Introduction to Vision and Robotics
Assessed Practical 1: Coin Counter

Ioannis Baris(s1443483) and Tudor Ferariu(s1408714)

October 26th, 2016

1. Introduction

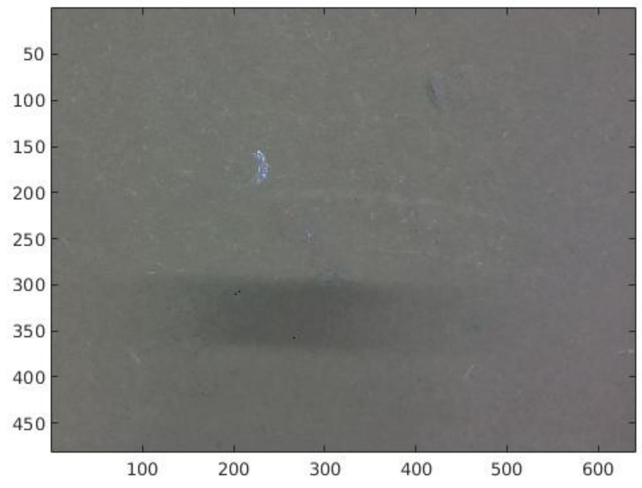
This report pertains to the methods we used and results we obtained in our attempt to identify, classify and count coins in an image. First we created a background by median filtering the provided images. By using background subtraction and thresholding we extracted the objects and used them to create a matrix of feature vectors. We then used this to train a Naive Bayes classifier.

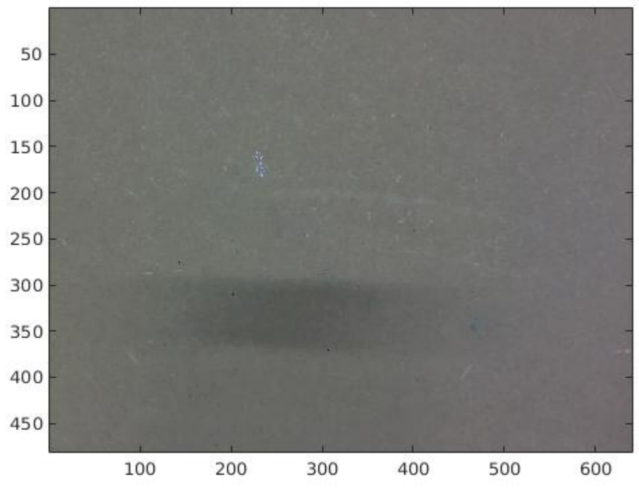
2. Methods

In order to extract the objects from an image we used *background subtraction*, for this we first needed to *normalise* the RGB images to account for changes in lighting.

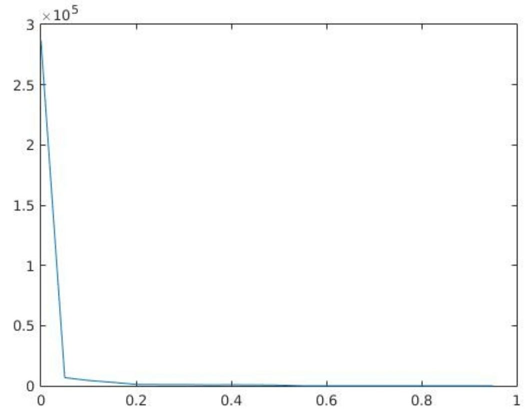
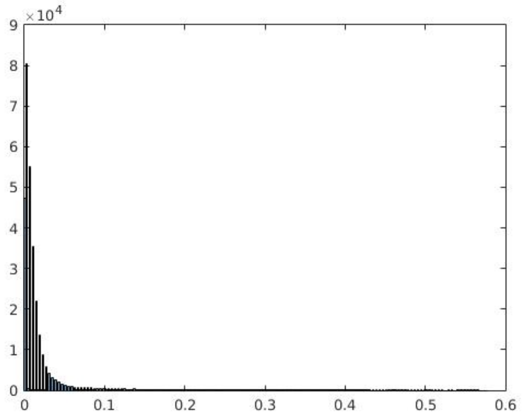
Our **normalise(Img)** function takes a normal RGB image, separates the three colour channels and then divides the value of each channel for every pixel by the sum of all three colours for the same pixel. Initially we used nested for loops to accomplish this task, but as that was very slow, we refined the method to the current state which is much faster because it uses matlab matrices more effectively. Trying to convert to grayscale instead of normalised RGB gave very poor results and as such was quickly scrapped as an idea.

The **background(Images)** function takes the desired 4D matrix of images and then normalises them in order to produce the best background by using median filtering. Here we experimented with the images we were given in order to obtain the best possible background and found out that using all the images except “02.jpg”, “05.jpg”, “17.jpg”, gave us the best result.

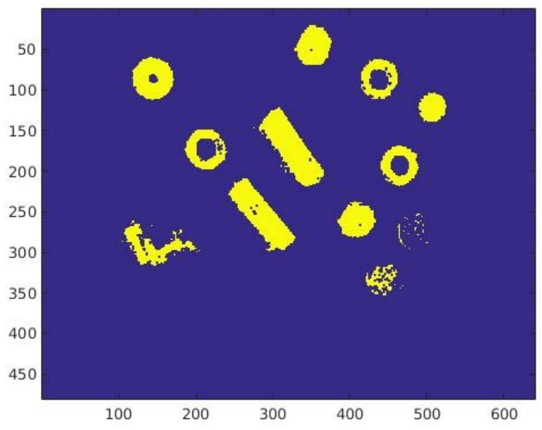

Using all images

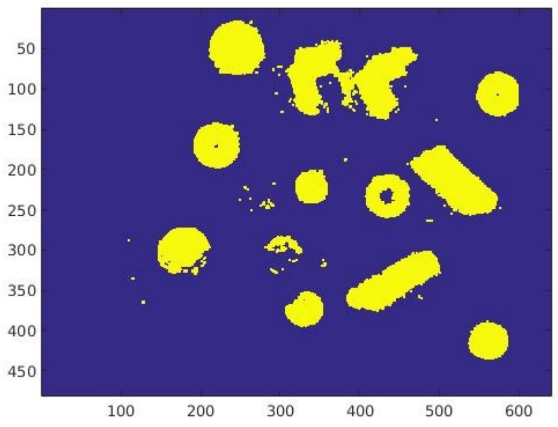

Improved Background

We created the **thresholding(Image,Background)** function in order to identify and extract each individual object. First it performs *background subtraction* after normalising the image and generates an intermediary where only the foreground objects remain, which is then split into the three colour channels. If any of the three colours are sufficiently different from the background, that pixel will appear in a new binary image that is created. We have chosen to represent the background as 0 and objects as 1. Because the normalised RGB values of the pixels in the intermediary are so close to 0, any automated method of determining the threshold has proven unreliable. Thus we experimented with different values in order to come across what we now think are suitable thresholds. Supporting this claim are the following histograms:

After obtaining the binary image we tried to process it using the **bwmorph** function, more notably using the "erode" and "dilate" parameters, but decided against including in the final implementation. Eroding is useful because it removes noise form the binary image, but due to the fact that some coins are very similar to the background, especially near the shadow, they also end up being removed, which we want to avoid. Dilating on the other hand makes objects more visible, but also creates more noise, sometimes causing close objects to appear as a single entity.


Eroded "02.jpg", some coins barely visible


Dilated "05.jpg", angle brackets connected

In order to remove noise we instead made use of the **bwareaopen** function and removed all blobs smaller than 225 pixels, this value made sure that noise, sometimes introduced by the imperfect background, was always removed. Unfortunately, it also removes the barely visible objects that appear within the cast shadow. Everything we tried that made objects in the shadow more visible, also generated massive amounts of noise, thus we concluded that we should instead focus on making the other objects as clear as possible.

The objects in the processed binary image are then labelled using the **bwlabel** function and stored as individual images in a cell array. We decided to store each object in a binary image the size of the original and not use a bounding box, as this would facilitate access to colour related information. This also makes it easier to obtain object features, as each object image has the same dimensionality.

For classification we chose **Naive Bayes**, making use of the helper functions provided for the labs. Our **train()** function uses many of the functions discussed above, as well as

buildmodel(Dim,Vecs,N,Numclass,Classes), **complexmoment(Image,u,v)** and a modified version of getproperties named **myproperties(Image,Original)**, to return the [Means, Invcors, Aprioris, Background] that we need for classifying new objects encountered. For the purposes of training we decided to use the first eight "simple" images and test on the ones remaining.

Our **myproperties(Image,Original)** function takes the binary representation of an object and the original image it was extracted form and produces a feature vector. The features we decided to use are

- Compactness
- Perimeter
- The first invariant moment
- The median value of the blue colour in the object

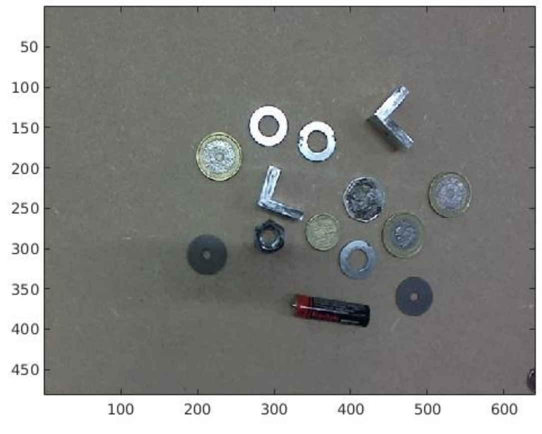
Compactness and the first invariant moment are the base that we started from, as they are generally good features. Perimeter was added because our classifier was confusing coins of different sizes (50p and 20p) and scale/rotation invariant moments did not seem enough to properly distinguish between some of the classes. Because the 1 pound coins were often misclassified, we decided to incorporate colour as well, specifically blue, as it was the one that varied most from the silver coins to the yellow ones. In order to obtain this feature we took the median value of the blue channel for each pixel in the object.

The **test(Means,Invcors,Aprioris,Background,NameOfImage)** function takes the output of training as well as the name of the image we want to classify the objects from. It shows the original image as well as each object and then prints the class for each object and the total amount counted in the image. For this we also defined a very simple function **obj(nr)** that takes the class number of the objects, prints its value and also returns it as a number for counting.

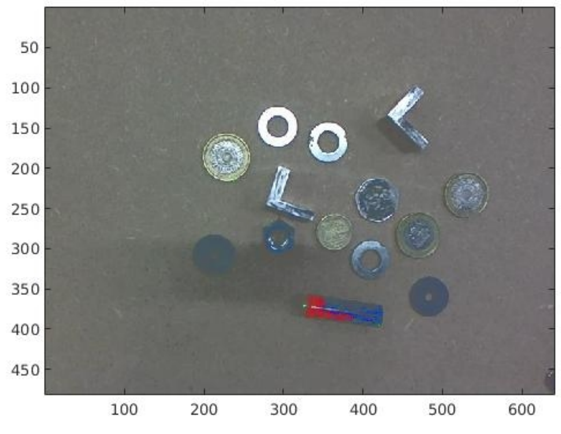
3. Results

The following is a step by step example of the way our implementation classifies objects from image "10.jpg".

Firstly the image is normalised:

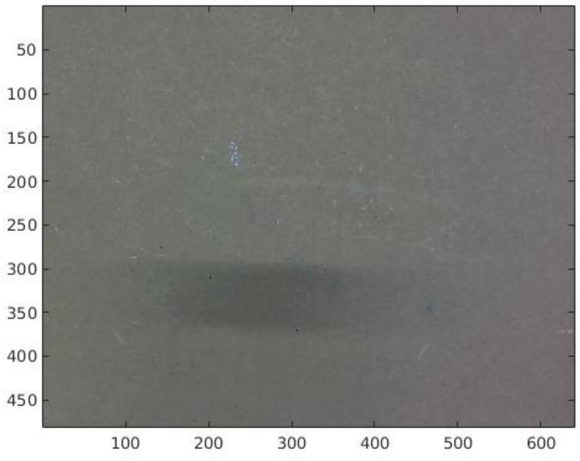


Original Image



Normalised RGB Image

Then the background subtraction takes place:


Background

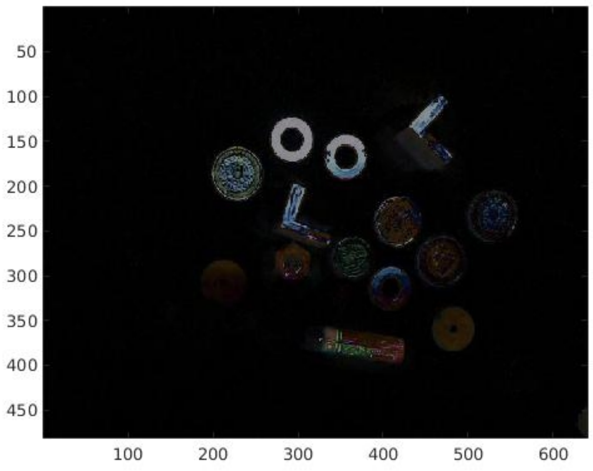
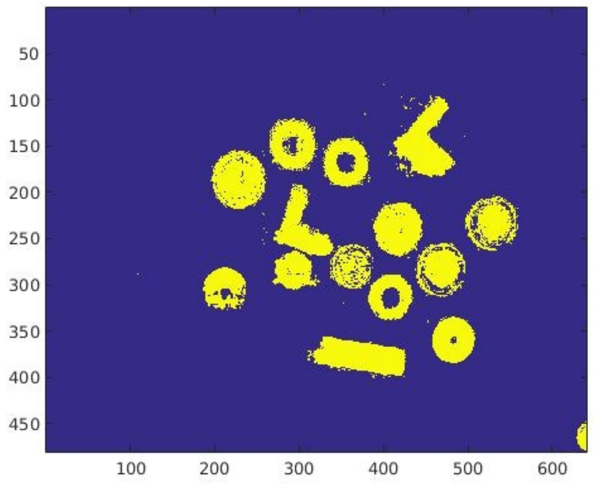
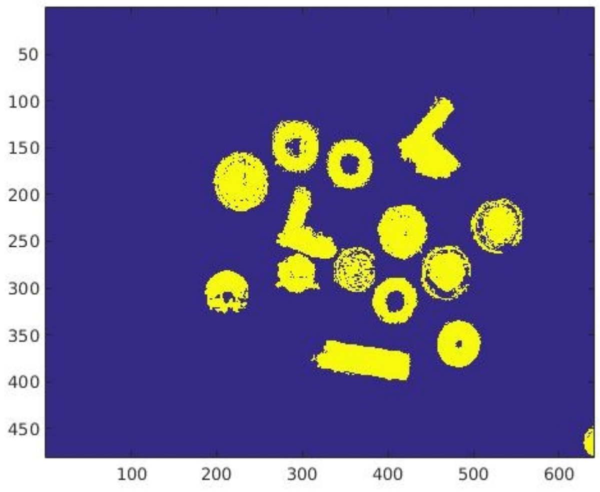


Image after background subtraction

Thresholding is applied to find the objects and noise is removed:


Binary image of identified objects


After noise removal

Individual items are labelled and then separated:

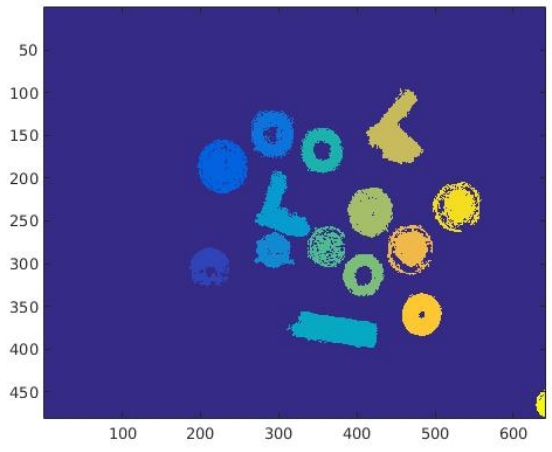
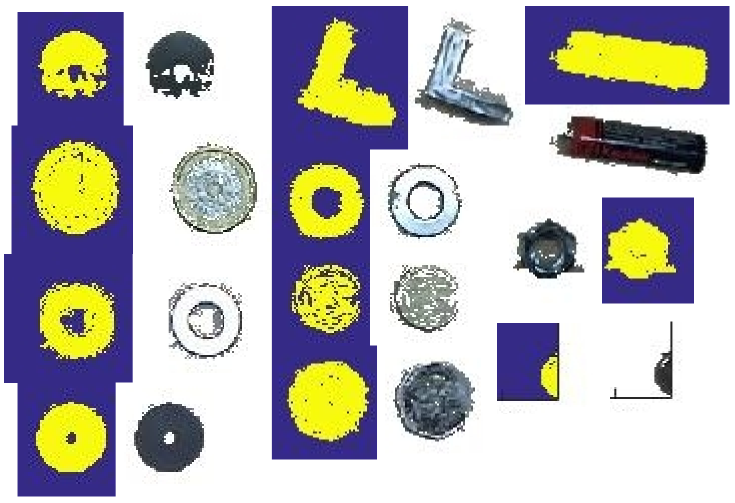


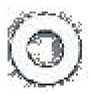
Image of labelled objects


Various individual objects identified

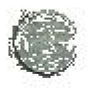
The objects are then used to generate a four feature vector for the purposes of either training the classifier or being classified themselves.

This particular image was used for testing and correctly classified 11/15 objects. We chose it as an example because it included some of the objects our classifier was having trouble with. Objects that get misclassified are the following:

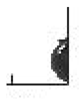
This 75p small hole washer is classified as a 2 pound coin, probably due to its erroneous size, which is likely caused by the shaded area.



For some reason the background inside this particular 25p large hole washer is not removed, as such it also appears as a 2 pound coin.



A problem we have encountered many times and tried our best to fix, the 1 pound coin often gets misclassified as 20p.



This is supposed to be a 5p coin, but given that it only appears partially, it classifies as a 2p angle bracket, as angle brackets likely have the most forgiving gaussian.

The remaining objects classify correctly, and almost all images used for training get classified with over 85% accuracy when re-tested. We think these are acceptable results, and would deem our implementation to be successful overall.

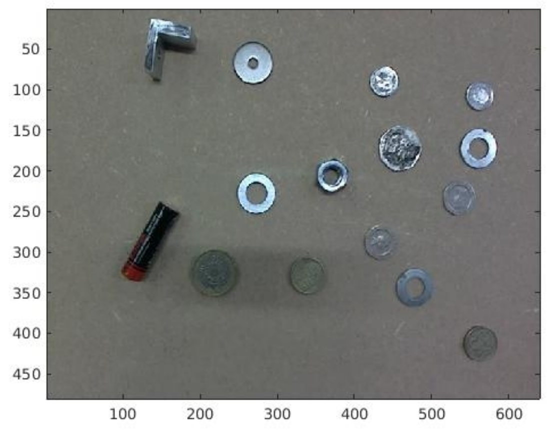
4. Discussion

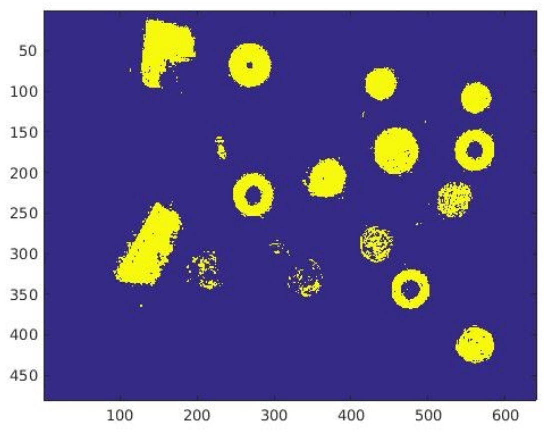
Our implementation assumes that we will be given enough images and that they all share the same background. It would also work if we have a separate image of the background.

We used **classify.m**, **complexmoment.m**, **buildmodel.m** and **multivariate.m** out of the helper functions given to us from the labs.

Of small note is the **myhist(Image,show)** function we created as a small variation to the **dohist()** provided to us. The initial intent was to use it for finding the threshold, but as we decided an automated method was unreliable, it was simply used to provide one of the histograms mentioned in section 2 of this report.

When detecting objects, we tried to get as many of them, as clearly as we could, but the shadow present in all images proved almost unmanageable. Coins fully in the shadow showed up so poorly that we concluded getting rid of them along with the surrounding noise was the best choice:

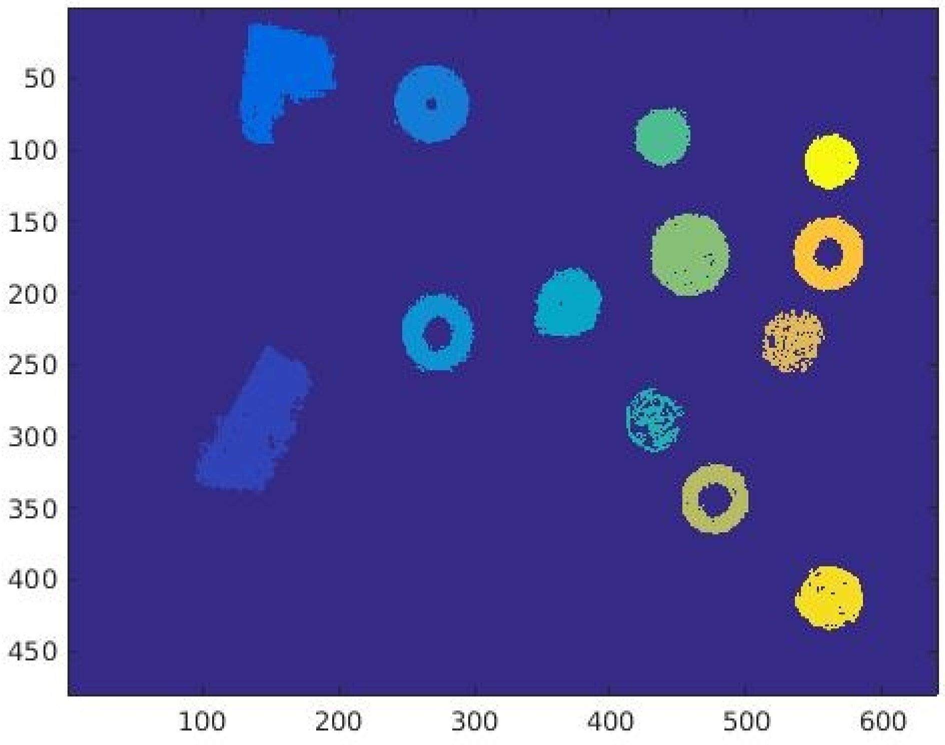




` Image “07.jpg” before

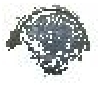
After tresholding

As the 2 pound and 1 pound coins in the shadow appear very fragmented and are similar to the noise caused by the background, they end up being removed altogether:



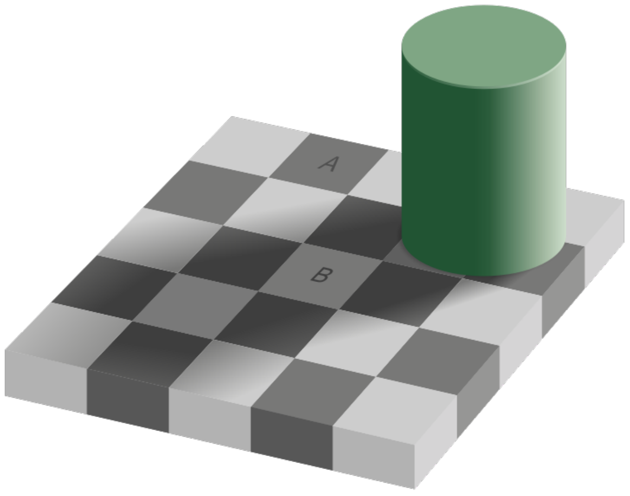
Coins partially in the shadow also appear very distorted, and often either get misclassified, or worse, confuse the training of the Gaussian:

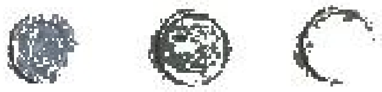
The lower halves of the 75p small hole washer on the left and the 2 pound coin on the right are within the shadow, and thus end up getting confused as the background, while only the upper halves appear correctly.



An interesting phenomenon that we observed was the fact that some bits of coin, especially from 20p and 1 pound coins, were getting subtracted alongside the background, although they appeared clearly in the image. Upon closer inspection, it did indeed appear that they had almost the same colour as the background that we created. This might be due to the use of our imperfect background, but immediately made us think about the Checker shadow illusion.

This raises fascinating questions about computer vision in general, as our brains are adept at pattern finding and often find differences between points that a computer would not be able to distinguish. The A and B squares in the image to the right are in fact the same colour, and the computer would identify them as such, and yet we can clearly perceive the checker pattern. These are some examples of objects where this happens:





For classification, the coins that were most often confused with one another were the 20p, 50p and 1 pound. We tried to manipulate our features to account for this, as the 20p and 50p have similar compactness, moments and colour, but different perimeters. For the 1 pound coins we created a colour feature, as they otherwise appear very similar to 20p. This might also be partly due to the abundance of 20p coins in the images, which gives them a higher apriori probability.

We chose to train on most of the provided images in order to have at least 4 features. If provided with more examples, features for all 3 colours and all the invariant moments may be added, possibly improving classification by a significant margin. Our decision was to not use duplicated images, as over-training with identical examples would result in lower performance on new images for even small differences. Similarly we only used the provided images and didn't take our own, as that would cause issues with the background algorithm, unless we could take them in the same conditions as the provided images.

In the beginning we briefly considered using k-Nearest Neighbours as a possible classifier, but the type of data seemed inadequate for it. Adding to this the fact that we were provided with helper code that aided in using Naive Bayes, we decided to quickly forego the idea.

Currently our code can not deal at all with pieces of coin, as can be seen with the partially missing 5p from the "10.jpg" example. Similarly, if objects are in contact with one another or overlapping, they will appear as a single object and likely get classified as angle brackets.

Despite all these limitations, we think our implementation works as intended and has acceptable classification accuracy. If provided with a much larger image set to extract features from, many aspects such as the background clarity and limit of features would increase, thus improving our overall accuracy.

Division of Work

We feel the division of work between us was about 50/50.

For this implementation we used pair programming, with Tudor as the *driver*, writing code and Ioannis as the *observer*, reviewing and correcting mistakes. Ideas of what to do and how to reach results came from both sides equally.

Ioannis experimented with the background, trying to find the best one, did the initial testing when we were deciding whether to use normalised RGB or grayscale and found many of the online resources for the functions we used to process the binary images. He was also the one to go through each identified object and individually label them. Bonus points for lightening the mood.

Tudor tested the classifier and came up with the list of possible feature combination and after reviewing all the results we both concluded which was the most successful. He also commented most of the code, wrote most of the report and extracted the various images used in this report.

The bulk of the code was written after brainstorming, back and forth discussion and mutual agreement, and thus can not be credited to one person in particular.

5. Code/Appendix

```
function [M,I,A,back] = train()

%load all the images
   img1 = imread('02.jpg');
   img2 = imread('03.jpg');
   img3 = imread('04.jpg');
   img4 = imread('05.jpg');
```

```matlab
    img5 = imread('06.jpg');
    img6 = imread('07.jpg');
    img7 = imread('08.jpg');
    img8 = imread('09.jpg');
    img9 = imread('10.jpg');
    img10 = imread('17.jpg');
    img11 = imread('18.jpg');
    img12 = imread('19.jpg');
    img13 = imread('20.jpg');
    img14 = imread('21.jpg');

    %Load the images we use for the background subtraction
    %Removed two of the images in order to get a better background

    %images =
cat(4,img1,img2,img3,img4,img5,img6,img7,img8,img9,img10,img11,img12,img13,img14);
    %images = cat(4,img3,img4,img5,img6,img7,img8,img9,img11,img12,img13,img14);
    images = cat(4,img2,img3,img5,img6,img7,img8,img9,img11,img12,img13,img14);

    %The images we used for training
    array= {img1,img2,img3,img4,img5,img6,img7,img8};

    %Generate the background
    back = background(images);

    %Display steps of image processing
    %figure()
    %imagesc(back)
    %disp(size(back))
    %figure()
    %n = normalise(img8);
    %disp(size(n))
    %imagesc(n);
    %figure()
    %imagesc(abs(n-back));

    %Generate a black feature vector for the training
    features = zeros(1,4);
    count = 1;
    [asdf, n] = size(array);

    %figure()
    %imagesc(array{1})

    %Start training process for the images
    for i = 1:n
        %Our thresholding function that uses the image and the background
        %to return the detected objects
        cells = thresholding(array{i},back);

        %figure()
        %imagesc(array{i});

        [nr,asdf] = size(cells);

        %Create feature vectors for each object
        for j = 1:nr
            %Our properties function takes black and white object and
            %original image to return the four features of our objects
            p = myproperties(cells{j},array{i});

            features(count,:) = p;
            count = count + 1;

            %figure()
            %imagesc(cells{j});
        end
    end

    [N,DIM] = size(features);

    %True labels of the objects for training
    labels =
[8,6,7,9,9,10,10,7,4,7,1,5,8,9,6,7,4,6,10,6,9,8,10,2,7,5,1,3,8,4,6,7,4,10,4,8,10,2,3,1,2,6,2,8,4,5,
8,9,7,9,1,6,9,6,7,10,9,8,4,4,8,1,6,9,8,6,7,10,4,5,3,7,4,7,1,5,6,9,8,2,10,2,5,1,4,5,7,3,7,7,4,6,8,2,
10,7,4,3,1,7,7,5,6,4];
        %6,2,7,10,8,9,7,1,7,3,8,2,6,2,5];

    %Build model for testing
    [M,I,A] = buildmodel(DIM,features,N,10,labels);
end


function b = background(images)

    %Function we created generate background from a set of images
    array = double(images);
    s = size(images,4);

    %Normalise the images
    for i = 1:s
        array(:,:,:,i) = normalise(images(:,:,:,i));
    end

    %Median filtering of the normalised images
    b = median(array,4);
end


function norm = normalise(img)

    %Function we created to normalise an image

    %Separate the three colours
    red = img(:,:,1);
    green = img(:,:,2);
    blue = img(:,:,3);

 %   disp(red(1,1))
 %   disp(green(1,1))
 %   disp(blue(1,1))

    %Normalise each colour
    nred = (double(red))./(double(red+green+blue));
    ngreen = (double(green))./(double(red+green+blue));
    nblue = (double(blue))./(double(red+green+blue));

    %Reform the image
    norm = cat(3,nred,ngreen,nblue);

end


function cells = thresholding(img,back)
```

```matlab
%Function we created to extract individual objects from an image using
%the background

%Performed background subtraction
aux = abs(normalise(img)-back);
[R,C,d] = size(aux);

%figure()
%imagesc(img)
%figure()
%imagesc(aux)
%figure()
%imagesc(back)

%Split image into 3 colours
red = aux(:,:,1);
green = aux(:,:,2);
blue = aux(:,:,3);

%myhist(red,1)
%figure()
%histogram(red)

%Initialise the black and white image
new = zeros(R,C);

%Use thresholds we found by experimenting to identify objects from the
%background
for i = 1:R
    for j = 1:C
            if ((red(i,j)>0.053) | (green(i,j)>0.055) | (blue(i,j)>0.051))
                new(i,j) = 1;
            end
    end
end

%We tried using the bwmorph function for processing
%new = bwmorph(new,'dilate',1);

%figure()
%imagesc(new)
%figure()
%imagesc(img)

%Removed small areas that are noise
blob = bwareaopen(new,225);
%figure();
%imagesc(blob);

%Label each individual object
[labels,nr] = bwlabel(blob,8);
%figure();
%imagesc(labels)

cells = cell(nr,1);

%Create the cell array of objects that the whole function returns
for k = 1:nr
    temp = blob;
    for i = 1:R
        for j = 1:C
```

```
                if (labels(i,j) ~= k)
                    temp(i,j) = 0;

                end
            end
        end
        cells{k,1} = temp;
    end

end


function vec = myproperties(Image,Orig)

    area = bwarea(Image);
    perim = bwarea(bwperim(Image,4));

    % compactness
    compactness = perim*perim/(4*pi*area);

    % get scale-normalized complex central moments
    c11 = complexmoment(Image,1,1) / (area^2);
    c20 = complexmoment(Image,2,0) / (area^2);
    c30 = complexmoment(Image,3,0) / (area^2.5);
    c21 = complexmoment(Image,2,1) / (area^2.5);
    c12 = complexmoment(Image,1,2) / (area^2.5);

    % get invariants, scaled to [-1,1] range
    ci1 = real(c11);
    ci2 = real(1000*c21*c12);
    tmp = c20*c12*c12;
    ci3 = 10000*real(tmp);
    ci4 = 10000*imag(tmp);
    tmp = c30*c12*c12*c12;
    ci5 = 1000000*real(tmp);
    ci6 = 1000000*imag(tmp);


    [R,C] = size(Image);
    b = [1,255];

    %Create a feature representing the colour blue that we wanted to add to
    %training
    for x = 1:R
        for y = 1:C
            if (Image(x,y) ~= 0)
                bb = [double(Orig(x,y,1))];
                b = cat(2,b,bb);
            end
         end
    end

    blu = median(b);


    %vec = [compactness,ci1,ci2,ci3,ci4,ci5,ci6];
    vec = [compactness,perim,ci1,blu] ;

end


function test(M,I,A,back,string)
```

```
    %Read the test image and obtain the objects from it
    img = imread(string);
    cells = thresholding(img,back);

    %Show the image to compare to the location of objects
    figure()
    imagesc(img);

    [nr,asdf] = size(cells);
    total = 0;

    %Classify each object
    for j = 1:nr
        %Obtain the features of an object
        v = myproperties(cells{j},img);
        %Show which object is being classified
        figure()
        imagesc(cells{j});
        %Obtain the class and value of the object
        temp = classify(v,10,M,I,4,A);
        total = total + obj(temp);
    end

    disp(sprintf('The total value is %dp',total))

end


function val = obj(nr)

    %Takes the class of the objects to print its name and return its
    %value

    val = 0;
    if (nr == 1)
        disp('1 Pound Coin')
        val = 100;
    elseif (nr == 2)
        disp('2 Pound Coin')
        val = 200;
    elseif (nr == 3)
        disp('50p Coin')
        val = 50;
    elseif (nr == 4)
        disp('20p Coin')
        val = 20;
    elseif (nr == 5)
        disp('5p Coin')
        val = 5;
    elseif (nr == 6)
        disp('75p Small H Washer')
        val = 75;
    elseif (nr == 7)
        disp('25p Large H Washer')
        val = 25;
    elseif (nr == 8)
        disp('2p Angle Bracket')
        val = 2;
    elseif (nr == 9)
        disp('No Value Battery')
    elseif (nr == 10)
```

```
        disp('No Value Nut')
    end
end


function thehist = myhist(theimage,show)

  % set up bin edges for histogram
  edges = zeros(20,1);
  a = 0.05
  b = 0
  for i = 1:20
      edges(i) = b
      b = b + a
  end


  [R,C] = size(theimage);
  imagevec = reshape(theimage,1,R*C);      % turn image into long array
  thehist = histc(imagevec,edges)';        % do histogram
  if show > 0
      figure(show)
      clf
      pause(0.1)
      plot(edges,thehist)
  end
end
```