,images_number);
green = zeros(480,640,images_number);
blue = zeros(480,640,images_number);

for i = 1 : images_number

            name = sprintf(image_name,i);
            image = imread(name);

            image = double(image);

            for r = 1 : 480
                for c = 1 : 640
                    red(r,c,i) = image(r,c,1);
                    green(r,c,i) = image(r,c,2);
                    blue(r,c,i) = image(r,c,3);
                end
            end
end
```

```
bgR = median(red,3);
bgG = median(green,3);
bgB = median(blue,3);


%Form and normalize background RGB
norm_background = cat(3, bgR, bgG, bgB);
norm_background = normalizeRGB(norm_background);

%Filter  background to remove any leftover noise
new_bgR = medfilt2(norm_background(:,:,1), [18 18]);
new_bgG = medfilt2(norm_background(:,:,2), [18 18]);
new_bgB = medfilt2(norm_background(:,:,3), [18 18]);

background = cat(3,new_bgR,new_bgG,new_bgB);
```

Cross-Validation

```
%Hold-out
[Train, Test] = crossvalind('HoldOut', data_points, 0.10);

train_data = data(Train,1:features_number);
train_true = data(Train,features_number+1);

test_data = data(Test,1:features_number);
test_true = data(Test,features_number+1);


 %% Model build

[N,F] = size(train_data);
[Means,Invcors,Aprioris] = buildmodel(F,train_data,N,10,train_true);

%% Classification

disp('\nHold-out validation');
% Predict training data.
predictions_train = zeros(1,N);
for i = 1 : N
   test_vec = train_data(i,:);
   class =  classify(test_vec,10,Means,Invcors,F,Aprioris);
   predictions_train(i) = class;
end

disp('Training data accuracy: ');
disp(sum(predictions_train == train_true')/length(train_true'));

% Predict test data.
[N, F] = size(test_data);
predictions_test = zeros(1,N);
for i = 1 : N
   test_vec = test_data(i,:);
   class =  classify(test_vec,10,Means,Invcors,F,Aprioris);
   predictions_test(i) = class;
end
disp('Test data accuracy: ');
disp(sum(predictions_test == test_true')/length(test_true'));
```

```
% K-fold
indices = crossvalind('Kfold', data_points, 10);
cp = classperf(data(:,end));
for i = 1:10
    test = (indices == i); train = ~test;

    train_data = data(train,1:features_number);
    train_true = data(train,end);

    test_data = data(test,1:features_number);
    test_true = data(test,end);

    [N,F] = size(train_data);
    [Means,Invcors,Aprioris] = buildmodel(F,train_data,N,10,train_true);

    [N,F] = size(test_data);
    predictions_test_kfold = zeros(1,N);
    for j = 1 : N
        test_vec = test_data(j,:);
        class =  classify(test_vec,10,Means,Invcors,F,Aprioris);
        predictions_test_kfold(j) = class;
    end
    classperf(cp,predictions_test_kfold,test);
end
```