Machine Vision Report

Bernardo Branco

Student ID: ucabbb0

All figures/images can be found inside the folder Figures

Part 1:

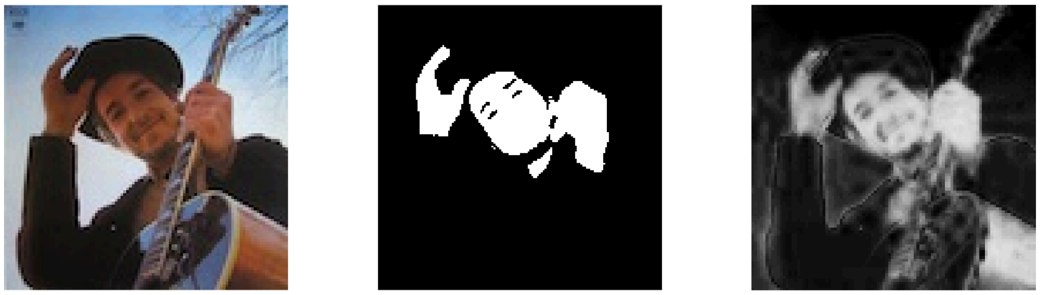
Practical a):

a) In this TODO I am calculating the mean of the data by adding all of the data points, thus obtaining a row vector with the sum of all the data points in the three dimensions. Afterwards, I am dividing each of entry of that row vector by the number of data points. I am calculating the covariance by finding the mean squared difference (for each dimension) and dividing the result by the number of data points minus 1.

b) In this function I am calculating the Gaussian likelihood of an individual pixel (this function is used for both Gaussians, one pertaining to skin pixels and the other to non-skin). The formula used was the multivariate normal distribution presented in the slides since we are dealing with data points with three dimensions.

c) In this TODO I am calculating the posterior probability of a particular pixel being skin. To do this I am using the Bayes rule, where the posterior probability is the product of the likelihood and the prior, divided by the normalizing factor (the likelihood of it being skin multiplied by the prior of a pixel being skin added to the product of the likelihood of the same pixel being non-skin and the prior probability for non-skin).

Result (Image 1):



Observations: As can be observed from the resulting image the current procedure does a relatively good job in identifying the skin pixels. However, for pixels that are not skin but have values more or less similar to those of skin the Gaussian model (for skin) incorrectly identifies them as skin, which is the case of the pixels belonging to the guitar. This suggests that more then one Gaussian should be used to classify to skin and non-skin pixels.

Practical b)

d) In order to randomly generate 1 dimensional data points from a Gaussian, the linear equation formula: $y = mx+b$ was used, where the sigma from the normal distribution is the slope (m), a randomly generated number between 0 and 1 is the 'x', and the bias is the mean. The output (y) are the 1 dimensional data points.

e) In this TODO we are calculating the likelihood for a particular data point (using the calcGaussianProb function) for each Gaussian, and then multiplying the likelihood by the respective weight of the Gaussian.

Results:

Figure 1: Visual representation of the true Gaussians fitting the data points:

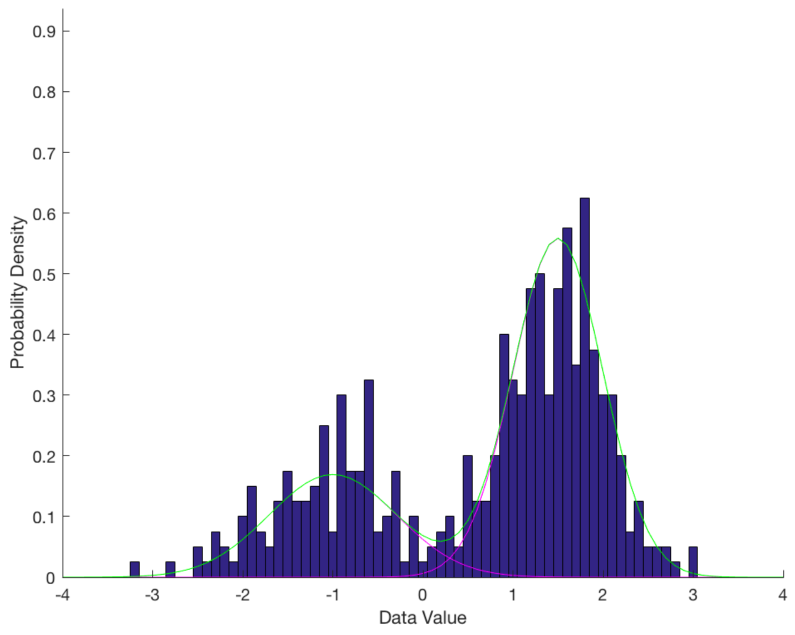
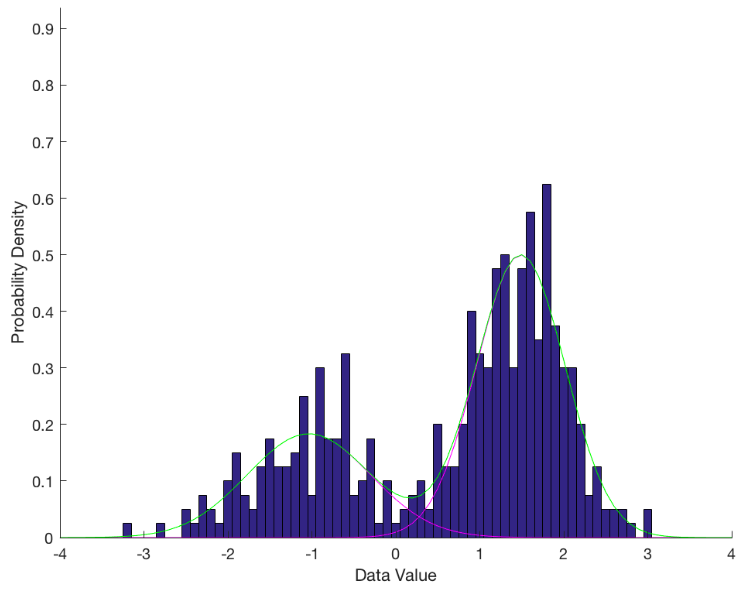


Figure 2: Visual representation of the approximated Gaussians using the EM algorithm fitting the data points.



Observations: In this practical the EM algorithm is pretty successful in generating Gaussians which are very close to the actual true Gaussians. Sometimes it need more then 20 iterations to actually converge (which was the case in the figures presented above since after 20 iterations the algorithm stopped but it had not reached convergence yet). However, it seems to obtain an optimal result, or very close to optimal almost all of the time.

Practical c)

f) Similarly to the d) section here we are generating our data points from the Gaussians that we are later on trying to approximate (through the EM algorithm). Again, the y = mx+b is being used. However, the biggest difference now is that our data points are not one dimensional, but in this particular case have 2 dimensions. Our m will be the result of cholcov (Cholesky-like covariance decomposition) which computes the square, upper triangular (if sigma is positive definite, not necessarily upper triangular otherwise) Cholesky factor (T) where Sigma = T'*T. This Cholesky factor is then multiplied by a randomly generated matrix of size 2*1 whose entries are in between 0 and 1 (in this case the randomly generated x is a vector because the data points have 2 dimensions) and then this is added to the bias (mean) thus generating 3 clusters of data points, each belonging to one of the three "true" Gaussians.

g) In this section we are first calculating the likelihoods of all the data points with respect to all of the Gaussians (the likelihood of each data point given a certain Gaussian is then multiplied by the respective weight of the Gaussian). Then the posterior (responsibilities) are calculated by normalizing (the three likelihoods of each data point are each divided by the sum of the three likelihoods).

h) In this TODO we update the weights of each Gaussian by summing all of the responsibilities of each Gaussian and dividing the result by the sum of all of the responsibilities of all Gaussians. Since for each data point the sum of the responsibilities is equal to 1, this normalizing factor is also equal to the number of data points (which is a less expensive calculation to do in matlab).

i) In this TODO we update the means of each Gaussian by multiplying each data point with its respective responsibility (for each Gaussian) and then normalizing it by dividing the result with the sum of all of the responsibilities for that respective Gaussian. In order to do this, the postHidden was transformed into a matrix with the number of rows equivalent to the number of dimensions, and where each row is a copy of the postHidden array. Then the array multiplication between this new matrix and the data is performed (since both have the same dimensions) and then by summing the transpose of this resulting matrix we obtain the value for the numerator.

j) Finally, in this TODO we update the covariances for each Gaussian. We do this by multiplying each responsibility value by the difference between the data and the mean and the same difference but transposed (resulting in the mean squared error for each data point, for each Gaussian). This is then again normalized by the sum of all of the responsibilities of the respective Gaussian.

Results:

Figure 3: Visual representation of the true Gaussians fitting the data points:

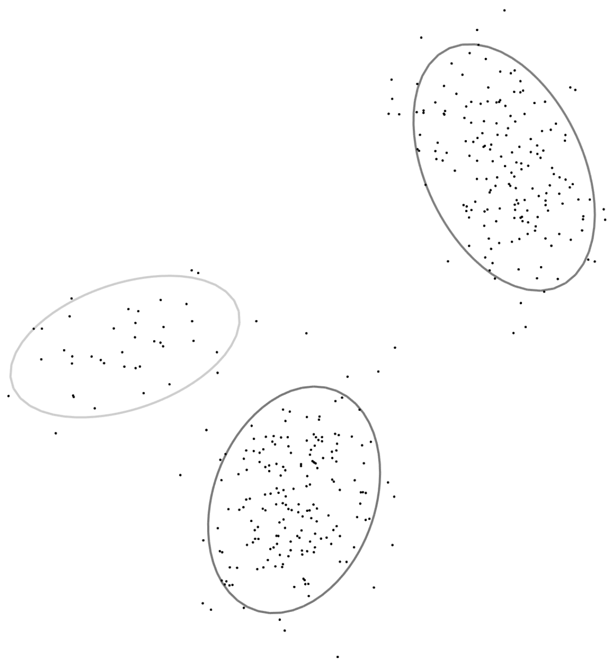
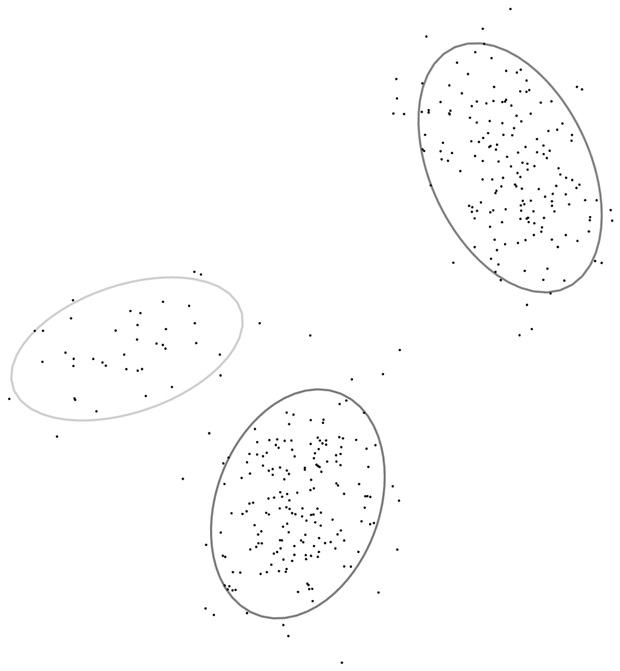


Figure 4: Visual representation of the approximated Gaussians, using the EM algorithm, fitting the data points.



Observations: In pratical C because the EM algorithm began by initializing 3 completely random Gaussians it happened very frequently that a Gaussian would be initialized as having a very small covariance and with a mean very similar to the values of a certain data point. As more iterations of the EM algorithm occurred this Gaussian would begin fitting to that one particular data point and eventually become singular and disappear (leaving two Gaussians to fit the data points generated by the three different true Gaussians). In order to improve upon this, instead of initializing the Gaussians in a completely random manner I built a new function (InitializeGaussians) that would pick K (number of Gaussians) number of random data points. This would be the mean for each of the Gaussians. The covariance I obtain by calculating the mean squared error for each data point (where the means would be the three randomly selected data points) and dividing that by the number of data points. These results would then be inserted in a diagonal matrix of size nDim by nDim, which is the resulting covariance for the Gaussian. This improved the algorithm significantly, since the mistake that was happening before was solved. Finally, since we are talking about approximating more Gaussians (where data points of higher dimension) then in pratical C I decided to increase the number of iterations of the EM algorithm to 40, since with 20 the EM would rarely converge. In addition, for efficiency purposes, I also added a little piece of code that stops the algorithm automatically in case it has already converged (even if it hasn't finished the 40 iterations).

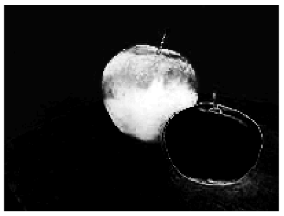
Part II)

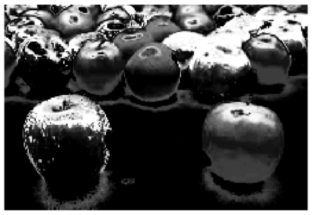
A) The file LoadData.m is what builds the training data. The objective of this function is to build two arrays, one with all of the pixels (and their 3 dimensions, rgb) corresponding to apple pixels and another with all of the non-apple pixels. After reshaping both the image and mask matrices into arrays, in order to obtain all of the apple pixels I simply multiplied (using array multiplication) the image by the mask, therefore all of the apple pixels were kept and all of the non apple pixels became 0. An important step I did before was to add to each pixel a value of 0.1 and then multiply the image by its mask so that the apple pixels which were white were still considered as apple pixels (and hence didn't have a 0 value after being multiplied by the mask). After removing every pixel which had a 0 value in all of the rgb dimensions I then subtracted 0.1 from each pixel, and thus end up with an array of all of the apple pixels in our training data. Similarly, the same procedure was done to obtain all of the pixels that were non-apples, the only difference was that instead of multiplying the image by its mask, it was multiplied by the negation of the mask (where all of the 1s are turned into 0s and all of the 0s into 1s). These arrays of pixels were then saved as appleTrain and nonAppleTrain, so that I didn't have to repeat these steps everytime I wanted to train my models.

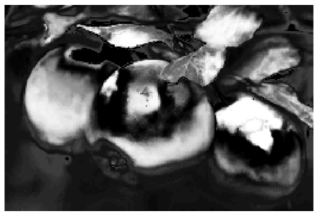
B) I trained my models (fed the training data into the fitMixGauss function in part c) using various numbers of Gaussians for both the apple and non apple pixels (the most Gaussians used were 4 for both the apple and non-apple pixels).

C) With a total of 16 different models (combination of up to 4 Gaussians for both apple and non-apple pixels) I tested each one in the images provided by the teacher. The posteriors for each pixel were represented in the output images, which use the gray colormap (as used in pratical a), where the lighter pixels were those that the model attributed a higher probability of belonging to apples, and the darker are, conversely, those that the model attributed a higher probability of belonging to non-apples. Below I show the results obtained using 3 Gaussians for both apple and non-apple pixels.

Image 2:







Observations: In the resulting images it can be seen that apples whose color (brighter green and darker red) diverges from that which was normally seen in the training data were usually misclassified by the models. The leaves in the third picture were also mistaken by apples because their green tone is very similar to that seen in apples.

D) For this part I tested the 16 different models (combination of up to 4 Gaussians for both apple and non-apple pixels) on the photo which had the mask image provided by the teacher. I did not however, try to find the optimal priors due to the fact that pictures containing apples can vary so much, and ideally, the more training data the less significant these priors will be. In order to be able to quantitatively compare the results of the different models on the images I calculated the ROC curve for each case. In order to produce the ROC curve I developed the function produceRoc1. Essentially, what is being done is separating the apple from non-apple pixels by applying 100 different thresholds (evenly distributed from a range of 0 to 1). Any posterior value above (or equal to) the threshold is considered to be apple (and thus substituted by 1) and any value below is considered to be non-apple (substituted by 0). Afterwards, I determine the number of pixels that are true positives, in other words, the number of all of the pixels that were above the threshold and that actually correspond to apples, as well as the true negatives, all of the pixels that were below the threshold and that actually correspond to non-apple pixels. By dividing the number of true positives by the number of apple pixels (positives calculated by just summing the entries of the mask vector) we obtain the true positive ratio (tpr) for that threshold. In order to obtain the false positive ratio (fpr) I used the formula fpr = 1-SPC, where SPC = TN/N (the number of true negatives divided by the number of non-apple pixels). I then plotted all of the ROC curves in one graph, where fpr is in the x-axis and tpr is in the y-axis. Finally, I used the metric of Area Under Curve (AUC) to compare and identify the best results. The area under the curve was calculated using the trapz function, which essentially finds the area of the rectangle beneath two points, and the remaining area (the triangle that fits on top of the rectangle) is obtained by halving the area of the rectangle above which joins the two points together (and it iteratively adds all of the areas for all of the thresholds, 100). These were the results (all the figures can be found in the project folder):

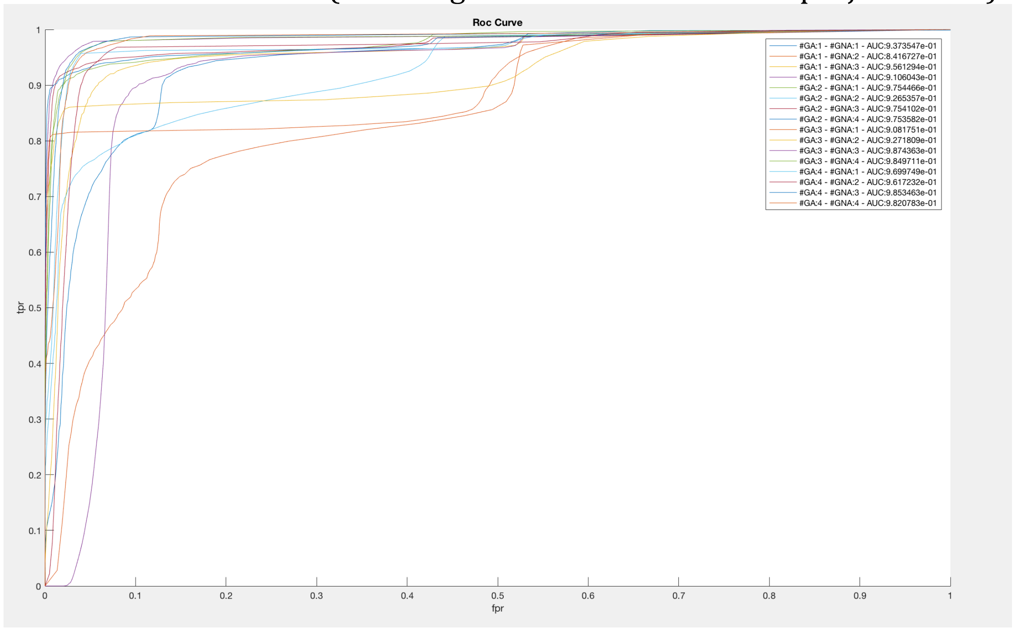


Figure 5

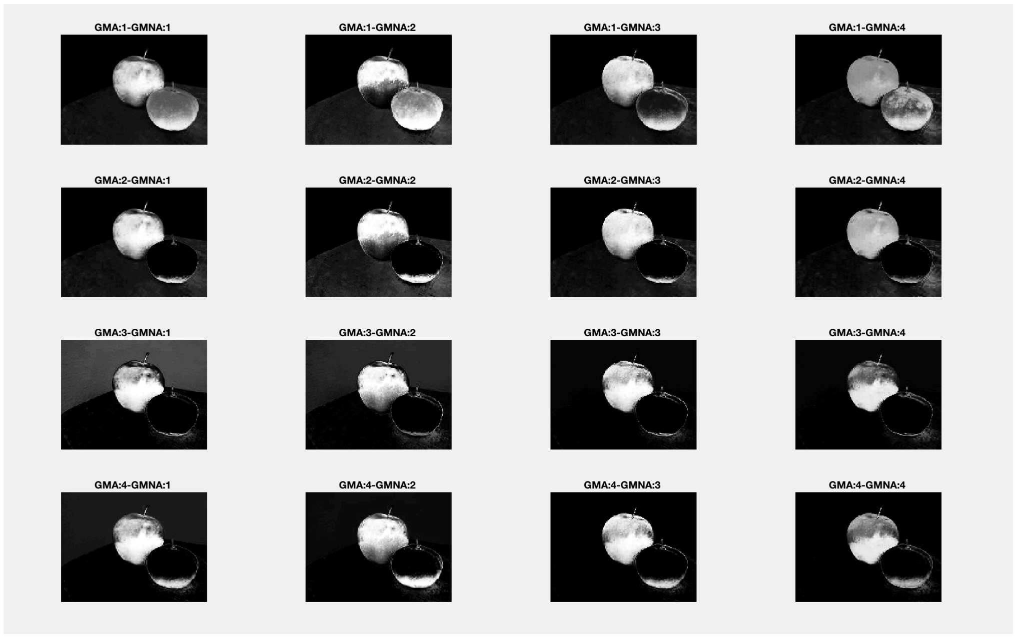


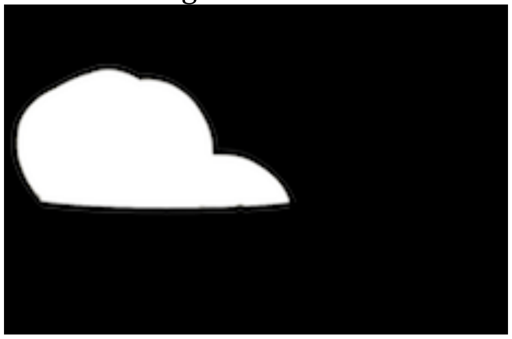
Figure 6

Observations: Generally, the models were pretty successful in correctly identifying the apple and non-apple pixels. According to the ROC curve, the combination of three Gaussians for both apple and non-apple pixels was what most correctly classified the pixels (with an AUC of 0.987). It must be mentioned though that this picture is very simple, since the background is very well contrasted to the apple (and does not have a similar color to those of apples) and the other fruit (the orange) has also a very different color from that of apples. In addition, red apples are best classified by the models, due to both the training data and also the fact that the tone of green in green apples tends to vary more then the red. Given the fact that there is only one picture (and that picture is very simple) this cannot be used as our validation set. This is because even though for this picture better results were seen with 3 Gaussians for apples and non-apple pixels, other, more complicated (with more apples, of different colors, and a background with smaller contrast) might work best with another combination of Gaussians.
Note: Due to efficiency considerations I decided to firstly reduce the size of the test images to 150 rows (the height-width proportion was kept), these images/masks have the number 1 at the end of their name, and I fitted the Gaussians to the training data using matlab's built-in function: fitgmdist. Even though my fitMixGauss function also works (and it was used to produce the image seen in part C) it takes a lot longer then the built-in function and would not allow me to properly test all of the different models. My function trainModels (in the file models.m) receives a flag as the first argument. If set to 1 then it will fit the Gaussians using fitgmdist, else it fits them using the fitMixGauss which I developed.

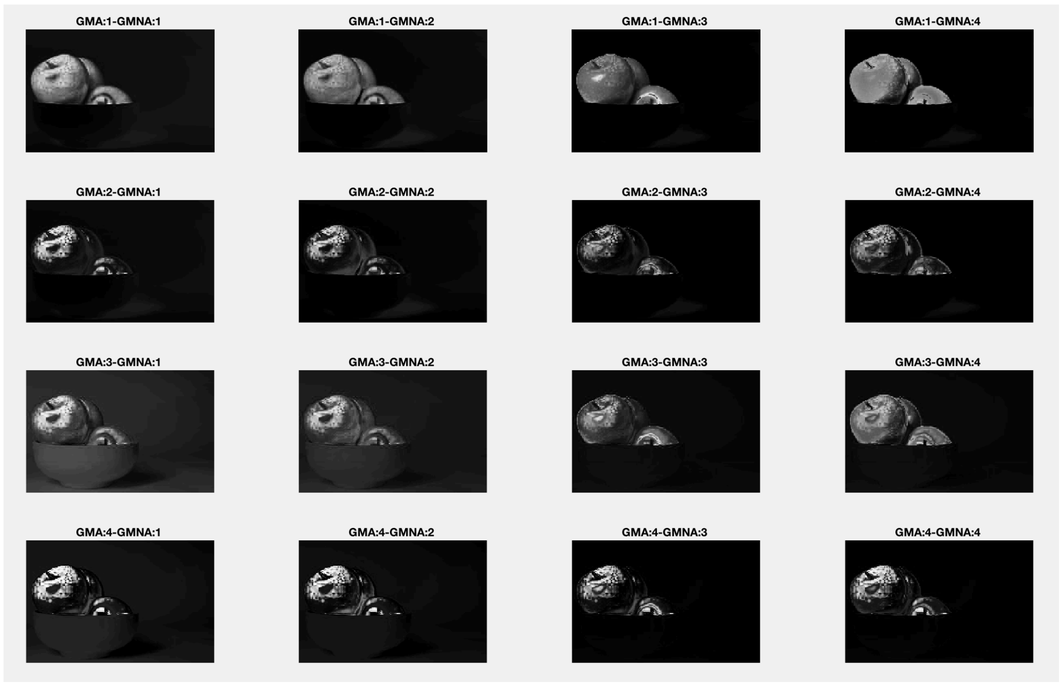
E) After downloading two non-copyrighted photos and producing their respective masks, both images were tested on the same models as above, these are the results:

Image 1:

Mask for image 1:



Posteriors for Image 1 (figure 7):



ROC curves (figure 8):

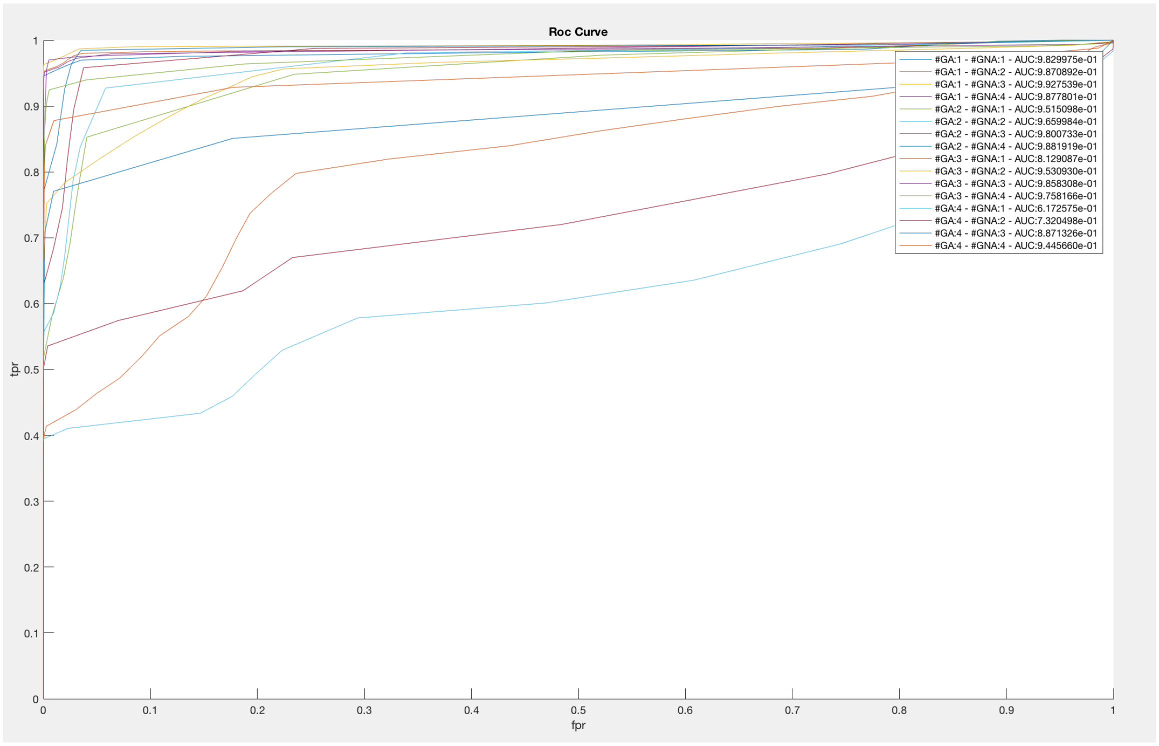
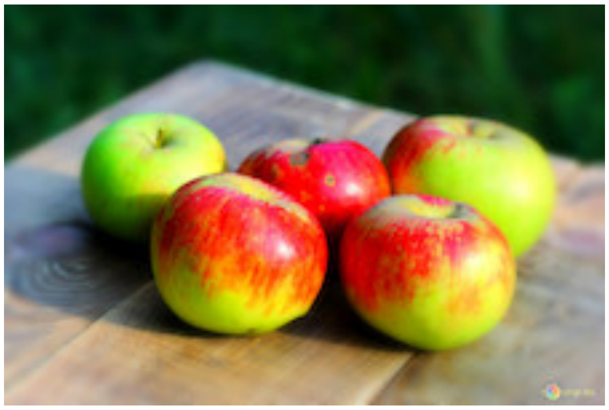
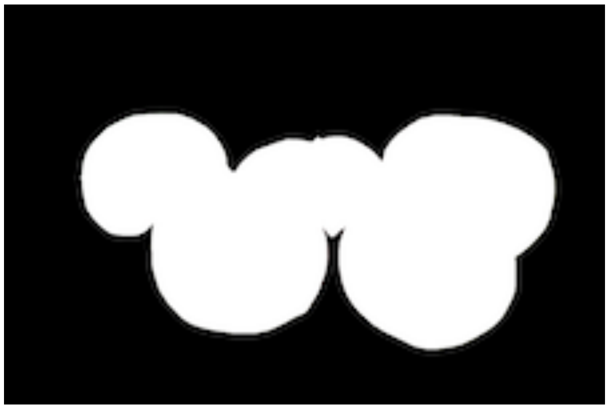


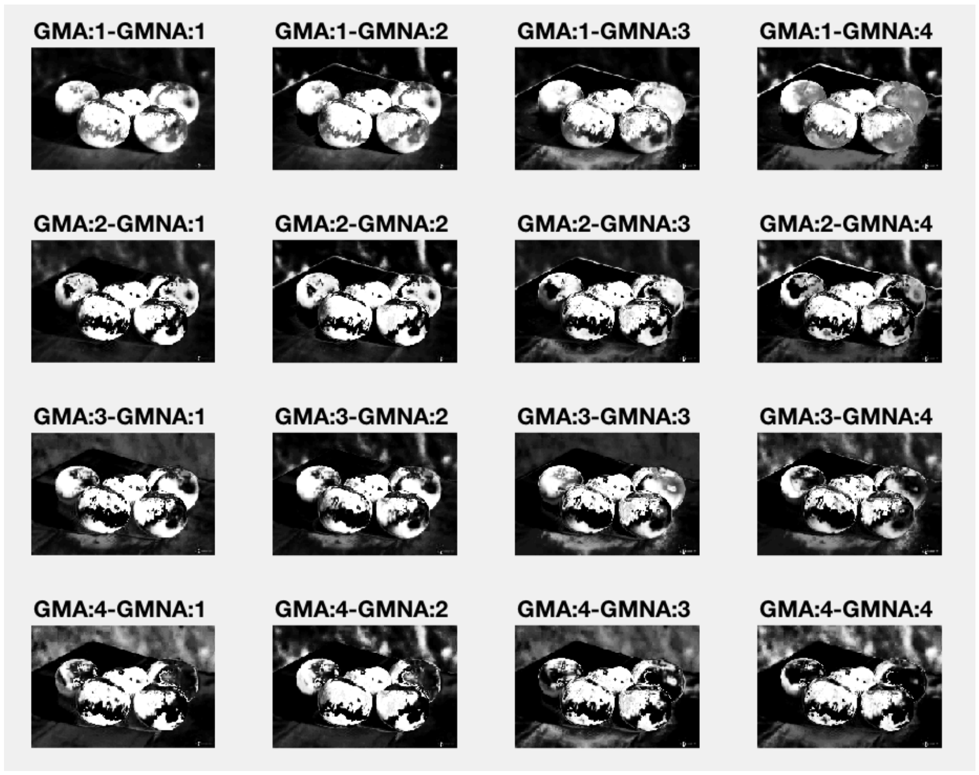
Image 2:



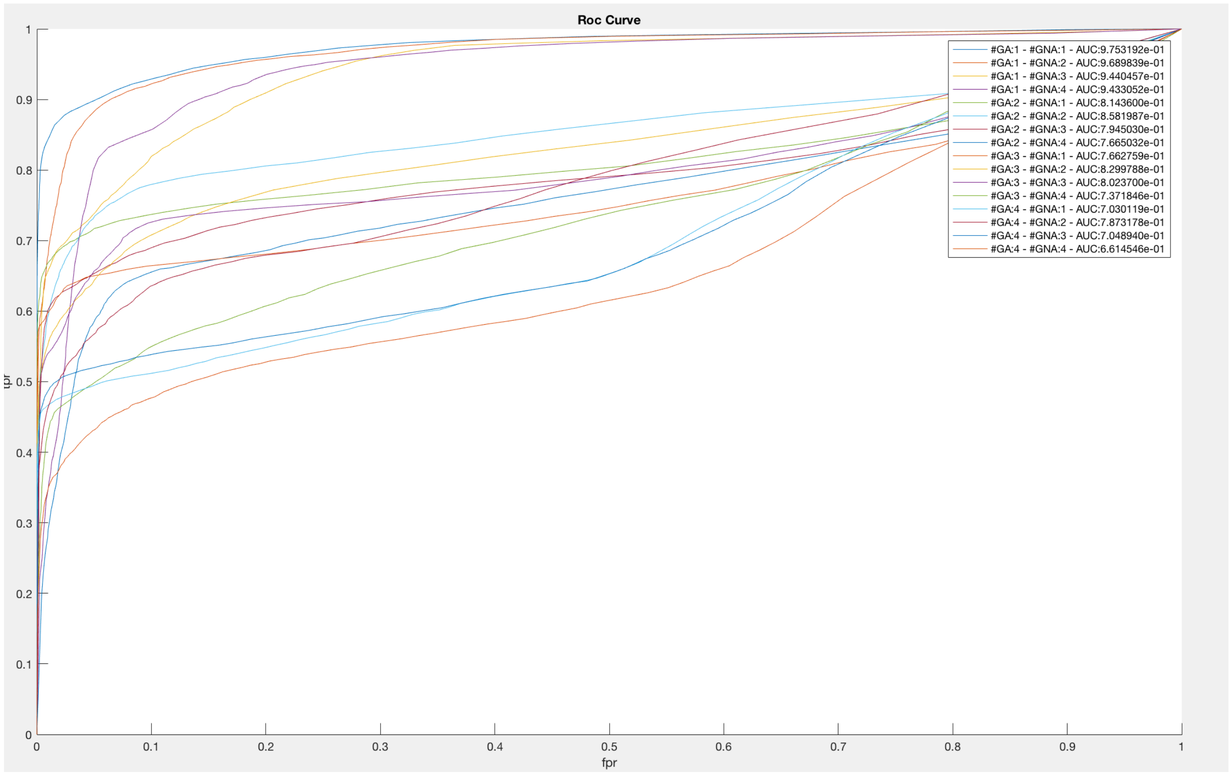
Mask for image 2:



Posteriors for Image 2 (figure 9):



ROC curves for image 2 (figure 10):



Results:
For the first image, the combination of Gaussians that produced the best results was 1 Gaussian for apple pixels and 3 Gaussians for non-apple pixels (with an AUC of 0.993). With respects to the second image, the combination of Gaussians that produced the best results was 1 Gaussian for apple pixels and 1 Gaussian for non-apple pixels (with an AUC of 0.975).
Observations:
Generally, the models were more successful in identifying the apple and non-apple pixels correctly in the first image than in the second image. This is obvious since in the first image the background greatly contrasted itself from the apples, and the apples used were all the same tone of green, which our models are used to seeing from our training data (however, even in the best case, the posteriors for the apples were gray and not completely white, suggesting once again that the models have a greater difficulty in identifying green apple pixels than red apple pixels).
The second image was chosen in purpose to be harder for the models to classify the pixels. In the picture, the apples are both bright red and green (tones which weren't seen often in the training data). In addition, part of the background is green (even though a darker tone then the one normally seen in apples). All of these characteristics can explain why the models had a harder time in correctly classifying the pixels. What, to be honest, I didn't expect was that the best combination of Gaussians was just 1 for both the apple and non-apple pixels. It seems that with

more Gaussians more pixels from the background were identified as being apple and more actual apple pixels were classified as being non-apple. To be more specific, with more Gaussians, the brighter green apple pixels were miss-classified (as being non-apple), and on the other hand, the darker green pixels from the background were considered as being apple.
These results just emphasize what was said above (in step E). In order to correctly validate and develop a model that generalizes the best for all of the different apples out there then a bigger validation (and also training set) would be needed.

F) We train our model in the training set, then we compare how well it does and perform the tuning of the parameters in the validation set, finally the test set allows us to have an idea how well our end result generalizes for new data. In this case, we have too little training data (even though it equates to a lot of pixels, only 3 images were used) and a very insignificant validation set to successfully train and tune a model which will generalize well for all of the different apples that exist. In addition, ideally the best way to train and validate a model would be to perform cross-validation. In this particular case the training and validation sets are combined and then broken into n different sets. Iteratively, we train our models in n-1 of theses sets and validate them in the remaining set (we finish when all of theses smaller sets have been both trained and validated on). This procedure is called n-fold cross validation. The tuned parameter will be the average of all of those that gave the best results in the different validation sets. Cross validation is better then just having a separate training and validation set because now we can train our model, which uses the estimated optimal parameter mentioned above (its estimated because we can't actually optimize a model that has been trained using the whole training and validation sets), on both the training and validation set, and a model with more training data is a superior model (especially if we have a small amount of training data, which was the case of this assignment).

Extra Credit:

- There are various ways to improve on the results obtained. Firstly, and the easiest, is to convert the color space from RGB to HSV/HSL. These color spaces are widely used in image recognition since you can usually obtain better information in these color spaces (color information, in rgb, is usually more noisy). For example, pixels belonging to shadow sections would not have as different values as those of the object itself, and therefore would be less frequently miss-classified. I implemented the function rgbtohsl (found in LoadData.m) which converts pixels from rgb to hsl however, unfortunately, I did not have enough time to properly test and compare the results.
- In addition, other models instead of mixture of Gaussians could be implemented. For example, instead of running the EM algorithm in a mixture of Gaussians, it could be used on a mixture of Student t-distributions (which has a wider covariance and hence allows for a wider range of color). The

results would obviously have to be compared to those obtained using normal distributions in order to understand if there is an actual improvement.

- Finally, and most importantly, I believe that in order to significantly improve these results, more then just the color of the apples needs to taken into consideration, the shape for example is another big factor. What I believe would show great improvement is using graphical models (such as grid models, and Markov random field) where the probability of a pixel being either apple or non-apple would be conditioned on its neighbor pixels. As a consequence, many of the current miss-classified pixels encountered in the middle of the apples would be influenced by their neighboring pixels (which are apple) and hence end up being correctly classified as apple.