Binary Image Classifier Report

CM20220 (Pattern Analysis)

Adam Jaamour (aj645)

Table of Contents

1. Abstract
2. Theory (Introduction)
   2.1. Introduction to classifiers
   2.2. Feature vectors, chain codes, Fourier Transform and filters
   2.3. Supervised GMM Training
   2.4. GMM Classification
   2.5. Confusion Matrices
   2.6. Classifier optimisation
3. Methodology
   3.1. Environment Setup
   3.2. Training
   3.3. Testing and Quality assessment
   3.4. Optimisation
4. Results
   4.1. Feature vector & Data matrix
   4.2. GMM
   4.3. Confusion Matrix and Quality
   4.4. Optimisation
   4.5. Extra testing
5. Conclusion
6. Bibliography

Note: The 10-page (with 10% extra allowed) limit is followed for content only, which goes from the abstract to the conclusion, excluding the title page, the table of contents and the bibliography.

1. Abstract

Pattern analysis is a branch of machine learning which focuses on recognizing patterns in data. This project focuses on an application of pattern analysis called classifiers, undertaking the objective of building one to classify binary-images. The classifier is built in MATLAB, a programming language specialised in numerical computing, being the main reason why it was chosen to build the classifier. Using supervised GMM training and classification, a set of approximately 220 binary images is used to train the classifier, which is then able to successfully classify over 80% of the images fed to it. The classifier itself never been 100% certain about the class of the input, it returns the class which is most likely to match the input. The quality of 83% achieved after optimising the classification within the code itself is consequently a solid result, meaning over four fifths of the input is correctly classified. This project includes a wide variety of mathematical theories and applications, which are covered throughout the theory and methodology used to build this classifier.

2. Theory (Introduction)

A precisely-followed succession of steps is required to create the binary-shape classifier. The first step consists in training the classifier using a set of training images, followed by testing that trained classifier using another set of testing images and calculating the likelihood of the test image being close to one of the classes the classifier can determine. The last step consists in assessing the overall quality of the classifier. All these steps are further analysed in the third part of the report (Methodology), but many aspects of the theory and the mathematics behind making the classifier function properly are discussed in this part, such as understanding how the classifier recognizes different binary shapes and how it can classify new shapes after having been trained.

2.1. Introduction to classifiers

A classifier is a computer program which takes some data *x*, 'learns' it and outputs a discrete class label *c* such as: $c = f(x)$. The learning role consists in analysing the input data *x* to create an inferred function *f*, corresponding to the training step, which can then be used in the testing step to output correct class labels for new input data. To go further with the mathematical definition of a classifier, it can be defined as a function with the following parameters: $i = classify(F(x), D)$ where x is a data point, $F(x)$ is the feature, D is the parameter and $i (\in \mathbb{Z})$ is the result label.

The input data *x* which is fed to the classifier can either be labelled or not. On one hand, labelled data such as: $D = \{(x_1, i_1), (x_2, i_2), \dots\}$ is used for supervised learning, while on the other hand, data with no labels such as: $D = \{x_1, x_2, \dots\}$ is used for unsupervised learning. A quick real-life example can be used to illustrate the difference between the two types of learning, assuming the classifier is a human being, the data to be labelled is a basket with different types of fruits, and the task is to arrange them per type. In the case of supervised learning, the human already knows the type of each fruit since the given data is labelled. This labelled data is usually discrete and comes in the form of a list of pairs, with each pair (x, i) consisting of an input object x and a desired output value i. In the case of unsupervised learning, the basket of fruits has four new types of fruits unknown to the classifier. He will therefore try to label them after having analysed features he could use to tell the fruits apart from each other such as size and colour (E. McNulty, 2015). Unsupervised learning being harder to implement, supervised learning is the chosen learning method to build this classifier.

2.2. Feature vectors, chain codes, Fourier Transform and filters

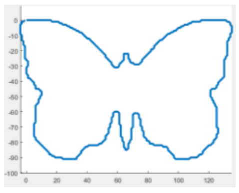
Feature vectors are used to both train and test the classifier. They are numerical representations of objects, which can be used in classification to recognize similar objects. Each of the input images have their own feature vectors characterizing them.

In this classifier, the features used for the input binary images are found by performing a succession of transformations and filtering, the first being a chain code transformation. Since the input is in binary images, meaning each pixel of the image can only take two different values (black and white), a chain code transformation can be applied to those images to create a numerical representation of the boundary of the shape in each binary image. The chain code itself is composed of a set of eight symbols (*right, left, up down, up-right, up-left, down-right, down-left*), each representing the direction the boundary of the shape is taking. These symbols are recorded as numbers (from 0 to 7), creating a signal. This means that the extremity of the shape in the binary image can be reconstructed from the chain code only, as shown in the images below:

Raw binary image

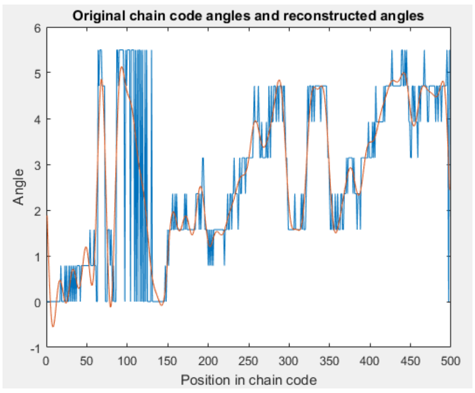
Image with boundary overlay

Reconstructed image

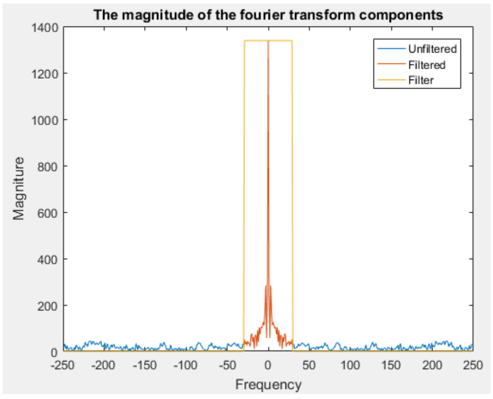


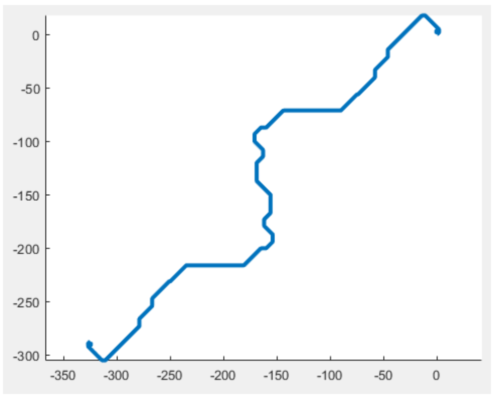
Once the chain code of an image is created, it can be converted to the frequency domain using the Fourier Transform (FT). The FT converts data from the time (signal) domain to frequency domain without any loss of information using the following formula: $F(\omega) = \mathfrak{F}[f](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} f(x)e^{-i\omega x}dx$. The formula being in the complex domain and complicated to implement, built-in functions in MATLAB are used to perform the FT and its inverse of signals.

Converting signals such as the chain code allows for new operations to be carried out which couldn't have been in the previous domain. In this case, a low-pass filter, which attenuates frequencies higher than a certain threshhold and keeps the lower ones, is applied to the chain code. This kind of operation, which is more efficient to carry out in the frequency domain than it is in the time domain, smoothens the signal by removing the short-term fluctuations and leaving the long-term trend (CM20219, lecture 4). It can be seen in the graph below that the chain code (blue) is not smooth before the low-pass filter is applied since it is based on pixels, but becomes smooth (orange) once the filter is applied:



The last step towards retrieving the feature vectors from the filtered chain code in the frequency domain consists in filtering it again using a top-hat filter. A top-hat filter has different uses in the real space and the frequency space, and in the latter, it selects a band of desired frequencies from the signal after specifying the lower and upper bounds. Applying the top-hat filter (yellow) to the filtered chain code creates the left graph below (red). At this point, attempting to reconstruct the original binary shape is impossible due to the filtering applied to the chain code. Because the top-hat filter gets rid of the data not "under" it, the chain code data which closes the shape is lost, causing the ends not to join up as depicted in the right graph below. The feature vector itselft corresponds to the part highlighted in red in the left graph, with only the absolute value and the real values retrieved since the signal is complex after the FT has been applied to it.





Adam Jaamour

2.3. Supervised GMM Training

The classifier can be trained using a multivariate Gaussian, which corresponds to a generalisation of a normal distribution to d dimensions: $N(x\,|\mu, C) = \frac{1}{(2\pi)^{d/2}\,|C|^{1/2}} \exp(-\frac{1}{2}\,(x-\mu)^T\, C^{-1}(x-\mu))$, where μ is the mean, C the covariance and d the number of dimensions ($d = 2$ in this case since the data corresponds to images). A Gaussian is fitted for each class, meaning the mean μ and the covariance C are determined for each class.

A Gaussian Mixture Model (GMM) $p(x)$ being the linear sum of N Gaussians, the Gaussians previously determined can be added together using $p(x) = \sum_{i=1}^{N} \alpha_i \times N(x|\mu_i, C_i)$. This is achieved by adding all the Gaussians together and calculating the prior α^C, or weight, of each Gaussian, which corresponds to the weight of each class c in the GMM. The prior is found by dividing the number of images for a class c by the total number of images M in the training set using formula (1):

$$(1)\quad \alpha^C = \frac{1}{M}\sum i\ [l_i = c]$$

For this binary-image classifier, a Maximum Likelihood Estimation (MLE) technique is used, meaning the Gaussian parameters mean μ_i and covariance C_i are found with the goal of maximising the sum of the likelihood for every image n over the entire training set N. They can be determined by using their usual formulae, and adapting it for each class of the GMM using the MLE, giving formula (2) for the mean and formula (3) for the covariance:

$$(2)\quad \mu_i = \frac{1}{N}\sum n\ d_n \qquad\qquad (3)\ C_i = \frac{1}{N}\sum n\ (d - \mu_i)^T (d-\mu_i).$$

The formula to calculate the covariance can be broken down into two steps (Stat Trek, 2017). The first one consists in calculating the deviation matrix, defined in formula (4), and then calculate the covariance matrix, defined in formula (5), by dividing each element of the deviation matrix by the sum of squares matrix by the number of rows in the size of the feature vector. x corresponds to the feature vector

$$(4)\ dev = x - \frac{ones \times x}{n}) \qquad\qquad (5)\ C = \frac{dev^T \times dev}{n-1}$$

Once the prior, mean and covariance have been determined for each class, the likelihood $p(x|\theta_i) = \frac{1}{(2\pi)^{K/2}\,|C_i|^{1/2}} \exp(-\frac{1}{2}\,(x-\mu_i)^T\, {C_i}^{-1}(x-\mu_i))$ can be found, which corresponds to the probability of a an image corresponding to a class. Note that the first different $(d - \ \mu_i)^T$ is transposed instead of the second because of the vectors are stored in a row. The combination of the parameters for each Gaussian constitute the GMM, which is the data used to classify new images from the testing set.

2.4. GMM Classification

Given a new test point/image x, the classifier should be able to find the class with the highest posterior likelihood. This is done using a Maximum a Posteriori (MAP) classification. For each new image from the testing set, a class which maximises the posterior $p(x|\theta_i)$ is selected, corresponding to the class that the classifier outputted based on its input. This is done using: $\arg\max[p(\theta|x)] \propto \arg\max[\alpha^\theta \times p(x|\theta)]$, where $p(x|\theta)$ is equal to the multivariate Gaussian for image x and class θ. The likelihood can be calculated using formula (6), where α_θ, μ_θ and C_θ are the prior, mean and covariance for a class θ

$$(6)\quad \arg\max p(\theta\,|x) = \arg\max[(\log\alpha_\theta) - (\tfrac{1}{2}\log|C_\theta|) - (\tfrac{1}{2}(x-\mu_\theta)^T)\ {C_\theta}^{-1}\ (x-\mu_\theta)].$$

"*arg max*" is used to only keep into account terms depending on the class θ, while "log" is used to prevent the likelihoods of being rounded to zero when stored on a computer.

2.5. Confusion Matrices

Confusion matrices are used to assess the quality of the classifier during the testing step. A confusion matrix is a table of numbers which is filled each time an image from the testing set has been classified. The columns correspond to the different classes which can be outputted by the classifier (the classes it has been trained to identify), while the rows correspond to the classification returned after an input image has been analysed by the classifier.

This confusion matrix can be achieved by separating the testing set into the same groups that were used during the supervised GMM training. Each testing group represents a row of the matrix, while the classification returned by the classifier represents a column on this specific row. For example, in the case of the binary-image classifier which has been trained to classify images from six different classes, if an image from the group 'Alien' is classified as 'Alien', then the element on row: 'Alien' and column: 'Alien' of the matrix is incremented. If another image from the group 'Alien' is misclassified as 'Butterfly', the element on row: 'Alien' and column: 'Butterfly' is incremented.

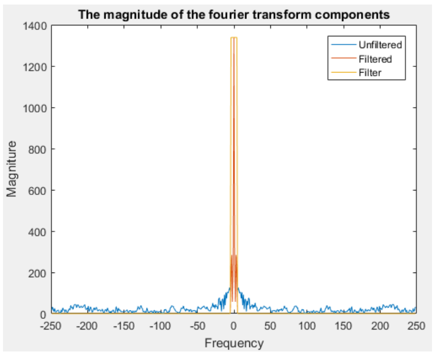
After the entire testing set has been classified, the matrix can be analysed to assess the classifier's quality. The number of images correctly classified can be found in the leading diagonal of the matrix, while misclassified images are scattered outside that diagonal. Formula (7) can be used to assess the quality of the classifier in the form of a percentage, where sum(c) is the sum of all correctly classified images in the diagonal of the matrix:

$$(7)\quad quality = \frac{sum(c)}{N} \times 100$$

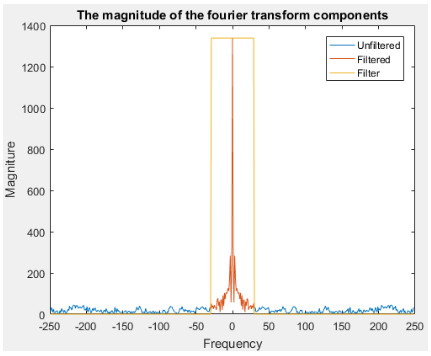
2.6. Classifier optimisation

The size of the feature vector is key to achieving a high-quality classifier with over 80% of the images from the testing set correctly classified. Indeed, the feature vector is the element used to train different classes of the GMM and then classify new images by comparing the features with the ones of each class. Therefore, the feature can neither be too short, nor too long. If it is too short, then not enough data will be used to train the GMM, whereas if it is too big, the features for different classes will start looking alike with a peak at the centre and decreasing magnitudes away from the centre, causing more misclassifications. This is shown in the three features (red) below. The first one shows the feature vector of the image *Butterfly001* with a feature of size $N = 5$, the second one the feature vector of the same image with a size $N = 30$, and the last one the feature vector of the image *Face008* with a feature of size $N = 30$. For $N = 5$ there is a lack of information, whereas for $N = 30$ the two different images' features look similar, especially towards the zero frequency where they both have a powerful magnitude peak. When the feature becomes too long, an issue referred to as the "curse of dimensionality" occurs, causing the classifier's performance to decrease after a certain length due to the accumulation of data making harder for the algorithm to process efficiently.

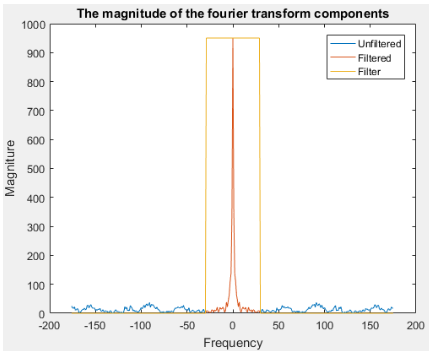
Butterfly, $N = 5$



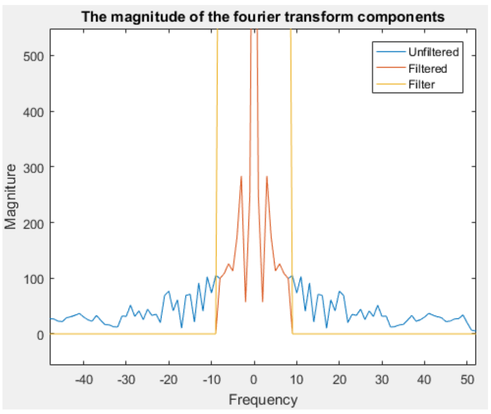
Butterfly, $N = 30$



Face, $N = 30$



Nonetheless, smaller feature vectors are more efficient than bigger feature vectors since the key elements characterizing an image's feature vector can be found at around the center of the peak (not the peak itself). Therefore to further optimise the classifier, the features can be retrieved starting three frequencies to the right of the zero frequency (peak), up to the $N + 3$ frequency, for feature vector of size N.



Additionally, it can be seen that the magnitudes are symmetric about the zero frequency, therefore the absolute value of the signal can be used to get more data while keeping the same feature vector length. In the case of the example above, the top-hat filter would be applied starting at the third frequency and go up to N-frequency.

3. Methodology

Now that all the theory and mathematics aspects needed to understand the functioning of the binary-image classifier have been covered, the different steps undertaken which go towards building one can be covered.

Note: the code snippets in this section represent only trimmed parts of the submitted code. Only the most important sections of the code are shown in the snippets.

3.1. Environment setup

Before actually training and testing the classifier, the environment must be setup. In the project directory, the code used to train and test the classifier is stored in the "*src*" folder, while the images are stored in the "*images*" folder and separated in three different sub-folders, one corresponding to the training set, another to the testing set, and the last one to a set of extra test cases.

The entire code is run from one main script called "*script.*m" which starts by storing all the different directories used in strings, as shown below, then trains the classifier, tests it, assesses its quality and performs extra test cases.

```
%% Image paths
imagedir = '../images';
verifyImageDir(imagedir);
imagedir_train = [imagedir '/train'];
imagedir_test = [imagedir '/test'];
imagedir_extra = [imagedir '/extra'];
```

3.2. Training

The steps in the training part follow the mathematics explained in sections 2.2 and 2.3 of the report. The script calls the "*train.*m" function, passing two arguments to the function. The first is the path to the testing set of images and the second a variable called N, which corresponds to the number of lower frequencies to keep for each feature vector. This means the argument N represents the length of each feature vector to retrieve.

```
%% Train classifier with training set of binary images
N = 9; % = number of lowest frequencies to keep
train(imagedir_train, N);
```

"*train.*m" fits a multivariate Gaussian for each class found in the supervised data (in the "*images/training*" folder) using the provided "*getClasses.m*" function. The Gaussians are fitted by calculating the mean, the covariance and the prior of the features of each class in a *for* loop, which goes through each class. The features are retrieved by calling the "*getDataMatrix.m*" function, which creates a matrix storing all the features of length N for a given class. The function returns a matrix, shown in section 4.1, with features stored in the columns and each row representing a shape:

```
function [D] = getDataMatrix(imagedir,class,N)
    imagelist = dir(sprintf('%s/%s*.gif', imagedir, class));
    for idx = 1:length(imagelist)
        imagepath = sprintf('%s/%s', imagedir, imagelist(idx).name);
        D(idx, :) = getFeatures(imagepath, N)'; %data matrix returned
    end
end
```

The features retrieved from the output of the "*getDataMatrix.m*" function are created using the "*getFeatures*" function. This function uses the same steps described in section 2.2 of the report. It starts by converting the input binary image from the training set into logical 0's and 1's, then calculates the chain code signal. It then applies the Fast Fourier Transform to the signal and filters it using a top-hat filter. The values chosen for the filter are explained in sections 2.6 and 3.4, since they correspond to quality optimisation. Once the filter has been applied, the real values of the absolute value of the filtered chain code constitute the feature vectors, as mentioned in section 2.2.

```
function [ features ] = getFeatures( image_path , N)
    im = imread(image_path);
    im = logical(im); % Convert intensity values to 1s and 0s

    %   calculate chain code
    c = chainCode(im);

    % Filter using the FT of the angles of the chaincode
    angles = c(3,:)*(2*pi/8);
    anglesFFT = fft(angles); %fft

    % top-hat filter
    filter = zeros(size(angles));
    filter(3:N+2) = 1;

    % A pply the filter by scalar multipliacation
    filteredFFT = anglesFFT .* filter;

    % absolute valye
    absFiltered = real(abs(filteredFFT));

     % transpose features
    features = (absFiltered(3:N+2))';
end
```

Once each feature of the images for a given class have been stored in the data matrix, the "*train*.m" function calculates the mean, covariance and prior of each classes' data matrix.

The mean, which corresponds to the average of all elements in a column, is calculated using formula (2) (specified in section 2.3) with the "*calcMean.m*" function below. It returns a row vector the length of the number of feature vectors, or *N*, with each value equal to the average of all elements in a column of the data matrix.

```
function [mean] = calcMean(data)
    columns = size(data, 1); % # of feature vectors in input
    rows = size(data, 2);    % # of data points in input
    mean = zeros(1, rows);   % initialize mean vector
    for i = 1:rows
        total = 0; % sum of all elements in a column
        for j = 1:columns
            total = total + data(j,i);
        end
        mean(1,i) = total/columns;
    end
end
```

The covariance is calculated using formulas (4) and (5) (specified in section 2.3) with the "*calcCov.m*" function. It returns a square matrix the size of the number of features. The leading diagonal shows the covariance matrix corresponds to the variance between the features of the images of a same class from the training.

```
function [covariance] = calcCov(data)
    % size of input matrix column
    columns = size(data,1);
    % square matrix of 1's with size of column of input matrix
    one = ones(columns,columns);

    % deviation matrix
    deviation = data - (one*data/(columns));
    % covariance matrix
    covariance = (transpose(deviation) * deviation)/(columns-1);
end
```

The prior is the last step towards having a complete GMM. It is calculated using formula (1) (section 2.3) within the "*train.m*" function. Corresponding to the weight of a class in the GMM, it can be found by dividing the number of training images for a given class by the number of images in the training set:

```
for idx = 1:length(classes)
    class = classes{idx};
    models(idx).name = class;
    dataMatrix = getDataMatrix(imagedir, class, N);
    models(idx).mean = transpose(calcMean(dataMatrix));
    models(idx).cov = ensurePSD(calcCov(dataMatrix));
    models(idx).prior = ((getNumImagesForClass(imagedir,class)) / totalImages);
end
save('models');
```

When "*train.m*" finishes running for each class, it stores each Gaussian in a structure named "*models.mat*" corresponding to the GMM, which can be found in section 4.2. This is the GMM used to classify the testing set of images.

3.3. Testing and Quality assessment

At this point the classifier can be considered as functional but it is yet to be tested with a new set of images to verify that it works correctly and to assess its quality. This can be done by using the GMM which has just been fitted previously. After having called the training function, the main script next calls "*getConfusionMatrix.m*", which is a function that classifies all the images in the testing set and assesses the overall quality of the classifier using confusion matrices.

```
%% Test classifier with new testing set of binary images
confusion_matrix = getConfusionMatrix(imagedir_test);
disp(confusion_matrix);
```

"*getConfusionMatrix.m*" loops through each image in the testing set and attempts to classify them, then increments the appropriate element in the confusion matrix.

The function makes use of the provided function "*classify.m*", which follows the Maximum a Posteriori (MAP) classification technique described in section 2.4. Using the formula (6) from section 2.4, the feature of each new image from the testing set is compared with each class the classifier can classify, and the class which maximises the posterior for the feature of the input image is returned by the function:

```
function [classname] = classify(imagepath)
    load('models'); % trained GMM
    N = length(models(1).mean);  %Assume models use the same number of features
    features = getFeatures(imagepath, N); % feature of input image
    maxscore = -inf;

    for idx = 1:length(models) % which class has the highest score
        model = models(idx);

        score = log(model.prior) - 0.5*(log(det(model.cov))) -
                0.5 * (transpose(features-model.mean)) *
                (inv(model.cov)) * (features-model.mean);

        % keep class which maximises posterior
        if score > maxscore
            maxscore = score;
            bestidx = idx;
        end

    end
    classname = classes(bestidx); % classification returned
end
```

Each returned classification by the "*classify.m*" function is next used to increment the confusion matrix. This is done by creating three nested for loops. The first loop goes through the six classes and separates the images from the testing set into groups corresponding to those classes. The second loop classifies each image from the groups in the testing set and compares the classification returned by "*classify.m*" with the classes in the third loop. This is where the confusion matrix slot, with the row corresponding to the testing group and the column corresponding the classification returned, is incremented.

```
classes = getClasses(imagedir);
numClasses = size(classes,2);
totalImages = getNumImages(imagedir);
accumulator = 0;
confusion_matrix = zeros(numClasses,numClasses); %initialize
for i = 1:numClasses % loop through each class
    imagelist = dir(sprintf('%s/%s*.gif', imagedir, classes{i}));
    length_list = size(imagelist,1); % # of images in created list

    for j = 1:length_list % loop through each image in class i
        imagepath = sprintf('%s/%s', imagedir, imagelist(j).name);
        classification = classify(imagepath); %classify images individually

        for k = 1:numClasses % increment classification matrix
            if(strcmp(char(classification),classes{k}))
                confusion_matrix(k,i) = confusion_matrix(k,i) + 1;
                break;
            end
        end

    end
end
```

Once the entire testing set has been classified, the quality of the classifier can be determined. As mentioned in section 2.5, the images correctly classified are found in the leading diagonal of the confusion matrix, while misclassified images are scattered over the rest of the matrix. To assess the overall quality of the classifier, formula (7) can be used to calculate the sum of all the values in the diagonal, divided by the total number of images that have been classified:

```
% total images which were correctly classified
for l = 1:numClasses
    accumulator = accumulator + confusion_matrix(l,l);
end

% calculate final classifier quality with current testing set
score = (accumulator / totalImages)*100;
disp(['Confusion matrix score = ' int2str(score) '%']); disp(' ');
```

The output of "*getConfusionMatrices.m*" can be found in section 4.3, with the confusion matrix and the overall quality of the classifier.

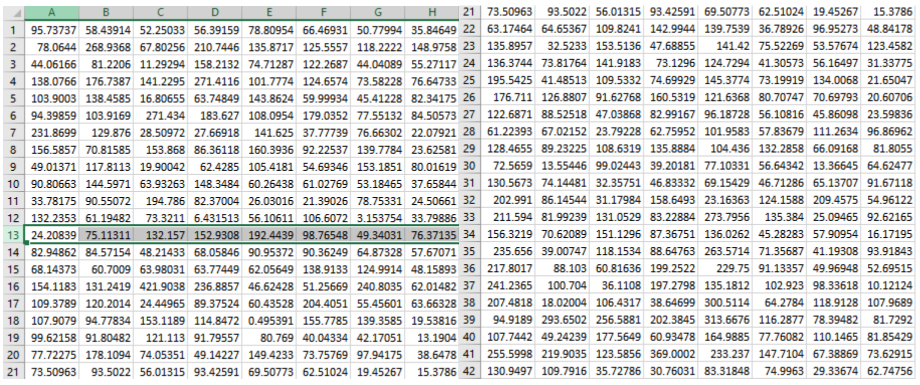
3.4. Optimisation

Once all the previous steps have been carried out, the classifier is fully functional and the confusion matrix is used to output the overall quality. After the first run, the quality of the classifier is at approximately 66%, which can be improved. This is due to the argument N which is passed at the beginning of the script in the "*train*.m" function. As explained in section 2.6, N corresponds to the vector length. A quality of 62% is achieved when $N = 30$, which is too long. Therefore, to optimise the classifier, the size of the feature vectors should not be too long nor too short. After plotting a function in Excel to see for which feature vector the classifier has the best quality, the optimal size found is $N = 8$, with the results shown in section 4.4.

4. Results

4.1. Feature vector & Data Matrix

The "*train.m*" function calls "*getDataMatrix.m*" for each class, which also calls "*getFeatures.m*" for each image in that class. This results in a matrix of size $x \times N$, where x corresponds to the number of input images for a given class, and N to the length of the features to retrieve. In this example, the data matrix retrieved corresponds to the one for the '*Alien*' class, which has 42 images, for features of size 8. The highlighted row corresponds to the feature vector of size 8 for the 13th image of the class '*Alien*'.

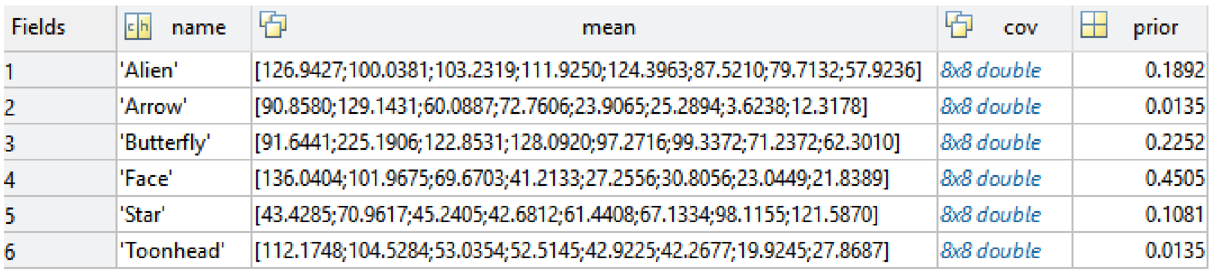


	A	B	C	D	E	F	G	H
1	95.73737	58.43914	52.25033	56.39159	78.80954	66.46931	50.77994	35.84649
2	78.0644	268.9368	67.80256	210.7446	135.8717	125.5557	118.2222	148.9758
3	44.06166	81.2206	11.29294	158.2132	74.71287	122.2687	44.04089	55.27117
4	138.0766	176.7387	141.2295	271.4116	101.7774	124.6574	73.58228	76.64733
5	103.9003	138.4585	16.80655	63.74849	143.8624	59.99934	45.41228	82.34175
6	94.39859	103.9169	271.434	183.627	108.0954	179.0352	77.55132	84.50573
7	231.8699	129.876	28.50972	27.66918	141.625	37.77739	76.66302	22.07921
8	156.5857	70.81585	153.868	86.36118	160.3936	92.22537	139.7784	23.62581
9	49.01371	117.8113	19.90042	62.4285	105.4181	54.69346	153.1851	80.01619
10	90.80663	144.5971	63.93263	148.3484	60.26438	61.02769	53.18465	37.65844
11	33.78175	90.55072	194.786	82.37004	26.03016	21.39026	78.75331	24.50661
12	132.2353	61.19482	73.3211	6.431513	56.10611	106.6072	3.153754	33.79886
13	24.20839	75.11311	132.157	152.9308	192.4439	98.76548	49.34031	76.37135
14	82.94862	84.57154	48.21433	68.05846	90.95372	90.36249	64.87328	57.67071
15	68.14373	60.7009	63.98031	63.77449	62.05649	138.9133	124.9914	48.15893
16	154.1183	131.2419	421.9038	236.8857	46.62428	51.25669	240.8035	62.01482
17	109.3789	120.2014	24.44965	89.37524	60.43528	204.4051	55.45601	63.66328
18	107.9079	94.77834	153.1189	114.8472	0.495391	155.7785	139.3585	19.53816
19	99.62158	91.80482	121.113	91.79557	80.769	40.04334	42.17051	13.1904
20	77.72275	178.1094	74.05351	49.14227	149.4233	73.75769	97.94175	38.6478
21	73.50963	93.5022	56.01315	93.42591	69.50773	62.51024	19.45267	15.3786
22	63.17464	64.65367	109.8241	142.9944	139.7539	36.78926	96.95273	48.84178
23	135.8957	32.5233	153.5136	47.68855	141.42	75.52269	53.57674	123.4582
24	136.3744	73.81764	141.9183	73.1296	124.7294	41.30573	56.16497	31.33775
25	195.5425	41.48513	109.5332	74.69929	145.3774	73.19919	134.0068	21.65047
26	176.711	126.8807	91.62768	160.5319	121.6368	80.70747	70.69793	20.60706
27	122.6871	88.52518	47.03868	82.99167	96.18728	56.10816	45.86098	23.59836
28	61.22393	67.02152	23.79228	62.75952	101.9583	57.83679	111.2634	96.86962
29	128.4655	89.23225	108.6319	135.8884	104.436	132.2858	66.09168	81.8055
30	72.5659	13.55446	99.02443	39.20181	77.10331	56.64342	13.36645	64.62477
31	130.5673	74.14481	32.35751	46.83332	69.15429	46.71286	65.13707	91.67118
32	202.991	86.14544	31.17984	158.6493	23.16363	124.1588	209.4575	54.96122
33	211.594	81.99239	131.0529	83.22884	273.7956	135.384	25.09465	92.62165
34	156.3219	70.62089	151.1296	87.36751	136.0262	45.28283	57.90954	16.17195
35	235.656	39.00747	118.1534	88.64763	263.5714	71.35687	41.19308	93.91843
36	217.8017	88.103	60.81636	199.2522	229.75	91.13357	49.96948	52.69515
37	241.2365	100.704	36.1108	197.2798	135.1812	102.923	98.33618	10.12124
38	207.4818	18.02004	106.4317	38.64699	300.5114	64.2784	118.9128	107.9689
39	94.9189	293.6502	256.5881	202.3845	313.6676	116.2877	78.39482	81.7292
40	107.7442	49.24239	177.5649	60.93478	164.9885	77.76082	110.1465	81.85429
41	255.5998	219.9035	123.5856	369.0002	233.237	147.7104	67.38869	73.62915
42	130.9497	109.7916	35.72786	30.76031	83.31848	74.9963	29.33674	62.74756

This data matrix is used, along with the five other data matrices for the other classes, to create the GMM.

4.2. GMM

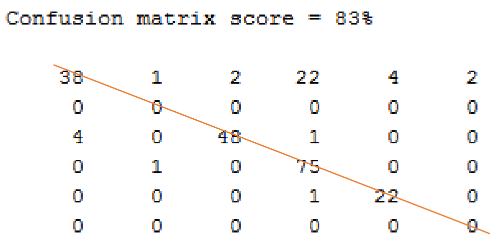
The GMM created after training is stored in a MATLAB structure, with fields (columns) for the class name, the classes' mean, covariance and prior. Each row represents the Gaussian fitted for a single class.



Fields	name	mean	cov	prior
1	'Alien'	[126.9427;100.0381;103.2319;111.9250;124.3963;87.5210;79.7132;57.9236]	8x8 double	0.1892
2	'Arrow'	[90.8580;129.1431;60.0887;72.7606;23.9065;25.2894;3.6238;12.3178]	8x8 double	0.0135
3	'Butterfly'	[91.6441;225.1906;122.8531;128.0920;97.2716;99.3372;71.2372;62.3010]	8x8 double	0.2252
4	'Face'	[136.0404;101.9675;69.6703;41.2133;27.2556;30.8056;23.0449;21.8389]	8x8 double	0.4505
5	'Star'	[43.4285;70.9617;45.2405;42.6812;61.4408;67.1334;98.1155;121.5870]	8x8 double	0.1081
6	'Toonhead'	[112.1748;104.5284;53.0354;52.5145;42.9225;42.2677;19.9245;27.8687]	8x8 double	0.0135

4.3. Confusion Matrix and Quality

The confusion matrix depicts the quality of the classifier. The next screenshot shows the command line output of the main script. The orange line corresponds to images from the training set which have been correctly classified using the "*getConfusionMatrix.m*" function.

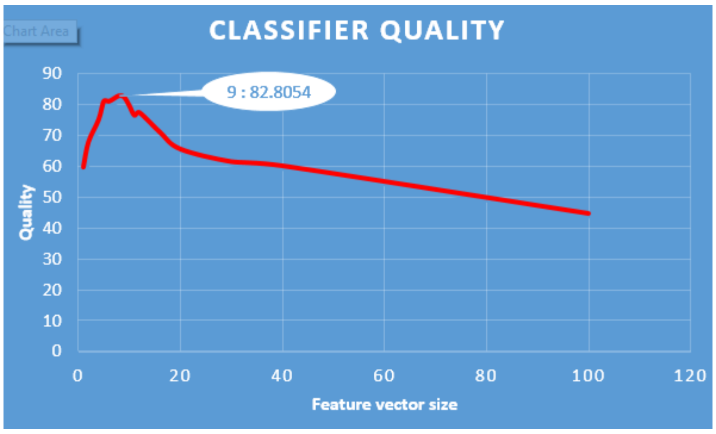


```
Confusion matrix score = 83%

   38       1       2      22       4       2
    0       0       0       0       0       0
    4       0      48       1       0       0
    0       1       0      75       0       0
    0       0       0       1      22       0
    0       0       0       0       0       0
```

4.4. Optimisation

To determine the feature vector size which provided the highest quality score, the classifier was run with feature vector sizes varying between 2 to 100. The returned quality score was then added to a table in Excel for each of the tested sizes, and then plotted to draw a function in red in the screenshot below. The optimal feature vector size was then determined by taking the peak of the function, as shown below:



4.5. Extra testing

Some additional tests were carried out on the classifier. The first one consisted in classifying a binary-image shape which isn't part of the six original classes. The second one consisted in classifying a binary-image which was used in the training set. Screenshots of the command line show the classifier's output:

```
Test 1. Image not in sets of classes. Class = Hand. Classified as :  Alien

Test 2. Image from training set. Image = Arrow001. Classified as :  Arrow
```

The hand was classified as an '*Alien*' while the Arrow from the training set was classified as an '*Arrow*'.

5. Conclusion

Different steps were undertaken in this project to build a binary-image classifier. From training the classifier by fitting a Gaussian Mixture Model based on each image's feature vector, to classifying new sets of testing images by comparing their feature vectors with the data from the Gaussian Mixture Model and finally assessing the classifier's quality using confusion matrices, each of these steps required precision and organization during the development process to ensure the highest possible quality classifier.

Apart from teaching me more about pattern analysis with the branch of classifiers, this project greatly reinforced my mathematical skills with concepts such as vectors, matrices, Gaussians, real & complex domains in general, but especially feature vectors, multivariate Gaussians, Gaussian Mixture Models, Maximum Likelihood Estimation, Maximum a Posteriori and confusion matrices for supervised GMM training and GMM classification. Binary images and Fourier Transforms from the time to the frequency domains also contributed greatly from a scientific applied point of view. In terms of technical aspects, applying the all mathematics previously stated into a practical environment with MATLAB to create a functional project also contributed towards my software developer skills.

Achieving a binary-image classifier with a quality of 83% required a lot of time and brainstorming implementing the provided functions with personal code. However, if time management wasn't an issue, I would have attempted at improving the classifier to work with unsupervised data instead of supervised data only.

6. Bibliography

CM20219 - (Lecture 4 – The Fourier Transform)

CM20220 - Lab 2 (Fourier Transform)

CM20220 - Lab 3 (Features and Chain code)

CM20220 - Lab 4 (Gaussian Mixture Models)

CM20220 - Lab 5 (Training & Testing)

CM20220 - Lectures 6, 7 & 8 (Classifiers)

Matlab logo. (2008). [image] Available at: https://en.wikipedia.org/wiki/MATLAB#/media/File:Matlab_Logo.png [Accessed 28 Apr. 2017].

McNulty, E. (2015). *What's The Difference Between Supervised and Unsupervised Learning?*. [online] Dataconomy. Available at: http://dataconomy.com/2015/01/whats-the-difference-between-supervised-and-unsupervised-learning/ [Accessed 29 Apr. 2017].

Stat Trek. (2017). *Variance-Covariance Matrix*. [online] Available at: http://stattrek.com/matrix-algebra/covariance-matrix.aspx [Accessed 17 Mar. 2017].