Introduction to Vision and Robotics:

Assessed Practical 1: Coin Counter

by
Levi Fussell (s1408726) & Daniel Clemente (s1333465)

Distribution of Work:

Levi: 60%
Daniel: 40%

Levi was responsible for the coding of the project. Daniel was responsible for managing the report. Both participated in testing, editing and ideas, as well as helped each other out with their respective portions.

Introduction:

In this assignment we were given a set of nine simple images and five harder images for classification. (Later the task was changed so the harder images were removed) The goal was to segment the image into the various objects (nuts, large and small washers, batteries, angle brackets, and coins, spread out across a table), classify them. The task was split into steps:

1. Segmentation: Performing pre-processing such as removing the background from a given image, and applying various functions in order to simplify the image. Then using the processed image, distinguish the individual objects.

2. Classification: Taking these segmented images of individual objects, using the features of each image and training in order to obtain a system that, given new segmented objects, will classify them correctly.

The final system had a classification accuracy of 75%, the details of our methods are described below.

IMAGE SEGMENTATION:

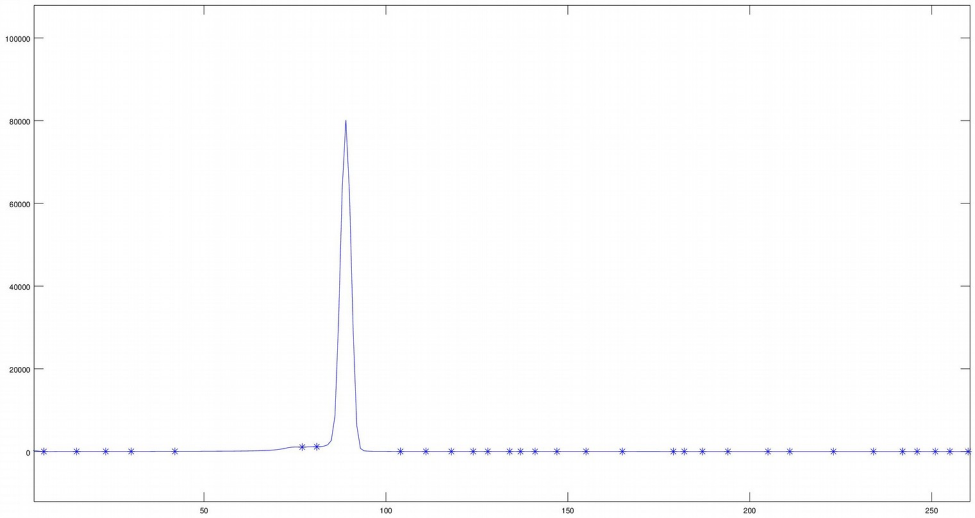
The whole process of segmenting the image was divided into four distinct steps: *Edge Detection, Pre-Processing, Clustering, Minimum-Bounding Box.*

Edge-Detection:

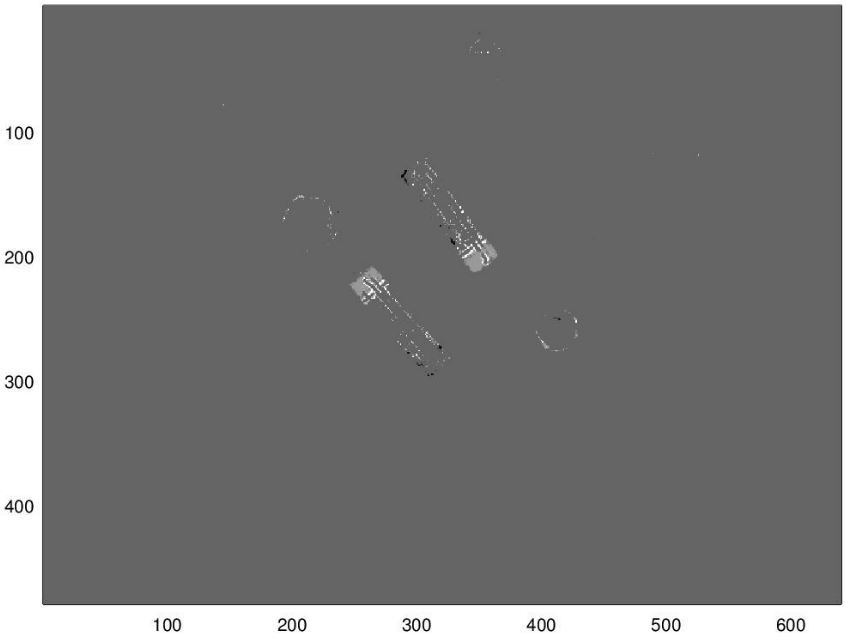
Although it's a simple process for the human eye to simply segment, remove and distinguish objects within these images, this does not mean this process is computationally trivial. Before objects can be analysed for *what* they are, they first have to be distinguished from the background. The coursework description suggested an approach of background removal by taking the median pixel values of all the images to construct an artificial background. Although this approach produced an effective background, it is not a general enough method for smaller datasets and inconsistent backgrounds. Instead, an approach of removing the background by using a single image and a general algorithm that would work for many basic backgrounds would be prefered.

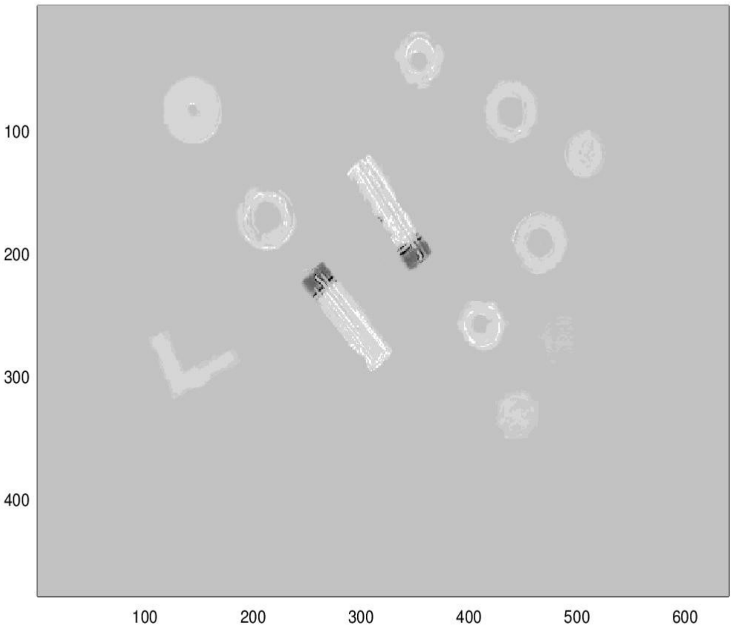
Thresholding:

As discussed in class, the histogram representation of an image is a powerful and informative graph to present the distribution of pixel values in an image. Usually represented by a series of spikes, finding the minimum of these spikes (after Gaussian smoothing), gives colour threshold boundaries for an image.

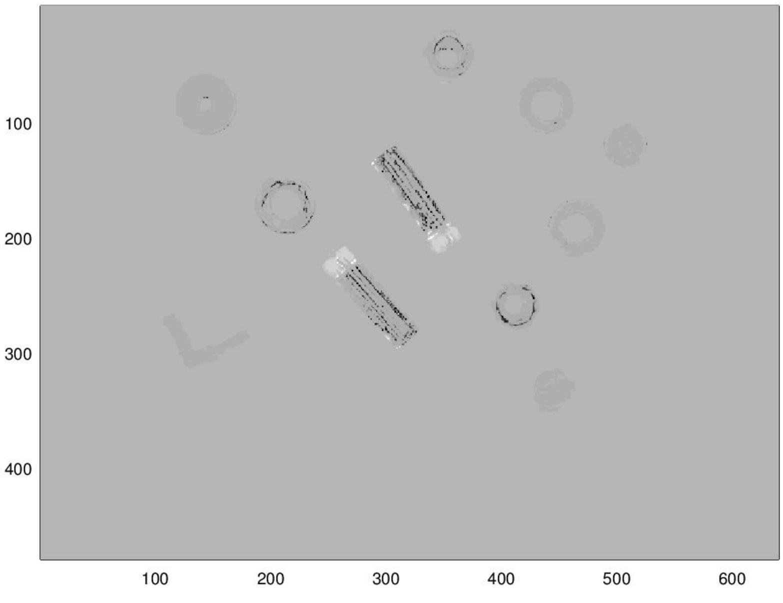


An image was first normalised to remove shadows and inconsistencies in the image (*image_segmentation.m* ln.8-19), then each red, green and blue matrix was converted to a smoothed histogram and thresholds were defined by the minimums (*image_segmentation.m* ln.22-33, *edge_detection.m*). Notice in the the histogram above there are a large number of thresholds; this is intentional, as it allows the image to maintain important details without over-generalising and blending the background and foreground. After applying these threshold boundaries to the image:



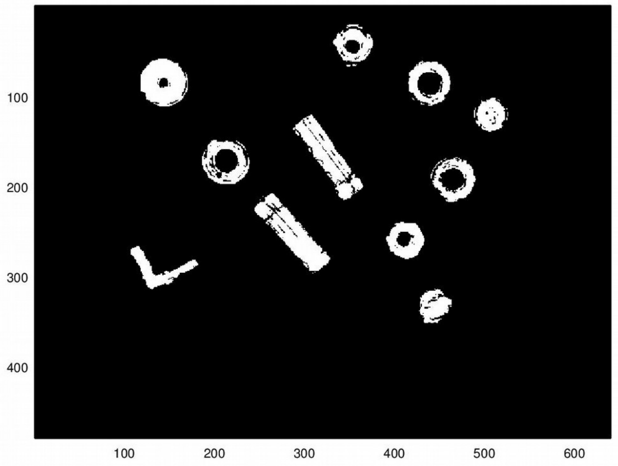


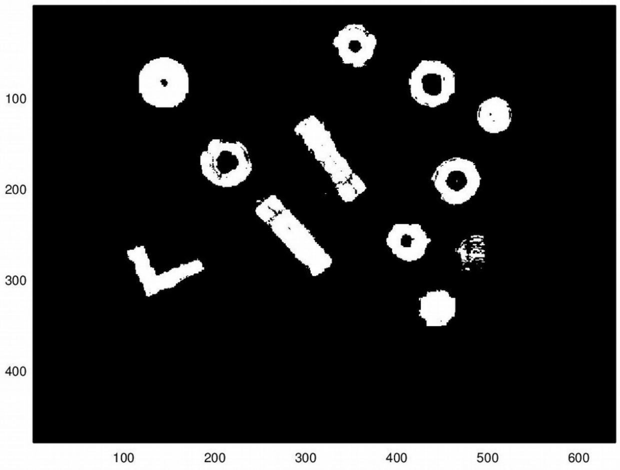
100
200
300
400
100
200
300
400
500
600

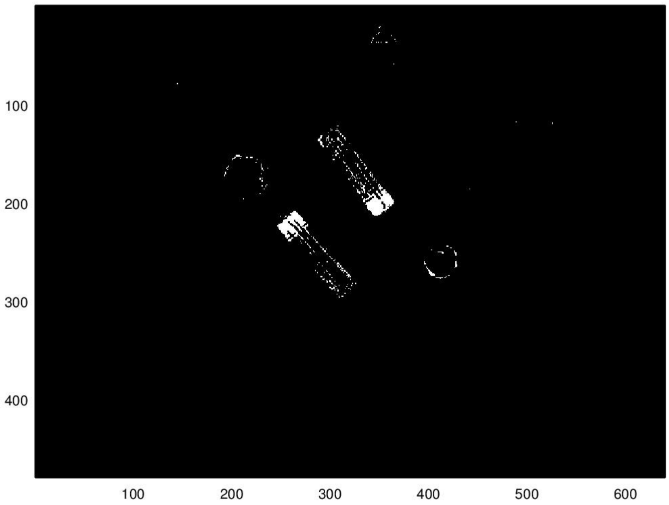


100
200
300
400
100
200
300
400
500
600

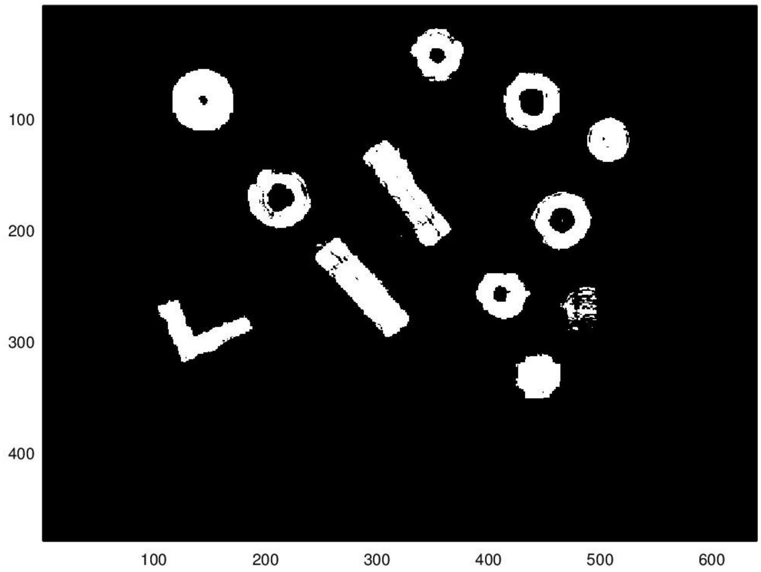
It is now clear that the image has been considerably simplified in its range of colour values. An important thing to note is that the background in all three threshold images has been converted to a constant colour and the noise has been removed. At this point it is easy to remove the background by finding the mode colour in all three images that occurs the most and using that as the new threshold to create three binary images:



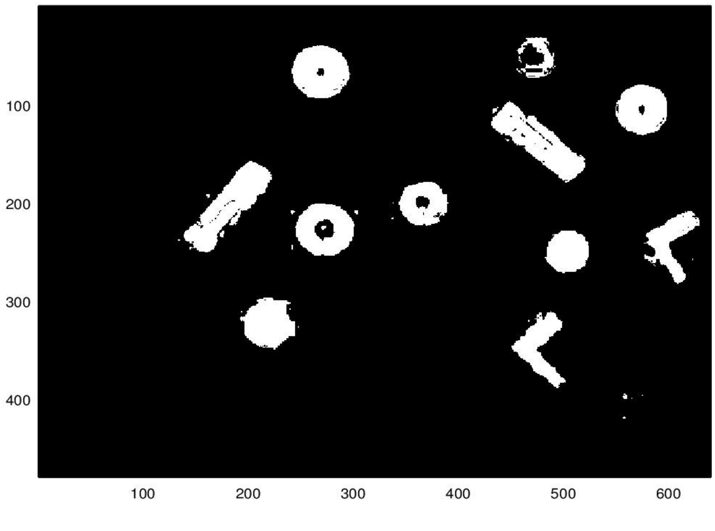


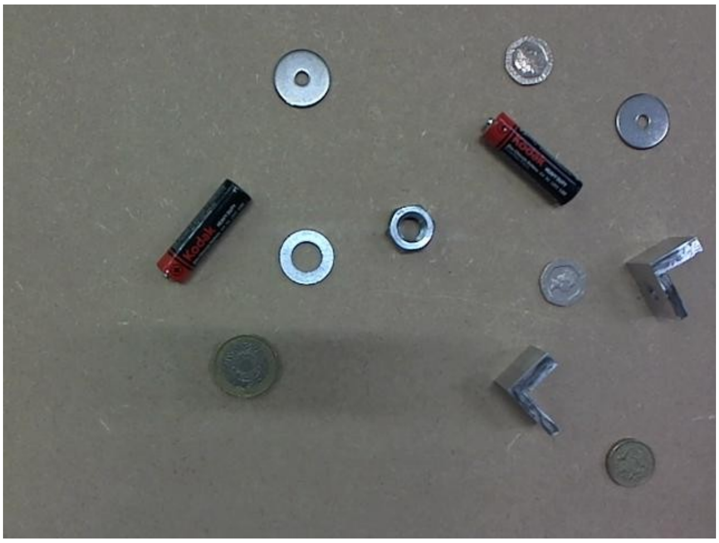


It is important to note that this thresholding process has to be done on all three colour types: red, green and blue. This can be proven by observing the above binary images and seeing how different detail is selected by each colour type; if only the green image was selected, only the batteries would be identified; if the blue path was chosen, most of the image would be identified, but without the red path the coin in the bottom right hand side would be lost in the thresholds. Hence all three images are used to give us three "attempts" at defining thresholds correctly. Merging the binary images with an OR statement gives us a single image matrix of all colour types:



At this point it may seem that this is an almost perfect result and the next steps would be to cluster, feature-select, etc. and we have our individual image data. The above result is a very ideal example of the process. Take a look at the example below on another image:



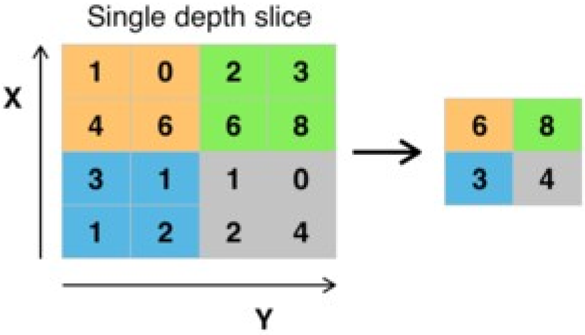


In this example, the image has done well to do a binary segmentation, but in the process it has kept some noise around the washers and batteries and also completely removed the one pound coin in the bottom right. It is now necessary to obtain more information and remove this noise.

Preprocessing:

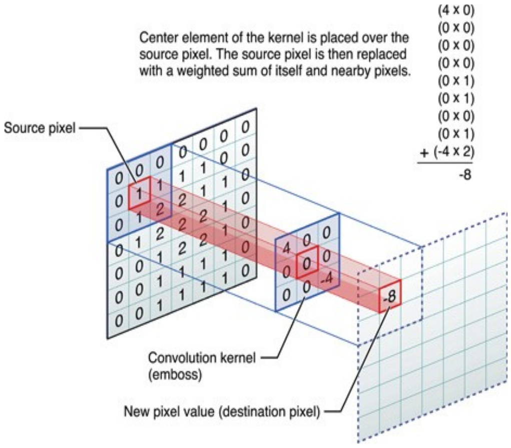
The inspiration for the preprocessing came from an approach used in a modern Machine Learning model, Convoluted Neural Networks (CNN). These complex Neural Networks use multiple layers of convolution matrices and max-pooling matrices to bisect significant features from images, adjusting the weights of these matrices for the best output. Implementing a CNN is complex and requires larger datasets, therefore the CNN methods of merging layers of max-pooling and convolutions are used but with pre-set matrices.

Max-pooling an image gives two advantages; it reduces the size ratio of the image making it easier to perform algorithms (i.e. space searching during clustering), reduces the noise in the image, and emphasises unique features by selecting pixels with the largest (and therefore most significant) value. The disadvantage to max-pooling is that it can be computationally expensive if the image is large and the pool size is small (the smallest case being a pool size of 1 and therefore having to traverse every pixel in the image). (max_pooling.*m*)



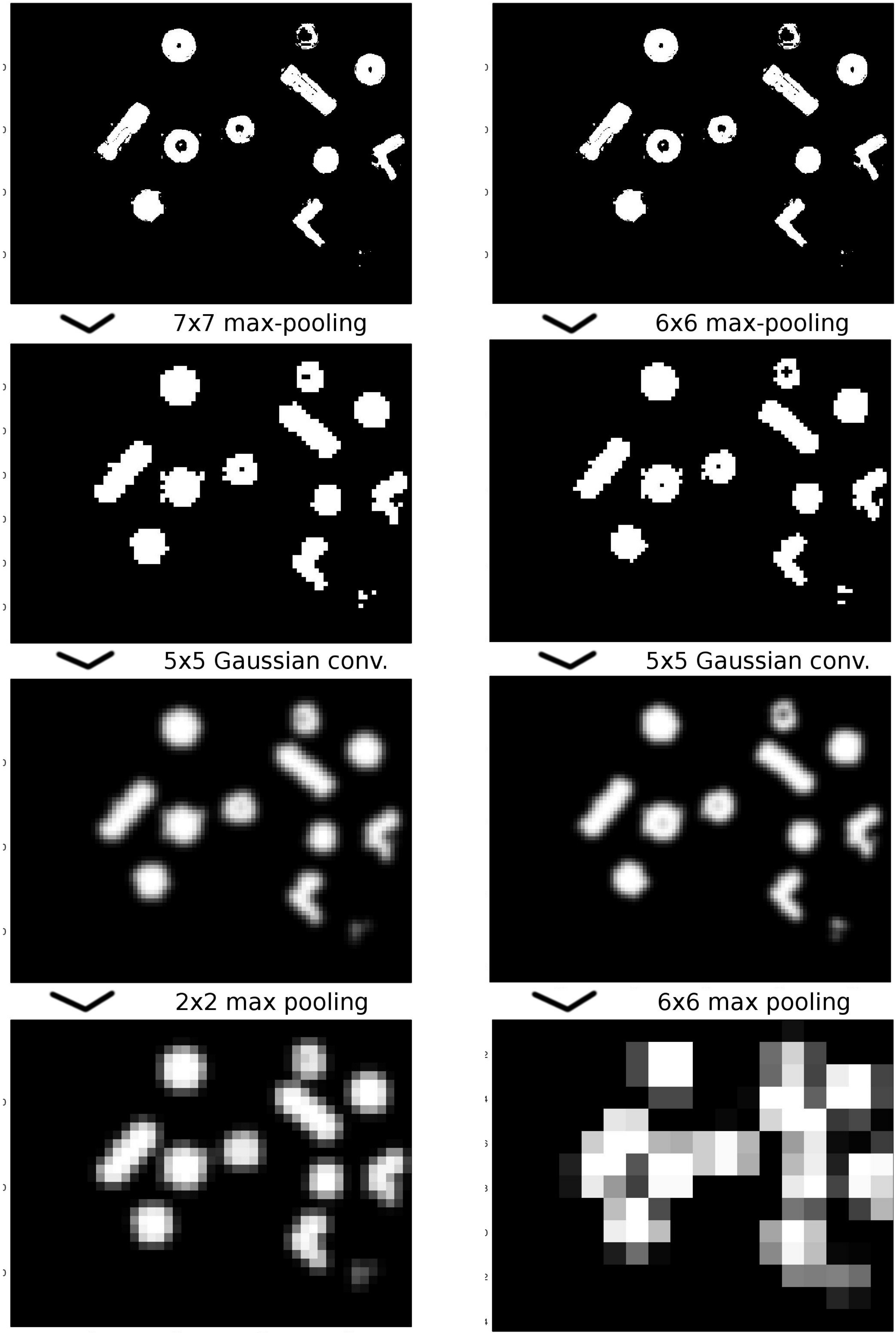
source: https://upload.wikimedia.org/wikipedia/commons/thumb/e/e9/Max_pooling.png/314px-Max_pooling.png

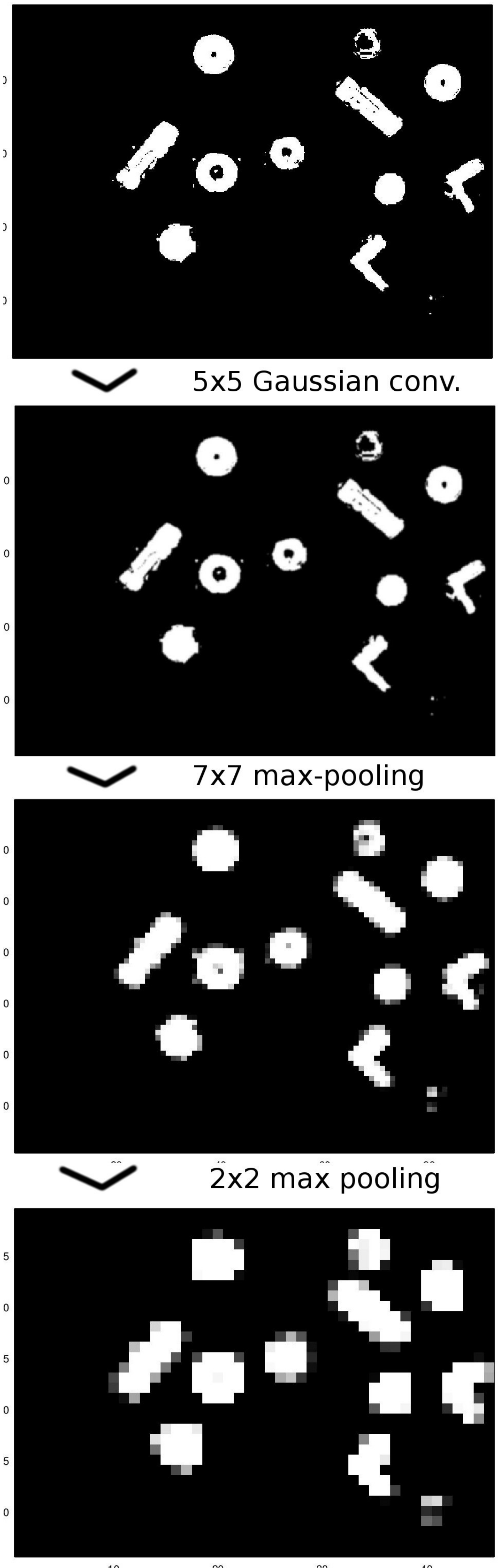
A convolution is used to smooth an image and act as another noise filter, removing any obvious colour differences in pixels within a localised area. This method was used to smooth the histograms for threshold analysis earlier.


Center element of the kernel is placed over the source pixel. The source pixel is then replaced with a weighted sum of itself and nearby pixels.
(4 x 0)
(0 x 0)
(0 x 0)
(0 x 0)
(0 x 1)
(0 x 1)
(0 x 0)
(0 x 1)
+ (-4 x 2)
-8
Source pixel
Convolution kernel (emboss)
New pixel value (destination pixel)
-8

Source:https://developer.apple.com/library/content/documentation/Performance/Conceptual/vImage/ConvolutionOperations/ConvolutionOperations.html

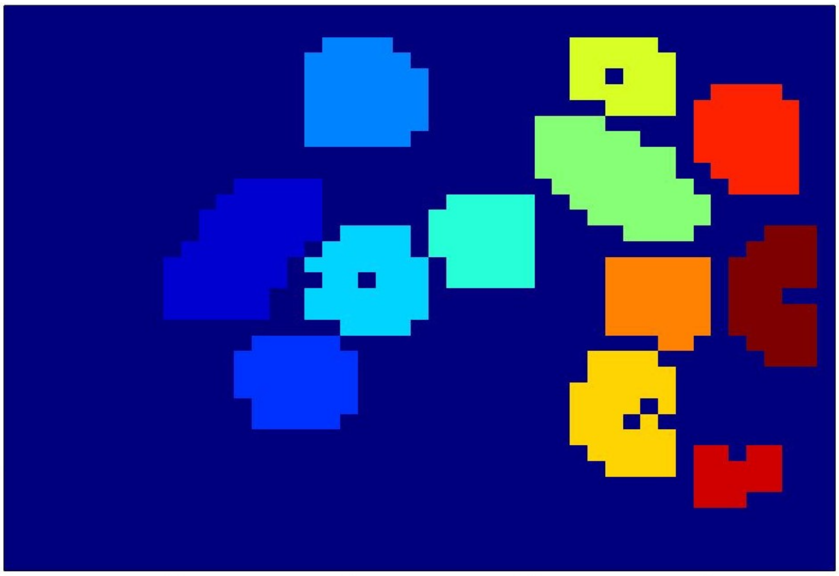
Both these filtering algorithms can be chained together in a variety of orders, with their parameters set to many different values (hence why using them in Neural Networks makes it easier for the network to determine the values instead of a human). Experimenting with altering the values is required to get the best results from these filters. Experimentation showed that two max-poolings and one convolution was generally the best approach due to computation and time constraints (see *analysis* at document end). A few filter orders are shown below; these demonstrate how much variation there is in filter choice and sequence:


7x7 max-pooling
6x6 max-pooling
5x5 Gaussian conv.
5x5 Gaussian conv.
2x2 max pooling
6x6 max pooling


5x5 Gaussian conv.
7x7 max-pooling
2x2 max pooling

Link Clustering:

Once a series of blobs has been created from the image, it is necessary to extract these blobs as separate entities so that features can be individually selected from each blob. The obvious method for extracting these blobs is via a clustering algorithm – the basic algorithms of which are k-means clustering or mean-shift clustering. Both these clusters have their drawbacks. K-means requires knowing the cluster count beforehand, and mean-shift can be computationally expensive. Neither utilises an important aspect of the blobs in the image: due to pre-processing, the image complexity (aspect ratio) has been greatly reduced (via max_pooling). Blobs are now composed of pixels that are directly neighboured to one another, simplifying the problem:



This allows for an algorithm that clusters in linear time, such that each pixel is only processed once by the algorithm. This link clustering algorithm does come with a drawback. Because link clustering relies on blobs being composed of neighboured pixels, there are cases where the algorithm can fail:

1. A blob overlaps with another blob in the image, resulting in the algorithm clustering the two blobs as a single blob.
2. A blob is not extracted correctly and has 'breaks' in its pixel composition, resulting in one blob being clustered as two blobs.

Both these cases can be avoided by performing the correct pre-processing of the image so that the blobs are segmented appropriately. Of course there are still cases where bad lighting, shadows and similarity between object and background can cause blobs to be incorrectly segmented. Avoiding these scenarios would require more powerful pre-processing methods and possibly an advancement on link clustering to incorporate a degree of 'error freedom' when choosing neighbours. This was neither analysed nor implemented due to time constraints, but would be an interesting area to explore further. (*linkclustering.m*)

Pseudo code for *link clustering* below:

```
while (true) {
    if queue == empty {
        a = get next non-zero value in binary image matrix M
        if a == NULL
            exit()
        else
            a.class ← nextClass
            queue.add(a)
            M{a} ← 0
            nextClass++
    }
    else {
        b ← queue.pop()
        {C} ← horiz. and vert. neighbours to b in matrix M
        foreach c in {C} {
            c.class ← b.class
            queue.add(c)
            M{c} ← 0
        }
    }
}
```

Minimum-Bounding Box:

The hardest part of the segmentation has been performed once the individual items in the image have been clustered into unique blobs. The next step involves either analysing each blob based on its relative location in the image (using moments, etc), or to separate each blob into a minimum-bounding image to make it easier to calculate features such as compactness, area, etc. The latter method was chosen for two reasons:

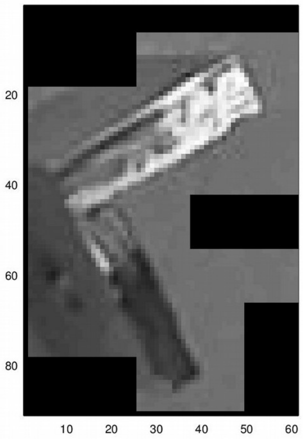
1. It allows for easier analysis of features such as perimeter, area, compactness and pixel counting.

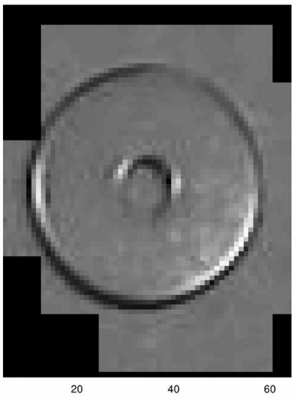
2. If a box is drawn around the blob and cut from the original image (not the blob image), some information may be restored that was lost in the segmenting of the image (i.e. colours, shapes, holes in objects). This is useful for classification.

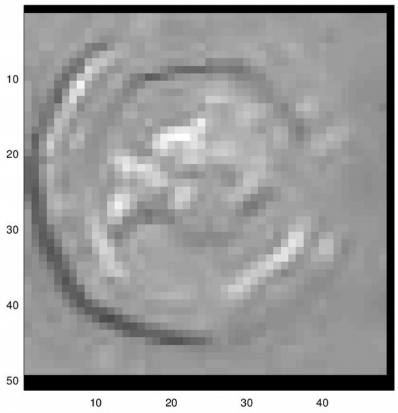
The algorithm for the minimum bounding box is simplified and, with more time, could be improved by modern approaches. This baseline method works well for this task:

```
data_gain ← MAX_INT
box_offset ← 5
while(data_gain > 0) {
    b_left ← cluster.centre.x - box_offset
    b_right ← cluster.centre.x + box_offset
    mb_box = image(b_left:b_right, :)
    data_gain ← num new non-zero pixels in new box
}
data_gain ← MAX_INT
box_offset ← 5
while(data_gain > 0) {
    b_top ← cluster.centre.y - box_offset
    b_bottom ← cluster.centre.y + box_offset
    mb_box = image(b_left:b_right, b_top:b_bottom)
    data_gain ← num *new* non-zero pixels in new box
}
return mb_box
```

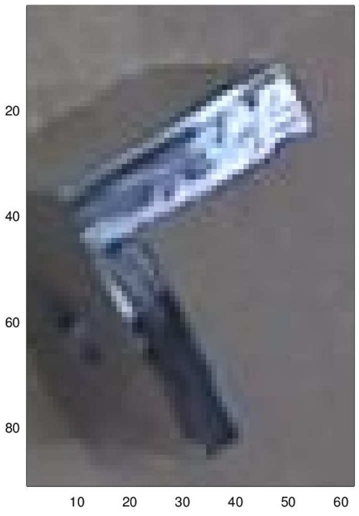
Here are some examples of segmented images without using only the blob to get the image area (the blob has been cropped from the larger image for convenience):

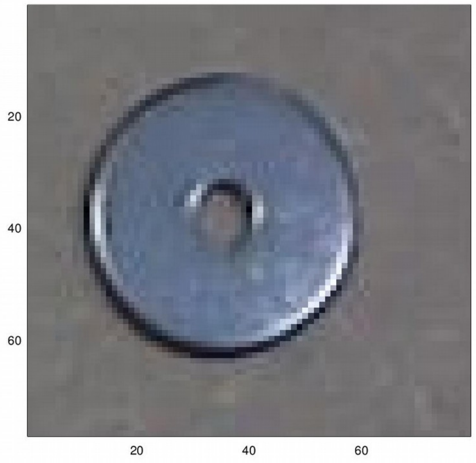






Note that the image on the right and middle are the only ones that *fully* preserved the entire image. This can be fixed by applying the minimum-bounding box algorithm to the blob area on the original image:



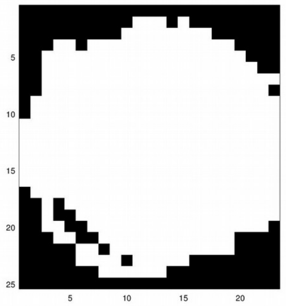


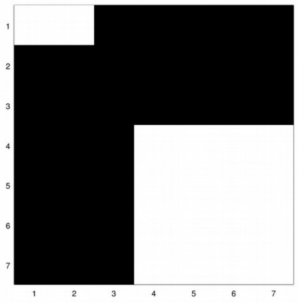
Segmentation Analysis:

Class	Ratio Correctly Segmented
One Pound	4/12
Two Pound	4/7
50p	4/4
20p	5/10
5p	6/7
Washer Small Hole	8/12
Washer Large Hole	10/12
Angle Bracket	11/11
AAA Battery	10/10
Nut	9/9
TOTAL	71/94 = 76.0% accuracy

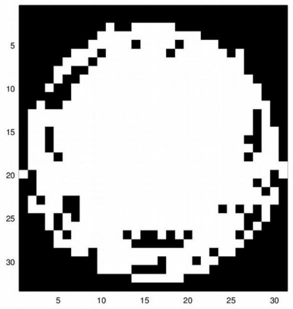
Below is the best and worst segmentation of each class:

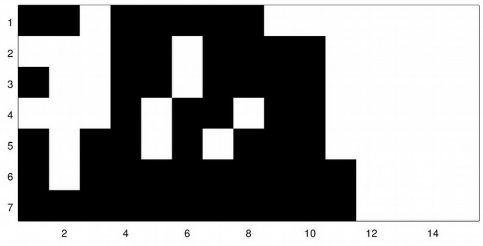
BEST **WORST**



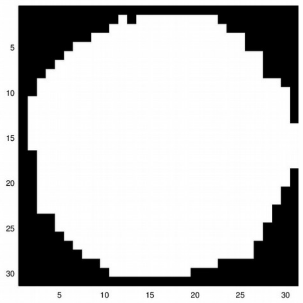


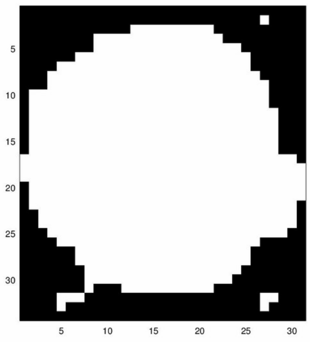
ONE POUND



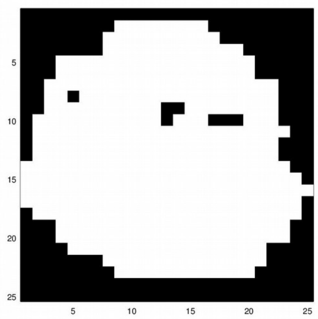


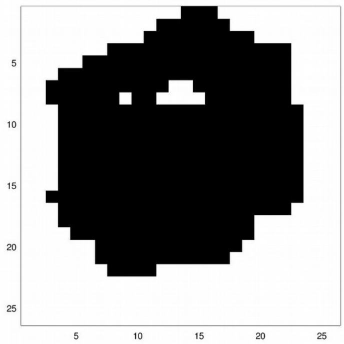
TWO POUND



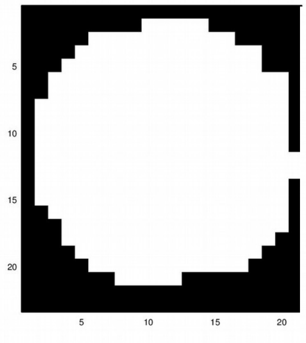


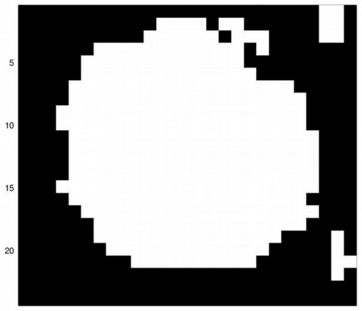
50P



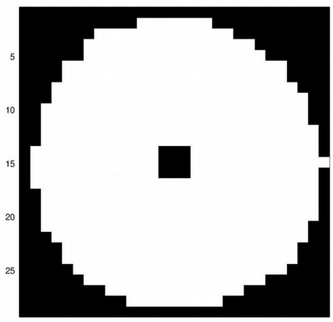


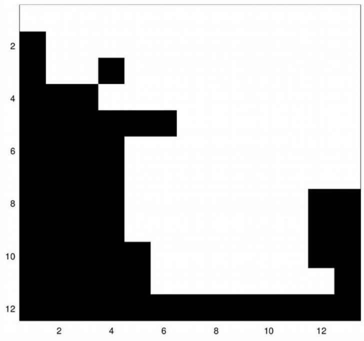
20P



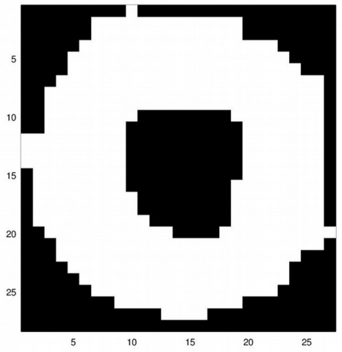


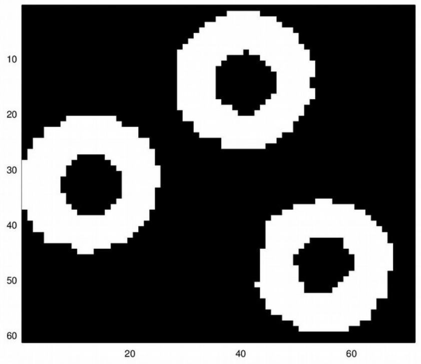
5P



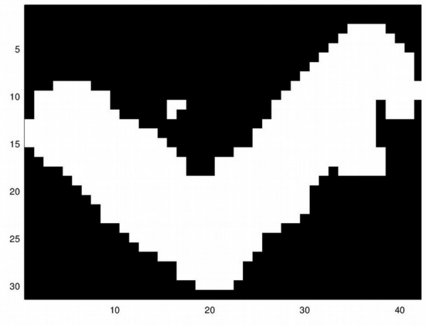


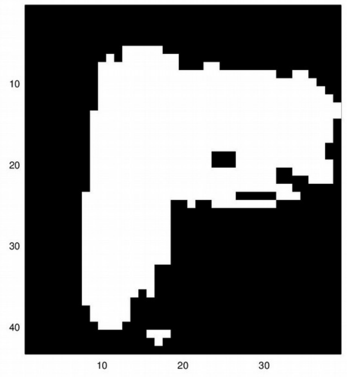
WASHER SMALL HOLE



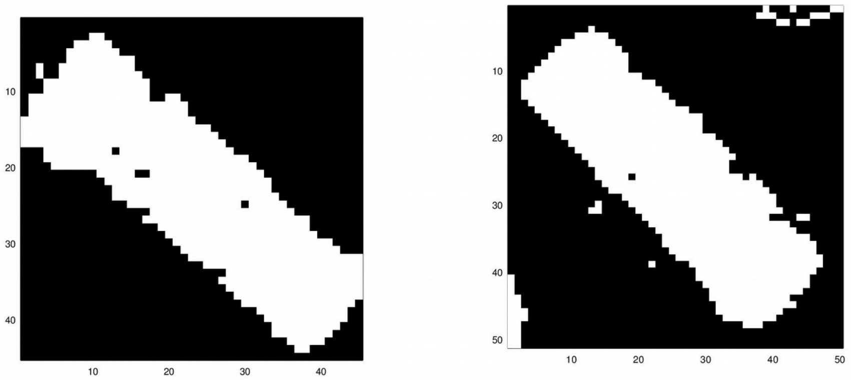


WASHER LARGE HOLE

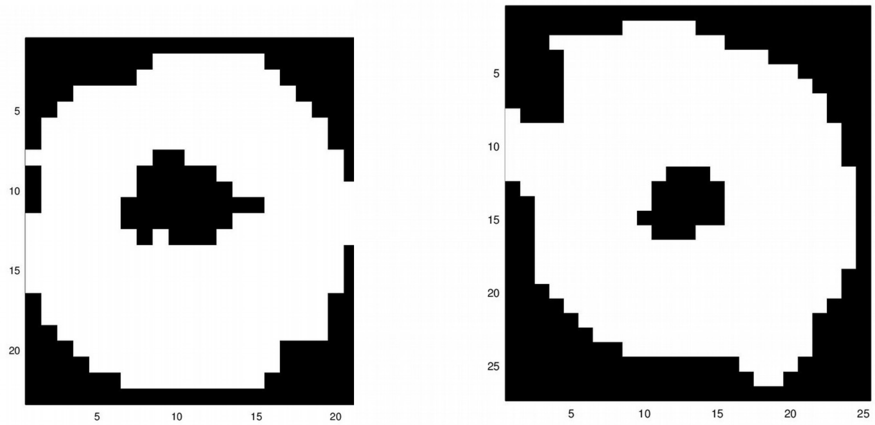




ANGLE BRACKET



AAA BATTERY



NUT

Now that the individual objects can be segmented from a larger image and bounded by a minimum-box, the next step is to analyse aspects of these sub-images to uniquely identify the objects within them.

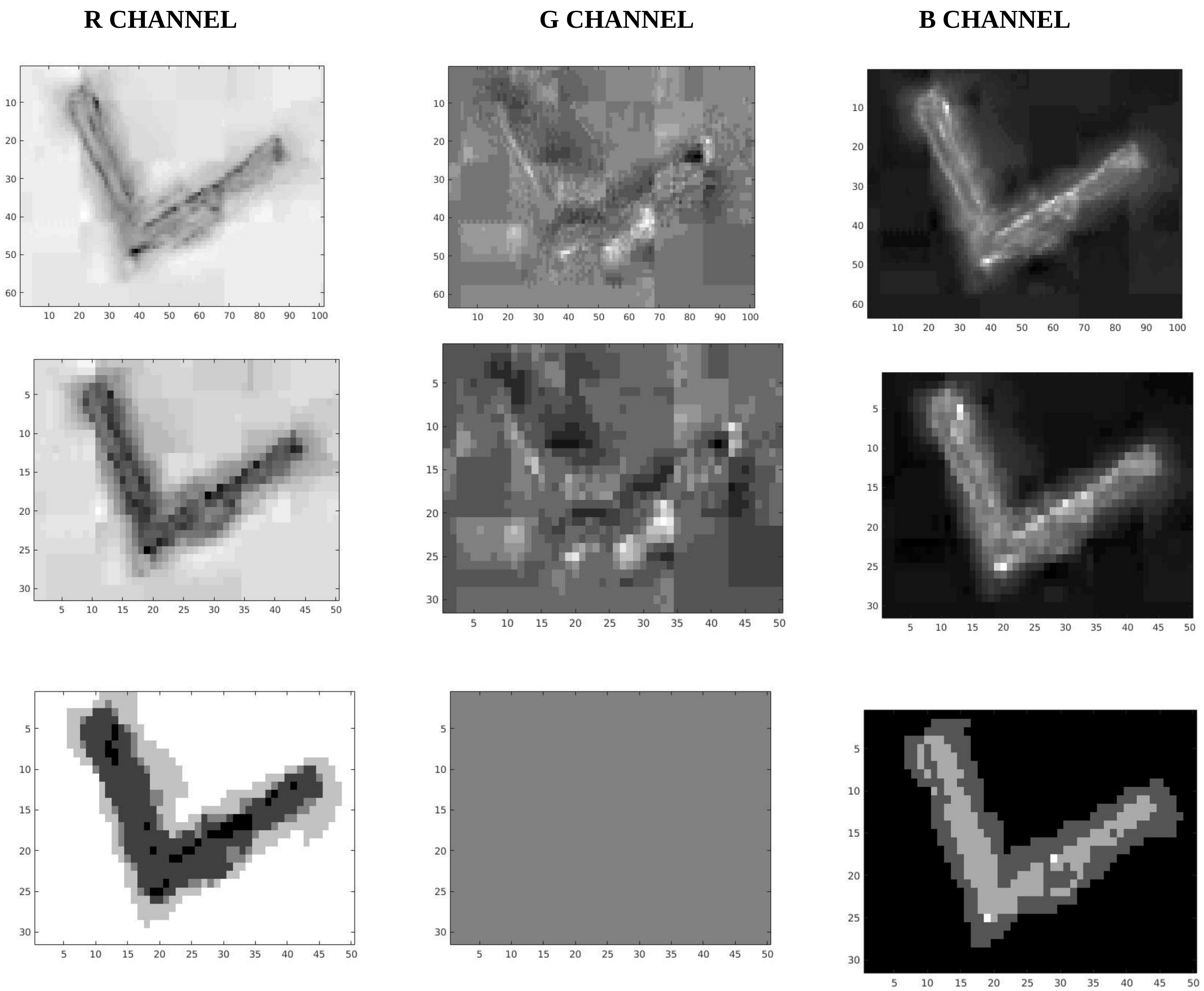
CLASSIFICATION:

Classification was divided into four sections, *Preprocessing, Feature Selection, Training Classification* and *Model Building.*

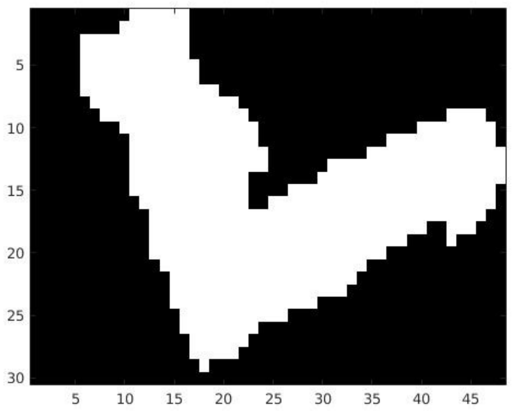
Preprocessing:

Features cannot be selected from an object image without first converting it to a higher-resolution binary image. Using the binary image from the segmentation step would not provide a high enough level of detail, due to the nature of max-pooling and convolution filtering. The process of segmenting the image from the background and then converting it to binary is almost the same as the process used to segment the objects from the larger main background, but excludes the clustering portion of the process.

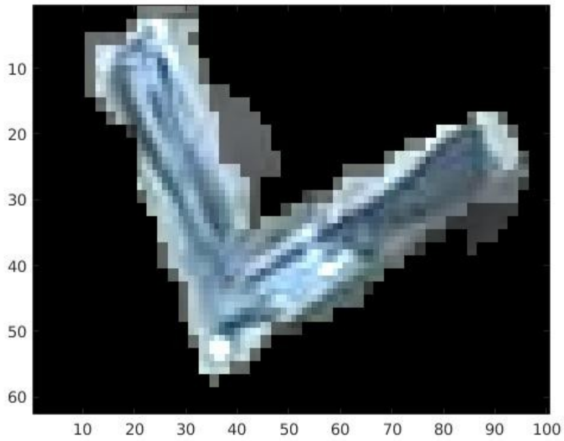
First the images are converted to normalised RGB values. Each RGB matrix has a threshold applied to it such that the object is segmented into approximately ten to fifteen different colour bins; the mode of these colour bins (the colour that is most frequent) is used as a final threshold to remove the background and convert the image to binary. Noise is removed from the image by doing a single max-pooling of size 2 (*get_object_feature_vector.m* ln.7-33). Additionally, this halves the size of the image, making it easier to work with computationally. Now that the segmented object image has been converted to a simpler binary format, the next process is to extract features.



Convert R, G, and B channels to binary and then merge with binary OR:



Overlay the binary image onto the original segmented image to obtain the removed image object:

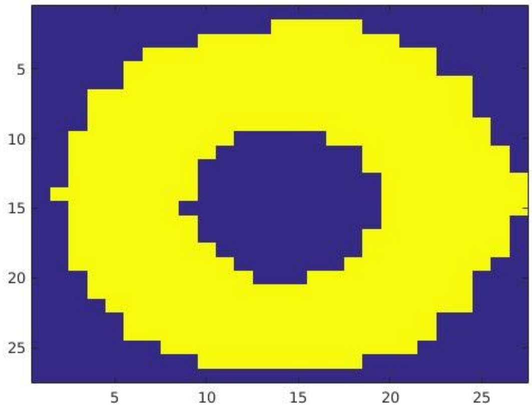


Choosing Features:

Before arbitrarily choosing a random set of features to train a classifier on, it is important to first analyse the aspects of the different objects and determine what the minimum number of features required to classify the objects could be (without using moments in this case, because this does not allow us to *understand* the model and the features it is being supplied – at least with our current level of understanding). Here is a layout of the objects' analysis:

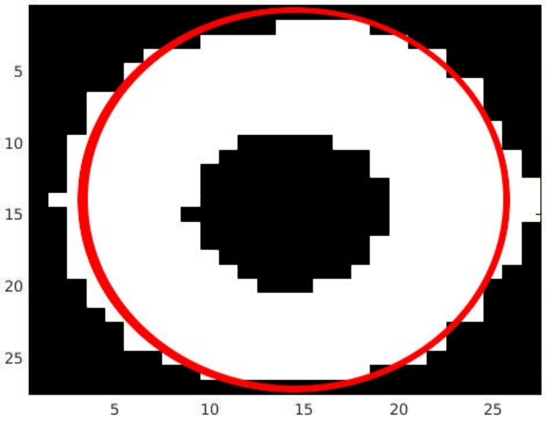
Features					
Class	has_hole	**size_hole**	is_circle	**size_circle**	**colour**
One Pound			x	med	gold
Two Pound			x	large	gold/silver
50p			x	large	silver
20p			x	med	silver
5p			x	small	silver
Washer Small Hole	x	small	x	medium	silver
Washer Large Hole	x	medium	x	medium	silver
Angle Bracket					silver
AAA Battery					black
Nut	x	small	x	small	silver

If each bold feature above was *perfectly* identified the classification between the objects would be simple; each object would contain its own unique vector of values. The problem is, accurately measuring these features is almost impossible, and relies heavily on the object being segmented and bounded accurately.

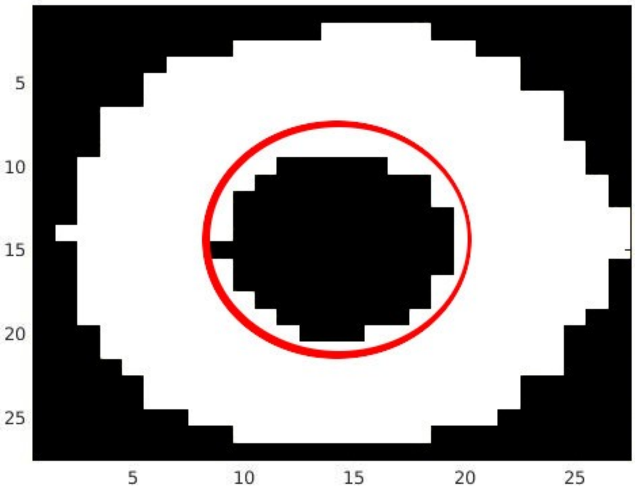


Under the assumption that the segmented object is centred in the sub-image with a minimum-bounding box around it, the following feature algorithms were formulated:

circle comparison: percentage of all white pixels that are within a maximum circle drawn on the image (*circle_comparison.m)*



hole-iness: percentage of black pixels within a half-radius maximum circle (*empty_pixels_in_circle.m*)



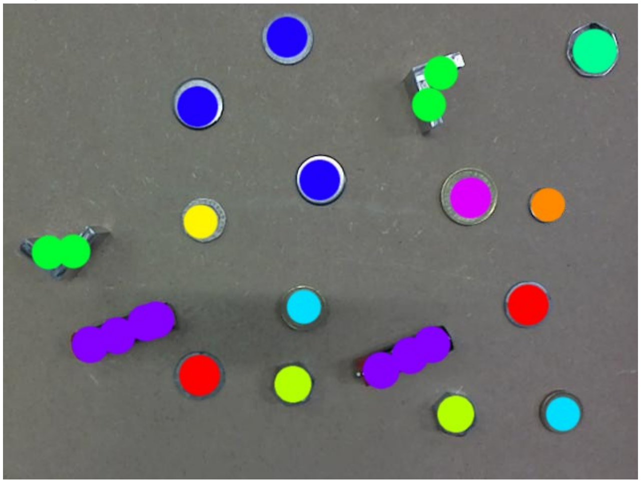
colour hash: *r* + (g + 1.0) ^ 2 + (b + 2.0) ^ 3 , where r,g,b ϵ [0, 1] and are red, green and blue colour values (*get_object_feature_vector.m, ln.63*)

(Code for construction of the feature vector: *get_object_feature_vector.m, ln.53-69)*

Classifying Objects:

A classifier cannot be trained until the segmented objects from each image have been assigned to their respective class. This would be an easier task if images were pre-segmented and could be labelled in the dataset – as this is not the case, a human classifier must be used. One option presented in the lectures is to implement a hand-labelling system into the training of the model, this method works, but requires the arduous task of someone determining the class of each segmented image. Instead the images can be directly designed to guide the classifier to the right class.

Each object class is assigned a specific RGB colour value which is drawn onto a duplicated 'training' version of the image at the object location, as shown below:



Once the initial image has been segmented, the same portion of the 'training' image is segmented and the colour values below are detected (*get_class_from_training_image.m*). The segmented object is assigned to the majority colour value class within the segmentation; if no class is found it is assumed to be a false segmentation:

- one pound: (0, 222, 255)
- two pound: (218, 2, 251)
- 50p: (0, 255, 151)
- 20p: (255, 246, 0)
- 5p: (255, 138, 0)
- washer small hole: (30, 0, 254)
- washer large hole: (254, 0, 0)
- angle bracket: (0, 255, 49)
- AAA battery: (132, 1, 255)
- nut: (180, 255, 0)
- *otherwise:* class = -1

The advantage of this method is that the model can be trained effectively without human interaction. This model becomes more convenient with larger datasets. The disadvantage of this method is that false segmentation cannot be filtered out by the computer. Human interaction has the benefit of removing edge-cases such as badly segmented images.

Feature Analysis:

Now that both the object class and object features have been obtained, some informative data can be collected to help build the classifier:

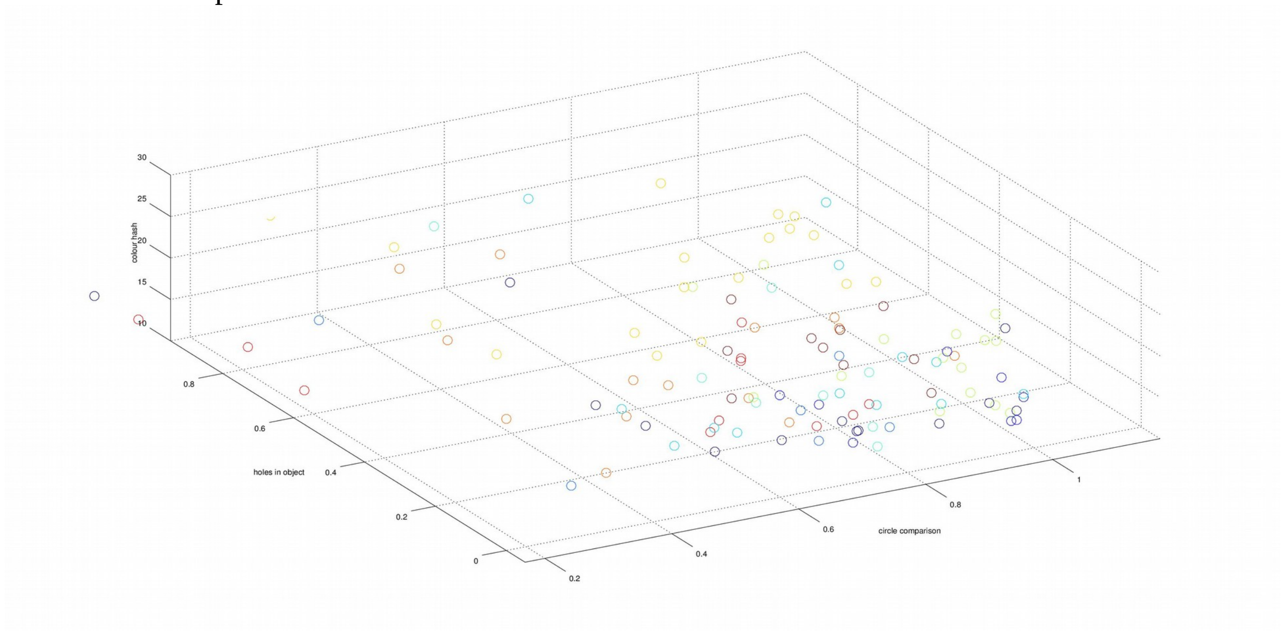
Mean Feature Evaluation of 94 segmented images (rounded to 2 digits)						
Class No.	**Class**	**Circle Comparison**	**Hole Value**	**Hashed Colour**	**Density**	**Compactness**
1	One Pound	0.60	0.51	15.	0.41	1.23
2	Two Pound	0.83	0.23	16.	0.41	3.07
3	50p	0.95	0.0055	16.	0.67	1.03
4	20p	0.80	0.42	20.	0.50	2.40
5	5p	0.96	0.011	20.	0.64	0.99
6	Washer Small Hole	0.91	0.13	22.	0.60	1.92
7	Washer Large Hole	0.86	0.59	20.	0.54	2.70
8	Angle Bracket	0.67	0.27	19.	0.38	2.72
9	AAA Battery	0.76	0.30	11.	0.35	2.31
10	Nut	0.89	0.33	17.	0.58	2.18

Looking at the above table, it is obvious that *hashed colour* is a strong indicator of an object’s type, especially for seperating the battery and silver objects. It is clear there are some errors concerning *hole value*. Error with the one pound is expected because it was the object with the worst segmentation score. The important thing to note is that the *hole value* difference between the Washer Large Hole and the Washer Small Hole is large, indicating that this feature can be used to distinguish between them. *Circle comparison*, is also very strong (apart from the one pound) and scores all objects that are circular a score of >0.80. Anything below this value can be safely classified as a battery or angle bracket (and to distinguish these two we easily use the *hashed colour*). Overall, the feature data appears separable for *some* classification improvement; it will be necessary to advance upon these features to help distinguish between harder cases such as the 20p and the Nut.

The features highlighted in green represent two features that were added near the end of the project after the final round of Naive Bayes (NB) classification on the Matlab model. Both these features together contributed to increasing the accuracy by approx. 5% and removing many edge cases in the confusion matrix. These features are discussed further on during the analysis segment.

Running the Naive Bayes Classifier:

Using a Naive Bayes Classifier was the simplest and fastest approach to implementing a classifier. Logistic regression was also considered in order to test if the feature space is linearly seperable. With a visualisation of the feature space below, it is clear that the data is not linearly separable without some implementation of kernel functions and/or a SVM:



(this feature space helps to show proof of the clustering of similar objects in the space. One aspect that cannot be seen from this angle is that the vertical axis (hash colour) linearly separates the battery object from the rest of the objects (because it is the only black object). This was as hypothesised earlier.)

Implementing the Naive Bayes was first done via a custom Matlab script and the results below were produced:

First Attempt - Confusion Matrix: (without using logs for multivariate)

	Actual										
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Predicted	**1**	0	0	0	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0	0
	4	3	2	1	3	2	2	1	3	3	3
	5	0	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	2	2	0	0	0
	7	0	0	0	0	0	0	0	0	0	0
	8	0	0	0	0	0	0	0	0	0	0
	9	0	0	0	0	0	0	0	0	0	0
	10	0	0	0	0	0	0	0	0	0	0

Second Attempt - Confusion Matrix: (using logs for multivariate to remove zero values)

	Actual										
Predicted		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	1	2	1	1	0	0	0	1	1	0	0
	2	0	0	0	1	0	0	0	0	0	0
	3	0	0	0	0	0	0	0	0	0	0
	4	0	0	0	0	0	0	0	0	0	0
	5	0	0	0	0	2	0	0	0	0	0
	6	0	1	0	0	0	2	0	0	0	0
	7	0	0	0	1	0	1	2	0	0	1
	8	1	0	0	1	0	0	0	2	0	1
	9	0	0	0	0	0	0	0	0	3	0
	10	0	0	0	0	0	0	0	0	0	1

After attempting to get the NB classification code to work without bugs, it turned out to be more effective to use the built-in Matlab NB classifier class. Using this function, the following convolution matrix was obtained:

	Actual										
Predicted		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	1	3	0	0	0	0	0	0	0	0	0
	2	1	0	0	0	0	0	1	0	1	0
	3	0	1	1	0	0	0	0	0	0	0
	4	0	0	0	1	0	1	0	0	0	0
	5	0	0	0	0	2	0	0	0	0	0
	6	0	1	0	1	0	3	0	0	0	0
	7	0	0	0	0	0	0	2	0	0	1
	8	1	0	0	0	0	0	0	3	1	0
	9	0	0	0	0	0	0	0	0	1	0
	10	0	0	0	0	0	0	1	0	0	2

The accuracy value for this model was 60.0%. Taking into account the 76% error value for the segmentation process which would propagate into the classification error, this level of accuracy is acceptable (see *baseline metric* further down). Improving the segmentation accuracy should therefore correspond to an improvement in classification accuracy.

Analysis of Accuracy with Different Convolution/Max-Pooling Filters

Following the above results for the chosen feature vector, it is clear that one strong way to improve accuracy results would be to reduce the error during segmentation. Changing the segmentation involves tweaking the values and sizes of the filters on each pre-processing layer and recording the accuracy results as below, choosing the structure with the highest accuracy:

<table>
<tr><th>Train Percentage</th><th>First Filter</th><th>Second Filter</th><th colspan="2">Third Filter</th><th>Accuracy</th></tr>
<tr><td>0.75</td><td>5x5 Gauss</td><td>6x6 max-pool</td><td colspan="2">2x2 max-pool</td><td>60.00%</td></tr>
<tr><td>0.75</td><td>7x7 max-pool</td><td>5x5 Gauss</td><td colspan="2">2x2 max-pool</td><td>43.33%</td></tr>
<tr><td>0.75</td><td>5x5 Gauss</td><td>3x3 max-pool</td><td>2x2 m-p</td><td>2x2 m-p</td><td>56.70%</td></tr>
</table>

Reporting the accuracy for a single training and testing instance proved inconsistent with multiple tests; the accuracy would never settle on a determined value due to the random nature of the training and testing data allocation. Additionally, the more badly segmented objects in the training set, the worse the model will perform on testing. To replace the random accuracy analysis, a cross-validation technique was used so that a broad range of models could be tested and trained on large datasets (this helps reduce noise from false positives).

30 bins were chosen arbitrarily, dividing the data into approximately 85% training and 15% testing for each of the 30 NB model iterations. The accuracy of each model was measured by the average of the 30 NB models and the values are recorded below. Note the 10% increase in accuracy measure for the second highest performing model. The highest performing model recorded the best accuracy, but the addition of the fourth max-pooling layer proved computationally expensive, proportional to the small size of the max-pooling. Although this would be an ideal system having many, small pooling layers, it was decided that a faster classifier would be more convenient.

<table>
<tr><th>Cross-Validation</th><th>First Filter</th><th>Second Filter</th><th colspan="2">Third Filter</th><th>Accuracy</th></tr>
<tr><td>30 bins</td><td>5x5 Gauss</td><td>6x6 max-pool</td><td colspan="2">2x2 max-pool</td><td>70.76%</td></tr>
<tr><td>30 bins</td><td>7x7 max-pool</td><td>5x5 Gauss</td><td colspan="2">2x2 max-pool</td><td>53.12%</td></tr>
<tr><td>30 bins</td><td>5x5 Gauss</td><td>3x3 max-pool</td><td>2x2 m-p</td><td>2x2 m-p</td><td>71.73%</td></tr>
</table>

Following this, additional features were experimented with that could capture more unique aspects of the objects. Density was found to be a very distinguishing feature, having the added ability of removing edge-case outliers that were badly segmented. Density was defined as the total number of white pixels (mass) divided by the size of image (volume). The battery and angle bracket had very sparse images and generally scored low on this value. Compactness helped to balance density by weighing images that were segmented well. The classes with good segmentation corresponded to classes with high average compactness (less noise in image), encouraging the classifier to be attentive to well classified images. With these features, our final classification **accuracy is measured at 75%** with cross validation on a dataset of 100 objects.

Analysis: Baseline Metric

It is important to explore what the minimum-level classification model would be for this dataset and training problem. Without any features, the baseline to measure a model on is the *a priori* probability distribution of the classes. Because we have 10 classes, assuming uniform distribution, the best model would be to guess one in ten of the classes when classifying; this results in a 10% accuracy or 1/10. The fact that all our model accuracies are higher than this baseline is a good representation that the model has been *improved* upon.

Code Outline:

The *main_loop.m* file is run to start user interaction for loading/building new models. The set of images for building the model is pre-loaded, but the user has the option of loading new images when classifying with a loaded model.

From the *main_loop.m* the user can run *train_new_model.m* to build a new model. Building a new model starts with getting the images and for each doing the following:

1. segmenting the image with *image_segmentation.m*, which will return a list of segmented image objects and the accompanying training object

2. getting the class of the object with *get_class_from_training_image.m* which takes a segmented training image and returns a class

3. getting the feature vector of the object with *get_object_feature_vector.m* which returns a vector of that specific image's feature values.

Data for analysing the model accuracy with cross validation is obtained from *create_cv_train_test_data.m* which takes as input the feature vectors. The model is then trained with cross-validation and analysed via *train_model_nb_cv_accuracy.m*. Data for training and testing the model is obtained from *create_train_test_data.m* which takes as input the feature vectors. A final model is then trained from this data using Matlab's built-in NaiveBayes classifier. All model training systems use *confusion_matrix.m* and *calc_accuracy.m* to analyse the results of the model via a confusion matrix and an accuracy percentage respectively.

During the running of *image_segmentation.m*, the file *edge_detection.m* is used to find the basic outline of objects via thresholding. After this, *apply_conv_pool_sequence.m* performs a sequence of convolution/max-pooling layers via the *max_pooling.m* function. This output is handed to *linkclustering.m* function to cluster the reduced image. Finally the image is scaled to its original size via *scale_image.m* and the clusters are removed as blobs with a reduced rectangle boundary generated by *minimum_bounding_box.m* and returned by *image_segmentation.m*.

During the running of *get_class_from_training_image.m*, the function *hash_rgb.m* is used to create unique hash values for (r, g, b) colour values.

The running of *get_object_feature_vector.m* uses the same functions as *image_segmentation.m* for the same reasons and in a similar flow pattern. The features that are then calculated are created by the following functions: *circle_comparison.m*, *density_of_image.m*, *empty_pixels_in_circle.m* and the compactness algorithm introduced in the lectures.

APPENDIX (CODE):

apply_conv_pool_sequence.m

```
function [image_processed, total_image_reduction] = apply_conv_pool_sequence(image_in,
sequence)
% this function is a quick setup for easy experimenting with multiple
% layers of max_pooling and Gaussian blurring. For now the Gaussian is a
% set filter; this has been fine during testing.

    image_processed = image_in;

    % record the amount the image has been reduced by (via max_pooling)
    total_image_reduction = 1;

    % for now the filter is pre-set as a gaussian blur
    filter = [1 4 7 4 1;
            4 16 36 16 4;
            7 26 41 26 7;
            4 16 26 16 4;
            1 4 7 4 1] ./ 273;

    % itterate through the layer filters
    for i=1:size(sequence, 1)

        % apply convolution
        if(sequence(i, 1) == 0)
            image_processed = conv2(image_processed, filter);

        % apply max pooling
        elseif(sequence(i, 1) == 1)
            pool_v = sequence(i, 2);
            image_processed = max_pooling(image_processed, pool_v);
            % update the total image reduction from max-pooling
            total_image_reduction = total_image_reduction * pool_v;
        end
    end

end
```

calc_accuracy.m

```
function [error_v] = calc_accuracy(model_output, expected_output)
% given the target output of the model and the expected output of the
% model, calculate the accuracy of the model

    num_correct = sum((model_output == expected_output), 1);
    error_v = num_correct / size(model_output, 1);

end
```

circle_comparison.m

```
function [circ_compare] = circle_comparison(binary_image)
% given a binary image, this function calculates the ratio of white pixels
% that are within the maximum-bounding circle of the binary image

    % circle defined by the smallest dimension
    circ_radius = min(size(binary_image, 1), size(binary_image, 2)) / 2;

    image_centre = size(binary_image) ./ 2;

    % create a grid of differences to the centre of the image
    x_grid = repmat([1:size(binary_image, 2)], size(binary_image, 1), 1);
    y_grid = repmat(transpose([1:size(binary_image, 1)]), 1, size(binary_image, 2));
    x_diff = x_grid - repmat(image_centre(1), size(binary_image, 1), size(binary_image, 2));
    y_diff = y_grid - repmat(image_centre(2), size(binary_image, 1), size(binary_image, 2));
    xy_diff = sqrt(x_diff .^ 2 + y_diff .^ 2);

    % find all differences that are less than the radius of the circle
    % (inside the circle)
    diff_binary = xy_diff < circ_radius;

    % calculate the ratio of white pixels in the circle to total white
    % pixels
    num_white_pixels_in_circle = sum(sum(diff_binary .* binary_image, 1));
    num_total_white_pixels = sum(sum(binary_image, 1));

    circ_compare = num_white_pixels_in_circle ./ num_total_white_pixels;

end
```

confusion_matrix.m

```
function [con_m] = confusion_matrix(model_value, expected_value, num_classes)
% given the output of the model and the expected output of the model (and
% number of epected classes), create a confusion matrix for analysing the
% correctness of the model

    con_m = zeros(num_classes, num_classes);

    for i=1:num_classes
        for j=1:num_classes

            con_m(i, j) = sum((model_value == i) .* (expected_value == j));

        end
    end
end
```

create_cv_train_test_data.m

```
function [data_bins] = create_cv_train_test_data(data_m, num_classes, num_bins)
% shuffles the data and sorts it into even sized bins for cross validation;
% each bin must also contain the same class distribution within it.

    data_bins = {};

    for c=1:num_bins
        data_bins{c} = [];
    end

    for i=1:num_classes

        class_data = data_m(find(data_m(:, size(data_m, 2)) == i), :);

        % shuffle the matrix of a single class data
        rand_v = rand(size(class_data, 1), 1);
        [r_sort, r_idx] = sort(rand_v);
        class_data_shuffled = class_data(r_idx, :);

        indx_rate = size(class_data_shuffled, 1) ./ num_bins;

        % dispense the class data evenly through all the bins
        for c=1:num_bins
            c_min = 1 + floor((c - 1) * indx_rate);
            c_max = ceil(c * indx_rate);
            class_train = class_data_shuffled(c_min:c_max, :);
            s_data_test = size(data_bins{c}, 1);
            s_data_train = size(data_bins{c}, 1);
            data_bins{c}((s_data_train + 1):(s_data_train + size(class_train, 1)), :) = class_train;
        end
    end
end
```

create_train_test_data.m

```
function [data_train, data_test] = create_train_test_data(data_m, num_classes, percent_train)
% shuffles the data and sorts into training and testing data, also making
% sure to maintain the distribution of classes within each training and
% testing (because the classes are not all uniformly distributed)

    data_test = [];
    data_train = [];

    % itterate through each class to split into random train and test
    for i=1:num_classes

        % get data of a single class
        class_data = data_m(find(data_m(:, size(data_m, 2)) == i), :);

        % shuffle the matrix
        rand_v = rand(size(class_data, 1), 1);
        [r_sort, r_idx] = sort(rand_v);
        class_data_shuffled = class_data(r_idx, :);

        % divide data into train/test
        class_data_idx = floor(size(class_data_shuffled, 1) * percent_train);
        class_train = class_data_shuffled(1:class_data_idx, :);
        class_test = class_data_shuffled((class_data_idx + 1):size(class_data_shuffled, 1), :);

        % push the train/test class data into the larger train/test sets
        s_data_test = size(data_test, 1);
        s_data_train = size(data_train, 1);
        data_train((s_data_train + 1):(s_data_train + size(class_train, 1)), :) = class_train;
        data_test((s_data_test + 1):(s_data_test + size(class_test, 1)), :) = class_test;

    end

end
```

density_of_image.m

```
function [density] = density_of_image(image_m)
% calculates the density of a binary image, assuming the image is a minimum-bounding box

    % density = mass / volume

    mass_v = sum(sum(image_m, 1));
    volume_v = size(image_m, 1) * size(image_m, 2);
    density = mass_v ./ volume_v;

end
```

edge_detection.m

```
function [image_edge] = edge_detection(image_m, gauss_window)
% uses thresholds and convolution matrices to segment an image into the
% most significant buckets. This function does not make the image binary,
% but is a pre-processing step for doing so; instead it reduces the
% complexity of the image and helps for background removal

    % create a bin of rgb byte values
    edges = transpose(1:255);

    [rows, columns] = size(image_m);
    Ig_vec = reshape(image_m, 1, rows * columns);

    I_hist = histcounts(Ig_vec, edges);

    % smooth the histogram with a gaussian
    filter = gauss_window;
    filter = filter/sum(filter);
    smooth_hist = conv(filter, I_hist);

    % find minimums between thresholds
    offset_error = 0;
    a = smooth_hist < ([10, smooth_hist(:, 1:(size(smooth_hist, 2) - 1))] - repmat(offset_error, 1, size(smooth_hist, 2)));
    b = smooth_hist < ([smooth_hist(:, 2:size(smooth_hist, 2)), 10] - repmat(offset_error, 1, size(smooth_hist, 2)));
    hist_maximums = a .* b .* smooth_hist;

    % create a set of thresholds
    I_thresholds = find(hist_maximums > 0);
    I_num_thresholds = size(I_thresholds, 2);
    threshold_rate = 255 ./ I_num_thresholds;
    Ig_cluster = zeros(size(image_m));

    % for each threshold, cluster the colours of the image into distinct
    % bins
    for i=1:I_num_thresholds

        Ig_cluster = Ig_cluster + (image_m < I_thresholds(:, i)) .* threshold_rate;
    end

    image_edge = Ig_cluster;
end
```

empty_pixels_in_circle.m

```
function [circ_compare] = empty_pixels_in_circle(binary_image)
% given a binary image, this function will calculate the percentage of
% black pixels in a small sub-circle centred in the image. It is assumed
% the image is a minimum-bounding box

    % create a circle half the size of the smallest dimension
    circ_radius = min(size(binary_image, 1), size(binary_image, 2)) / 4;

    image_centre = centre_of_mass(binary_image); %size(binary_image) ./ 2;

    % create a grid of differences to the centre on the image
    x_grid = repmat([1:size(binary_image, 2)], size(binary_image, 1), 1);
    y_grid = repmat(transpose([1:size(binary_image, 1)]), 1, size(binary_image, 2));
    x_diff = x_grid - repmat(image_centre(1), size(binary_image, 1), size(binary_image, 2));
    y_diff = y_grid - repmat(image_centre(2), size(binary_image, 1), size(binary_image, 2));
    xy_diff = sqrt(x_diff .^ 2 + y_diff .^ 2);

    % find all differences less than the radius (in the circle)
    diff_binary = xy_diff < circ_radius;

    % calculate percentage of black pixels in the sub-circle
    black_pixels_in_circle = sum(sum(diff_binary .* (binary_image == 0), 1));
    total_pixels_in_circle = sum(sum(diff_binary, 1));

    circ_compare = (black_pixels_in_circle + 1) ./ (total_pixels_in_circle + 1);
end
```

get_class_from_training_image.m

```
function [class_num] = get_class_from_training_image(m_image)
% given a segmented object from a training image, identify the colour that
% corresponds to the classification of that image

    % colours to look for for each class type
    class_rgb = zeros(10, 1);
    class_rgb(1, 1) = hash_rgb([0, 222, 255]); %one pound
    class_rgb(2, 1) = hash_rgb([218, 2, 251]); %two pound
    class_rgb(3, 1) = hash_rgb([0, 255, 151]); %50p
    class_rgb(4, 1) = hash_rgb([255, 246, 0]); %20p
    class_rgb(5, 1) = hash_rgb([255, 138, 0]); %5p
    class_rgb(6, 1) = hash_rgb([30, 0, 254]); %washer small hole
    class_rgb(7, 1) = hash_rgb([254, 0, 0]); %washer large hole
    class_rgb(8, 1) = hash_rgb([0, 255, 49]); %angle bracket
    class_rgb(9, 1) = hash_rgb([132, 1, 255]); %AAA battery
    class_rgb(10, 1) = hash_rgb([180, 255, 0]); %nut

    %hash the colour values (to make it single-value comaprison
    hashed_im = hash_rgb(double(m_image));

    class_amounts = zeros(10, 1);
    % itterate through each colour and count the number of class colours in
    % the training image (the reason we do this is so that if an image is
    % badly segmented and contains extra class colours, we can pick the
    % colour that appears the most and still classify correctly)
    for i=1:10
        class_amounts(i, 1) = size(find(hashed_im == class_rgb(i, 1)), 1);
    end

    % if no class was found, the image was not segmented properly; set to
    % class -1 so we can ignore this case
    if sum(class_amounts, 1) == 0
        class_num = -1;
    else
        % otherwise return the class colour that appears most in the image
        [max_v, class_num] = max(class_amounts);
    end
end
```

get_object_feature_vector.m

```
function [feature_vec, feature_image, full_image_chunk] = get_object_feature_vector(m_image,
class_image, i)
% given an image and a class image, this function will return a feature
% vector that represents that object based on predefined built-in features


    % normalise the image
    norm_im = m_image ./ repmat(sum(m_image, 3), [1, 1, 3]);

    % seperate the R, G, B components and convert to bytes
    r_im = int64(norm_im(:, :, 1) .* 255);
    g_im = int64(norm_im(:, :, 2) .* 255);
    b_im = int64(norm_im(:, :, 3) .* 255);

    % for each R, G, and B channel, perform max pooling to remove image
    % noise
    r_im = max_pooling(r_im, 2);
    % use edge detection and thresholds to simplify the image colours
    edge_r = edge_detection(r_im, gausswin(2, 1));
    % remove the mode colour (most common colour) for background removal
    edge_r_back = mode(reshape(edge_r, 1, size(edge_r, 1) * size(edge_r, 2)));
    edge_r = edge_r ~= edge_r_back;

    %---
    g_im = max_pooling(g_im, 2);
    edge_g = edge_detection(g_im, gausswin(2, 1));
    edge_g_back = mode(reshape(edge_g, 1, size(edge_g, 1) * size(edge_g, 2)));
    edge_g = edge_g ~= edge_g_back;

    %---
    b_im = max_pooling(b_im, 2);
    edge_b = edge_detection(b_im, gausswin(2, 1));
    edge_b_back = mode(reshape(edge_b, 1, size(edge_b, 1) * size(edge_b, 2)));
    edge_b = edge_b ~= edge_b_back;

    % create a shared binary image and remove any more noise via mode
    % colour
    edge_binary = edge_r + edge_g + edge_b;
    edge_binary = edge_binary - mode(reshape(edge_binary, 1, size(edge_binary, 1) *
size(edge_binary, 2)));
    % find the minimum-bounding box of the binary image
    [box_f, l, r, t, b] = minimum_bounding_box(edge_binary > 0);

    % scale the binary image to the original ratio and convert the current
    % image to the new shared ratio
    [scl_binary, m_image_new] = scale_image(edge_binary, m_image(:, :, 1));
    m_image = m_image(1:size(m_image_new, 1), 1:size(m_image_new, 2), :);
    %extract the object from the rest of the image by overlaying the binary
    %image onto the original object image
    im_subtr = m_image(1:size(scl_binary, 1), 1:size(scl_binary, 2), :);

    % save the image bounding box with full detail
    full_image_chunk = im_subtr;

    %subtract image
    im_subtr = im_subtr .* repmat(scl_binary, [1, 1, 3]);

    % get the average colour of the subtracted object for a classification
    % feature
    color_avg = reshape(sum(sum(im_subtr, 1)) ./ sum(sum(scl_binary, 1)), 1, 3);

    % define a feature vector for the object
    feature_vec = zeros(1, 5);

    % feat 1: how similar the object is to a circle
    feature_vec(1, 1) = circle_comparison(box_f);
    % feat 2: how much of a hole the object has
    feature_vec(1, 2) = empty_pixels_in_circle(box_f);
    % feat 3: the hashed value of the objects average colour
    feature_vec(1, 3) = color_avg(1) + (color_avg(2) + 1) .^ 2 + (color_avg(3) + 2) .^ 3;
    % feat 4: the density of the white pixels in the object image
    feature_vec(1, 4) = density_of_image(box_f);
    % feat 5: the compression of the object
    area = bwarea(box_f);
    perim = bwarea(bwperim(box_f, 4));
    feature_vec(1, 5) = perim*perim/(4*pi*area);

    feature_image = box_f;

end
```

hash_rgb.m

```
function [hash_val] = hash_rgb(rgb_m)
% this function will hash an rgb byte vector

   % check that the vector is not a 3D matrix as a vector
   if size(rgb_m, 3) ~= 1
      hash_val = rgb_m(:, :, 1) + (rgb_m(:, :, 2) + 255) .^ 2 + (rgb_m(:, :, 3) + 510) .^ 3;
   else
      hash_val = rgb_m(1) + (rgb_m(2) + 255) .^ 2 + (rgb_m(3) + 510) .^ 3;
   end
end
```

image_segmentation.m

```
function [final_images, final_class_images] = image_segmentation(image_m, class_image, pool1, pool2)
% given an image, a training class image and 2 max-pooling values, perform
% image segmentation via blob-analysis and attempt to remove blobs as
% individual minimum-bounding boxes

   I = image_m;

   Id = double(I) / 255;
   % greyscale the image
   Id_norm = Id ./ repmat(sum(Id, 3), [1, 1, 3]);

   Ig = sum(image_m, 3) / 3;
   % normalise the image
   norm_im = Id ./ repmat(sum(Id, 3), [1, 1, 3]);

   % take the seperate R, G, and B matrices of the image
   r_im = int64(norm_im(:, :, 1) .* 255);
   g_im = int64(norm_im(:, :, 2) .* 255);
   b_im = int64(norm_im(:, :, 3) .* 255);

   % perform edge detection on each R, G, and B matrix
   edge_r = edge_detection(r_im, gausswin(5, 2));
   % remove the background by finding the most common colour
   edge_r_back = mode(reshape(edge_r, 1, size(edge_r, 1) * size(edge_r, 2)));
   edge_r_back_remove = edge_r ~= edge_r_back;

   edge_g = edge_detection(g_im, gausswin(5, 2));
   edge_g_back = mode(reshape(edge_g, 1, size(edge_g, 1) * size(edge_g, 2)));
   edge_g_back_remove = edge_g ~= edge_g_back;

   edge_b = edge_detection(b_im, gausswin(5, 2));
   edge_b_back = mode(reshape(edge_b, 1, size(edge_b, 1) * size(edge_b, 2)));
   edge_b_back_remove = edge_b ~= edge_b_back;

   % perform binary OR on R, G, and B to get a full edge image
   Ig_edged = ceil((edge_r_back_remove + edge_g_back_remove + edge_b_back_remove) ./ 3);
```

```
    % perform a sequence of max-pooling and guassian convolution filters
    conv_pool_filter= [0, 0;
                        1, pool1;
                       1, pool2];

    [filtered_I, total_image_reduction] = apply_conv_pool_sequence(Ig_edged, conv_pool_filter);

    % cluster the blobs using linkclustering
    [num_clusters, clustered] = linkclustering(filtered_I);

    % scale the image back to the original size (scaled due to convolutions
    % and max-pooling)
    [scaled_filtered_I, Ig] = scale_image(clustered, Ig);

    final_images = {};
    final_class_images = {};

    % itterate through each cluster
    for i=1:(num_clusters - 1)
        % figure(13 + i)
        % get the cluster
        im_cluster = Ig .* (scaled_filtered_I == i);
        % get the minimum-bounding box of the cluster
        [im_bounded, b_left, b_right, b_top, b_bottom] = minimum_bounding_box(im_cluster);
        % im_bounded = minimum_bounding_box(im_cluster);

        % take the bounding box from the original image
        final_images{i} = Id(b_left:b_right, b_top:b_bottom, :);

        % take the bounding box from the training image
        final_class_images{i} = class_image(b_left:b_right, b_top:b_bottom, :);
    end

end
```

linkclustering.m

```
function [num_clusters, assigned_cl] = linkclustering(m_data)
% a simple form of clustering that involves clustering pixels in an image
% only if they are direct neighbours (touching, not corners). This
% clustering algorithm is quick because it only requires each pixel be
% processed once. Downsides are that it relies on effective preprocessing of
% the image

    testData = m_data;

    % create a new Java Queue object
    q = javaObject('java.util.LinkedList');

    % array for storing classification of pixels
    assigned_cl = zeros(size(testData));
    num_clusters = 1;
    % maximum allowed number of itterations before clustering fails
    itterations = 100000;

    while itterations > 0
        % check if queue is empty
        if q.size() == 0
            % find a non-zero pixel
            [v_row, v_col] = find(testData > 0);
            v = [v_row, v_col];

            % if no non-zero pixels exist, the algorithm is done
            if size(v, 1) == 0
                return
            else
                % otherwise add the first non-zero pixel to the queue and
                % assign it a class
                a = v(1, :);
                assigned_cl(a(1), a(2)) = num_clusters;
                num_clusters = num_clusters + 1;
                q.add(a);
            end
        end

        % pop the top value from the queue
        n = q.remove();
        testData(n(1), n(2)) = 0;
        n_edge = zeros(1, 2, 4);
        % get the direct neighbours of the popped pixel (space complexity)
        n_edge(:, :, 1) = [n(1), n(2) - 1];
        n_edge(:, :, 2) = [n(1) - 1, n(2)];
        n_edge(:, :, 3) = [n(1), n(2) + 1];
        n_edge(:, :, 4) = [n(1) + 1, n(2)];

        % itterate through neighbours, do edge-case checks and add all
        % neighbours to the queue after assigning them the class of the
        % popped pixel
        for i=1:4

            n_01 = n_edge(:, :, i);
            if sum((n_01(1) <= 0) + (n_01(2) <= 0) + (n_01(1) > size(testData, 1)) + (n_01(2) >
size(testData, 2))) == 0
                if(testData(n_01(1), n_01(2)) > 0)
                    assigned_cl(n_01(1), n_01(2)) = assigned_cl(n(1), n(2));
                    q.add(n_01);
                end
            end
        end
        % count down the max itterations
        itterations = itterations - 1;
    end
end
```

main_loop.m

```
% the file to run to begin interactive with the interface and train/load models

% get training data from file
I_collection = {};
I_collection{1} = imread('simpler/02.jpg');
I_collection{2} = imread('simpler/03.jpg');
I_collection{3} = imread('simpler/04.jpg');
I_collection{4} = imread('simpler/05.jpg');
I_collection{5} = imread('simpler/06.jpg');
I_collection{6} = imread('simpler/07.jpg');
I_collection{7} = imread('simpler/08.jpg');
I_collection{8} = imread('simpler/09.jpg');
% I_collection{9} = imread('simpler/10.jpg');
I_training = {};
I_training{1} = imread('simpler/training/02.jpg');
I_training{2} = imread('simpler/training/03.jpg');
I_training{3} = imread('simpler/training/04.jpg');
I_training{4} = imread('simpler/training/05.jpg');
I_training{5} = imread('simpler/training/06.jpg');
I_training{6} = imread('simpler/training/07.jpg');
I_training{7} = imread('simpler/training/08.jpg');
I_training{8} = imread('simpler/training/09.jpg');
% I_training{9} = imread('simpler/training/10.jpg');

% interface begins here
lacm = input('load a current model? Y/N\n\n   ', 's');
if lacm == 'y' || lacm == 'Y'
    new_model = false;
else
    new_model = true;
end

% build a new model
if new_model

    % series of inputs from user
    ss = input('show image segments? Y/N\n\n   ', 's');
    if ss == 'y' || ss == 'Y'
        show_segments = true;
    else
        show_segments = false;
    end

    sfvs = input('show feature vector space? Y/N\n\n   ', 's');
    if sfvs == 'y' || sfvs == 'Y'
        show_vec = true;
    else
        show_vec = false;
    end

    model_name = input('type name of model: \n\n   ', 's');
    input('...click enter to begin building the model...');

    % train the model
    [nb_model, model_train_vecs, confusion_m, m_accuracy] = train_new_model(I_collection,
I_training, show_segments, show_vec);

    % save the model file locally
    file_name = strcat([model_name, '.csv']);
    file_data = model_train_vecs;
    delim = ',';

    dlmwrite(file_name, file_data, delim);
else
    % load a current model
    old_model_file = input('type the filename of your model: \n\n   ', 's');
    old_model_file = strcat([old_model_file, '.csv']);

    delim = ',';
    file_data = dlmread(old_model_file, delim);

    input('loaded successfully. Hit enter to format the model...\n\n   ');

    % build the model from the training data (we can do this because models
    % are so small the user won't notice it being rebuilt)
    nb_model = fitNaiveBayes(file_data(:, 1:(size(file_data, 2) - 1)), file_data(:, size(file_data, 2)));

    feed = 'c';

    while feed == 'C' || feed == 'c'

        im_file = input('type image file to classify:\n\n   ', 's');
        pred_im = imread(im_file);

        % segment the objects from the image
        [final_images, class_images] = image_segmentation(pred_im, pred_im, 6, 2);

        obj_vectors = [];
        obj_images = {};

        % itterate over each segmented object
        for i=1:size(final_images, 2)
            % get the feature vector for this object
            [obj_vectors(size(obj_vectors, 1) + 1, :), feature_image, obj_images{size(obj_images, 2) +
1}] = get_object_feature_vector(final_images{i}, i * 101);
            figure()
            imagesc(feature_image)
        end

        pred_class = nb_model.predict(obj_vectors)
    end
end
```

max_pooling.m

```
function [M] = max_pooling(m, m_size)
% given an image m and a size of the pool, perform max pooling wit step
% size equal to the size of the pool over the image. This will reduce the
% images size by the size of the pool (note: if image dimensions and pool
% size are not divisible, some image data will be lost)

   % set the step rate for rows and columns
   pool_r = floor(size(m, 1) / m_size);
   pool_c = floor(size(m, 2) / m_size);

   % create the new image matrix at the expected size
   M = double(zeros(pool_r, pool_c));

   for r=1:pool_r
      for c=1:pool_c
         % get the starting index of the pool at (r, c)
         r_ind = (r - 1) .* m_size + 1;
         c_ind = (c - 1) .* m_size + 1;
         % extract the m_size x m_size matrix from the image
         overM = zeros(size(m));
         overM(r_ind:(r_ind+m_size - 1), c_ind:(c_ind+m_size - 1)) = double(ones(m_size,
m_size));
         pooled_out_m = overM .* double(m);
         % find the extracted matrix's max value and assign it to the
         % new image
         M(r, c) = max(max(pooled_out_m));
      end
   end
end
```

minimum_bounding_box.m

```
function [box_final, box_left, box_right, box_top, box_bottom] =
minimum_bounding_box(m_data)
% this function will attempt to find a minimum-bounding box of a binary
% image (or assuming that any pixel with 0 value is not the object to be
% bounded). This is not the most state-of-the-art method, but it does the
% trick effectively.

   testData = m_data;

   % find the centre of the non-empty area of pixels
   [v_y, v_x] = find(testData > 0);
   v_centre = [(sum(v_x, 1) ./ size(v_x, 1)), (sum(v_y, 1) ./ size(v_y, 1))];

   % set the data gained values to big numbers
   box_data = -100;
   data_gain = 100;
   box_offset = 5;

   % loop and expand the box until no data is gained
```

```
    while(data_gain > 0)

        % expand the box starting with a left and right expansion
        box_left = round(max(1, (v_centre(2) - box_offset)));
        box_right = round(min(size(testData, 1), (v_centre(2) + box_offset)));
        box_1 = testData(box_left:box_right, :);

        % calculate the amount of new data gained by expanding the box (num
        % new non-empty pixels)
        data_box_1 = sum(sum(box_1, 1));
        data_gain = data_box_1 - box_data;

        % if no data gain, save box and end loop
        if(data_gain > 0)
            box_final = box_1;
        end

        % otherwise expand the box
        box_data = data_box_1;
        box_offset = box_offset + 1;

    end

    %---- REPEAT the same process as above, but now expand the box
    %vertically, keeping the same left and right positions of the box
    %----

    box_data = -100;
    data_gain = 100;
    box_offset = 5;

    while(data_gain > 0)

        box_top = round(max(1, (v_centre(1) - box_offset)));
        box_bottom = round(min(size(testData, 2), (v_centre(1) + box_offset)));
        box_1 = testData(box_left:box_right, box_top:box_bottom);

        data_box_1 = sum(sum(box_1, 1));
        data_gain = data_box_1 - box_data;

        if(data_gain > 0)
            box_final = box_1;
        end

        box_data = data_box_1;
        box_offset = box_offset + 1;

    end
end
```

scale_image.m

```
function [image_scaled, target_scaled] = scale_image(m_image, m_image_target)

    edge_loss = mod(size(m_image_target), size(m_image));
    scale_v = floor(size(m_image_target) ./ size(m_image));

    image_scaled = kron(m_image, ones(scale_v));

    target_scaled = m_image_target(1:(size(m_image_target, 1) - edge_loss(1)), 1:
(size(m_image_target, 2) - edge_loss(2)));

    % loss_of_pixels = size(image_expand_tiled) - size(m_image_target);
    % border_loss_l = ceil(abs(loss_of_pixels / 2));
    % border_loss_r = floor(abs(loss_of_pixels / 2));
    % image_scaled = image_expand_tiled(border_loss_l(1):(size(image_expand_tiled, 1) -
border_loss_r(1) - 1),
    %                              border_loss_l(2):(size(image_expand_tiled, 2) - border_loss_r(2) - 1));

end
```

train_model_nb_cv_accuracy.m

```
function [ accuracy, accuracies ] = train_model_nb_cv_accuracy( cv_data )
% using a set of cross-validation bins, his function will perform x (num
% bins) model trainings using each bin as a test set. At the end it will
% report the average of the accuracies of the classifiers.

    accuracies = zeros(size(cv_data, 2), 1);

    % itterate through each bin
    for i=1:size(cv_data, 2)

        % set the current bin as the test set
        test_data = cv_data{i};

        % compile a training set from the rest of the bins
        train_data = [];
        for j=1:size(cv_data, 2)

            if i ~= j
                s_data_train = size(train_data, 1);
                train_data((s_data_train + 1):(s_data_train + size(cv_data{j}, 1)), :) = cv_data{j};
            end
        end

        % fit a model to the training and test set
        nb_model = fitNaiveBayes(train_data(:, 1:(size(train_data, 2) - 1)), train_data(:,
size(train_data, 2)));
        pred_classes = nb_model.predict(test_data(:, 1:(size(test_data, 2) - 1)));
        % calculate the accuracy
        accuracies(i, 1) = calc_accuracy(pred_classes, test_data(:, size(test_data, 2)));

    end

    % get the average accuracy of the different models
    accuracy = sum(accuracies, 1) / size(accuracies, 1);

end
```

train_new_model.m

```
function [nb_model, model_train_vecs, con_m, accuracy ] = train_new_model(I_collection,
I_training, show_segmentation_ims, show_vec_space)

    obj_vectors = [];

    obj_classes = [];

    if show_segmentation_ims
        obj_images = {};

        for i=1:11
            obj_images{i} = {};
        end
    end

    % itterate over each image in the collection of images
    for j=1:size(I_collection, 2)

        % segment the objects from the image
        [final_images, class_images] = image_segmentation(I_collection{j}, I_training{j}, 6, 2);

        % itterate over each segmented object
        for i=1:size(final_images, 2)

            % get the class for this object
            class_type = get_class_from_training_image(class_images{i});
            obj_classes(size(obj_vectors, 1) + 1, 1) = class_type;

            % get the feature vector for this object
            [obj_vectors(size(obj_vectors, 1) + 1, :), feature_image] =
get_object_feature_vector(final_images{i}, j * 101 + i);

            if show_segmentation_ims
                if(class_type > 0)
                    obj_images{class_type}{size(obj_images{class_type}, 2) + 1} = feature_image;
                else
                    obj_images{11}{size(obj_images{11}, 2) + 1} = feature_image;
                end
            end

        end
    end
    % find feature vectors that were made incorrectly (error in segmentation) and remove them
    indx_nan = find(sum(isnan(obj_vectors ), 2) > 0);
    obj_vectors(indx_nan,:) = [];
    obj_classes(indx_nan,:) = [];
    indx_neg = find(obj_classes < 0);
    obj_vectors(indx_neg,:) = [];
    obj_classes(indx_neg,:) = [];

    % create a data|class matrix for training
    data_m = [obj_vectors, obj_classes];

    % % divide data into train/test
    [data_train, data_test] = create_train_test_data(data_m, 10, 0.75);

    % train a Naive Bayes classifier on the data
    nb_model = fitNaiveBayes(data_train(:, 1:(size(data_train, 2) - 1)), data_train(:, size(data_train,
2)));

    % apply the NB model to the test set
    pred_classes = nb_model.predict(data_test(:, 1:(size(data_test, 2) - 1)));

    % do cross-validation on the modeling
    num_bins = 30;
    data_cv = create_cv_train_test_data(data_m, 10, num_bins);
    [accuracy, accuracies] = train_model_nb_cv_accuracy(data_cv);

    % create a confusion matrix of the ouput
    con_m = confusion_matrix(pred_classes, data_test(:, size(data_test, 2)), 10);

    model_train_vecs = data_train;

    if show_vec_space
        % display a feature vector space of first 3 features for analysis
        scatter3(obj_vectors(:, 1), obj_vectors(:, 2), obj_vectors(:, 3), 12, obj_classes);
        xlabel('circle comparison');
        ylabel('holes in object');
        set(get(gca, 'ZLabel'), 'String', 'colour hash');
    end

    if show_segmentation_ims
        for c=1:10
            for i=1:size(obj_images{c}, 2)
                figure(c * 20 + i)
                colormap(gray)
                imagesc(obj_images{c}{i})
            end
        end
    end

end
```