CM30080: Computer Vision Coursework

Patrick Millais

April 25, 2017

Introduction

This report details the development of a basic correlation detector, allowing faces to be detected in our input image. We take a breadth-based approach to optimise the detector, which achieves a 67.92% face detection rate. The second half of this report focuses on using a basic, and then improved, classifier to detect specific faces inside our input image. Optimisations and directions for future work supplement the final requirement.

1 Requirement #1: Correlation Detector

We begin with the basic correlation detector provided, which enables users to either select or use a pre-defined template for matching. The detector then uses non-maximal suppression to detect the maximums over the image. This is used in conjunction with the final function – *EvaluateDetections* – effectively suppressing image information which is not applicable to our detector, and then evaluating our template detection in the context of the Viola-Jones face detection success rate.

Initially we passed in several different correlation windows to the detector. Two points immediately became clear:

- The basic detection rate was relatively poor. Most of our testing correlations had a 15-20% overlap with Viola-Jones detection.
- The detection rate was, unsurprisingly, highly dependent on the template initially selected.

Accordingly we set the *selectRegion* flag to false in *FaceDetect.m*, enabling a consistent template to be used throughout Requirements 1 and 2. This was the face of Barack Obama which in the basic detector achieves 13.39% overlap with Viola-Jones detection. Consequently, the results of our detector will not be dependent on the template selected. Rather, a consistent template will be used allowing results to be accurately compared during our breadth-based approach.

The next step was to implement normalised correlation in place of the dot product approach used in the basic detector. We first defined the basic formula for normalising an input image X:

$Y = \frac{X - \bar{X}}{\sigma}$

and translated this into executable code in Matlab:

```
function normalisedImg = normalise(img)
    normalisedImg = (img - mean2(img))./std2(img);
end
```

Listing 1: Normalise Function

We then defined the strategy for reaching a matrix of summed dot products of the normalised template and the appropriate regions of the G20 image. This is represented by Figure 1, in which we *slide* the normalised template over the G20 image. The main image region under the template (the *patch*) is calculated through the dimensions defined as x, y, t_x, t_y, and then normalised. The dot product of the normalised template and the normalised patch is computed, and finally the sum of this dot product is returned into the output matrix.

In our program this is represented as follows:

```
% Slide the template over the entire image and normalise the area
    of the
% image relative to the template (the patch)
for i = 1:(x - t_x)
    for j = 1:(y - t_y)
        % Get the normalised patch of the image
        normalisedPatch = normalise(imf(i : (i+t_x)-1, j : (j+t_y)
    -1, :));

        % Get the dot product of the template and patch
        dp = dot(normalisedTemplate, normalisedPatch);

        % Sum the values in the dot product and add to our output
        output(i + floor(t_x/2), j + floor(t_y/2)) = sum(dp);

    end
end
```

Listing 2: Sliding Normalisation

Importantly this process only normalises the patch under the template. For instance, when calculating the mean of the patch it does not take the entire image into account, just the region which is bound by t_x and t_y. If the entire image was taken into account this would have a reduced time complexity, from $\mathcal{O}(n^2)$ to $\mathcal{O}(1)$ as it would be a simple one-time operation to compute the normalisation of the entire image. This approach, however, would be incorrect. Significantly we are only interested in normalising the patch region, and taking the mean and standard deviation from the patch region. We are not interested in image information from outside this region when normalising.

Nevertheless, this is compute-heavy approach. We are moving the template on a pixel-by-pixel basis across the x and y dimensions of the G20 image, repeating the process until the last pixel has been reached. We can somewhat alleviate the amount of computation by pre-computing the normalised template outside of the nested for loop, so that this is not repeatedly computed. However, the repeated patch normalisation is relatively compute-intensive in this $\mathcal{O}(n^2)$ algorithm. A parallelised approach where each thread, or core, normalises a sub-region of the image and simultaneously writes to distinct indexes of the output matrix is feasible, and would improve upon our approach.

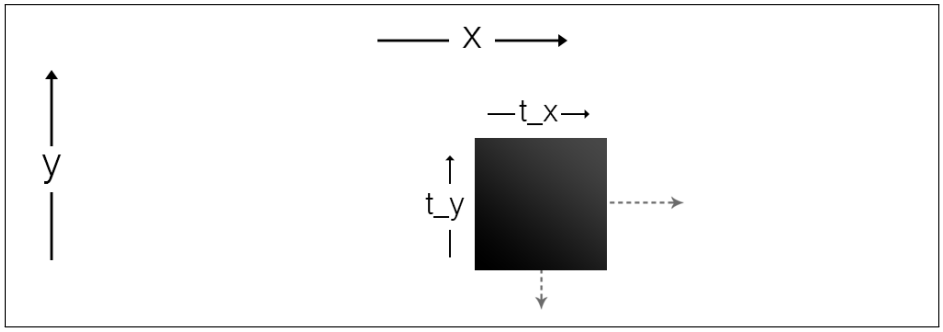

Figure 1: The Sliding Normalisation Process

The purpose of this process is to control the intensity of pixels (in terms of brightness) and compensate for variations across the selected patches. This is completed via subtracting the mean and dividing by the standard deviation respectively, smoothing the image and reducing noise. This had a positive effect on our correlation detector. From the original face detection result of 13.39%, the algorithm described above increases this to 21.43%. Our final face detection rectangles for this requirement are displayed in Figure 1. The following section describes the breadth-based approach we took to improve this towards a success rate of 67.92%.

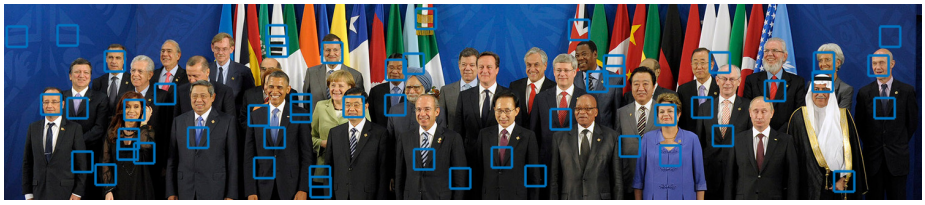

Figure 2: Normalised correlation face detector with a 21.43% success rate

2 Requirement #2: Improved Detector

This section investigates various pre-processing techniques which we used to improve and optimise our face detector. We start with a simple inspection of detection rates across different RGB channels, before delving into edge detection, noise reduction through Gaussian and Averaging kernels, and edge thinning. Our final detector improves upon our original detector by 217%, with a final detection rate of 67.92%.

2.0.1 RGB Channels

Our exploration begins by separating the G20 image into its constituent colour channels: red, green and blue. Our hypotheses was that certain colour channels would lend itself to our face detection methodology by highlighting specific facial features. In particular, the red channel in Figure 3(a) shows the greatest contrast between the skin tone and surrounding environment.

(a) Red Channel (b) Green Channel (c) Blue Channel

Figure 3: RGB Constituent Channels

Matlab makes it simple to access these channels of information:

```
image = image(:,:,i);
```

where i is $\{1, 2, 3\}$ representing Red, Green and Blue respectively. We then ran the detector with the main image and template configured to certain RGB channels. Our hypotheses were validated in the results below:

Table 1: RGB Channel Results

Colour Channel	**Detection %**
Red	38.90
Green	16.55
Blue	16.95

Pre-processing the image to a red channel before normalisation is clearly beneficial for the face detection success rate. While the images included in Figure 4 are visually similar, there are some subtle differences which affect this detection rate. This includes more prominent face edges and a reduction in less prominent features (such as hair) causing false positives. The normalisation

process benefits from an input image in which the face skin-tone has a greater contrast against the background, enabling the significant shape of the faces to be highlighted and emphasised.

(a) Red Channel Normalisation (b) RGB Normalisation

Figure 4: RGB and Red Channel Normalisation Comparison

2.0.2 Edge Detection

The benefits of selecting the red channel in the section above was that, through the virtue of its natural visual properties, it explored a basic and naive form of edge detection. This enabled us to speculate about how a more advanced form of edge detection might work before the face detector completed the normalisation process.

Given that we were exploring this requirement in a breadth-based manner, rather than a deeper exploration of certain functionality, we did not build our own edge detector. Instead we looked towards the *edge()* function which is included with Matlab, and how we could optimise the parameters to our advantage. This function was ultimately applied to both the G20 and template images.

We began by applying the *edge* function to the grayscale G20 image and template as below:

```
% Get the edges of the image & template.
imf = edge(imf, 'Canny');
template = edge(template, 'Canny');
```

Listing 3: Edge Detection

The edge function returns a binary image, with 1s for the edges and 0s for non-edges. We used the *Canny* parameter as a starting method for edge detection. We presumed that its use of two separate thresholds[1] would result in a larger number of edges being detected – not only *strong* edges, but also interconnected *weaker* edges as well.

[1] http://uk.mathworks.com/help/images/ref/edge.html

(a) Canny Edge Detection (b) Sobel Edge Detection

Figure 5: Edge Detection Parameters

Nevertheless we quickly realised that, to a certain extent, Canny edge detection was not what we wanted. By outputting *weaker* edges in the binary image, we were including information which was not necessary for the face detector. This had a direct, adverse effect on normalisation – particularly when taking the mean and standard deviation of patches, where an increased amount of edges had the result of muddling this process. Instead we used the Sobel edge detection method, which only returns *strong* edges in the binary image dependent on default or parameterised threshold (Figure 5).

2.0.3 Towards Edge Detection Optimisation

Optimising the senstivity threshold at which edges are recognised allowed us to fine-tine our detector and achieve better results. Potentially this does create somewhat of a light dependency on the source image, but arguably this is appropriate given that vision algorithms will ultimately target a particular domain, or general style, of input image.

Our initial edge detection results are included below:

Table 2: Edge Detection Results

		Detection %	
Edge Detection	**Threshold**	**35 Faces**	**50 Faces**
Canny	0.10	28.76	26.25
Canny	0.05	20.23	34.13
Sobel	0.10	42.58	45.40
Sobel	0.05	16.00	20.80

We adjusted the *numDetections* parameter in the *EvaluateDetections()* function and ran two sets of test. The first test ran with the default value included as part of the basic face detector – 50 faces. The second test ran with 35 faces, which is the number of faces in the image. We felt that this was justified given that there was a known quantity of faces to detect within the image. However, within practical scenarios this value would need to be dynamic and generalisable – particularly if there are not a known quantity of faces to detect.

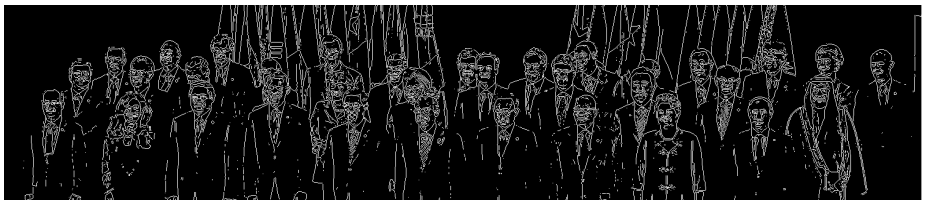

Figure 6: Edge Detection using the 'Sobel' method at 0.05 precision

Figure 6 shows the image after the *edge()* function has been applied, before it is passed to the normalisation function. In this example, the precision is low – we are detecting a great deal of edges in the input image. Furthermore, we are also detecting additional artifacts such as the flags in the background and the G20 leaders' body outlines. Our next strategy was to balance the threshold determining how many edges we detect, with the factor which the background flags and body outlines have in common – vertical edges.

We hypothesised that removing the vertical edges in the image would lead to a better chance of the remaining faces in the image being detected. Fortunately the *edge()* function contained an additional parameter, enabling us to specify the direction of the edges we wanted to detect – in our case *'horizontal'*. We therefore balanced this parameter with the threshold parameter. Although in our initial edge detection tests, the 0.1 precision had performed well, combining this with the 'horizontal' parameter led to a detrimental reduction in detected edges. In many scenarios, the circular edge shape of the head was lost. We found the threshold of 0.05 to be the best balance between detecting enough edges and removing non-associated face vertical edges.

Figure 7 shows the difference removing the vertical edges makes – body outlines and the background flags have largely disappeared, leaving just the faces plus some additional noise. This positive change is reflected in our results for the Sobel method at 0.05 precision: 52.92% for 50 face detections, and 46.45% for 35 face detections.

Figure 7: Edge Detection using the 'Sobel' method at 0.05 precision, with 'Horizontal' edges only

2.0.4 Further Image Processing

We took forward the most successful results and began several new approaches to strengthen our face detector. Inevitably edge detection led to unwanted smaller artifacts in the image being sharpened which were not necessary towards our goal. The logical next step was therefore to reduce this noise in order to enhance the facial structure which we were interested in.

Initially we used a basic Gaussian kernel to smooth the G20 image. This slightly improved the 50 face detections result (52.95% to 54.03%) and significantly improved the 35 face detections result (46.45% up to 56.26%). The Gaussian smoothing convolved the image in a similar manner to the mean filter during normalisation in Requirement 1, except the bell curve shape weighted the output pixels towards the central pixel value, rather than a uniform average.

Nevertheless, we found that using an averaging filter produced better results during our exploration. This is most likely due to the inclusion of being able to specify the filter size in the function call. For instance:

```
imf = filter2(fspecial('average', [2 2]), imf);
```

specifies a filter size of [2 2], where the output pixel will have the average value of the 2x2 region around a pixel in the input image. This was extremely successful in reducing noise resulting from edge detection – in fact, so much so that we applied this function twice. The resultant image is shown in Figure 8.

Figure 8: 2x 'Average' Kernel filtering, using a 2x2 neighbourhood

With the most consistently high detection rate occurring when the filter size was $\{2, 2\}$ across a detection rate of 35 and 50 faces (Figure 9), our final optimisation investigated the effects of changing suppression distance.

Our hypotheses was that there would be an optimum size of circular radius – the suppression distance – which is used to find local maxima within the image. For the maximum points within this radius, their borders would be eliminated, culminating in thinner edges which would ideally approach a small number of pixels thick. Superfluous pixels would be ignored, and the edges we were interested in would be sharper, leading to a greater detection rate percentage. Figure 10 shows that the optimum suppression distance was discovered at a radius of 17 with 35 faces.

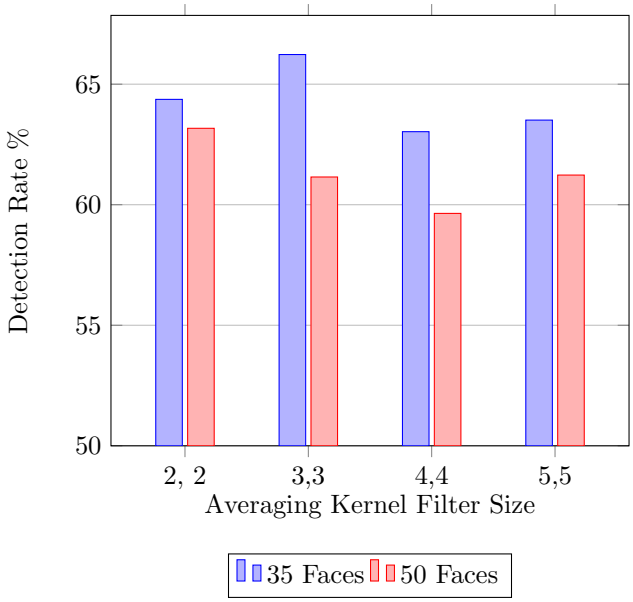

Figure 9: Variations in Kernel Size on Detection Rate

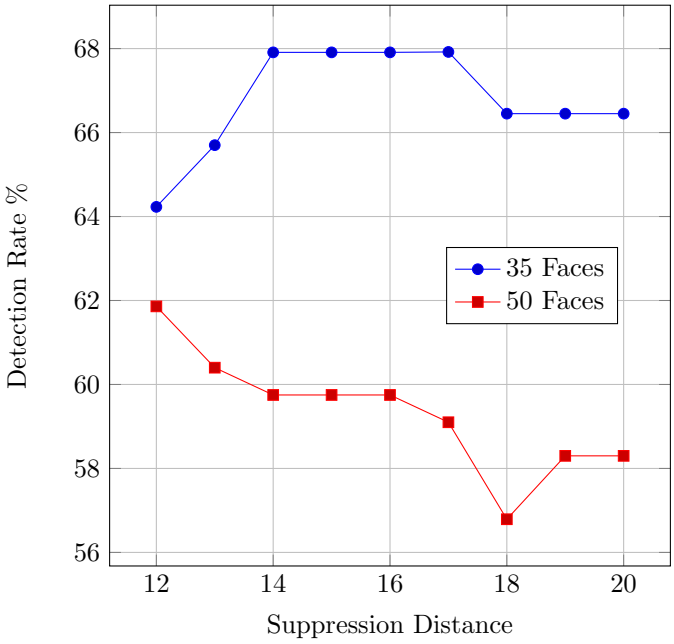

Figure 10: Variations in Suppression Distance on Detection Rate

Our final detector improved the detection rate from 21.43% in Requirement 1, up to 67.92% in Requirement 2. Our breadth-based exploration began by splitting the RGB image into its individual colour channels. This led us to investigate edge detection methods, specifically Canny and Sobel methods. Later we excluded the vertical edges, and applied Gaussian and Averaging kernels to remove noise from the image. Finally we considered the optimum suppression distance and the edge thinning effects this exhibits. This resulted in an improved face detector, with a 67.92% face detection rate shown in Figure 11.

Figure 11: Improved face detector with a 67.92% success rate

Possible extensions to this work include additional pre-processing methods. For instance, applying a Gaussian blur filter before and after edge detection, in order to remove noise at both stages of the pipeline. Additionally, the inclusion of low and high pass filters tuned to the domain of input images, enabling irrelevant frequency bands to be eliminated. This would filter only to a range of frequencies we were interested in. Finally we could use Haar-like features to detect faces using many weak classifiers in a cascading classifier setup. This would bring our detector closer to the successful Viola Jones approach of face detection.

3 Requirement #3: Nearest-Neighbour Classifier

We were provided with the class *FaceClassify.m* which defined the ground truth for the Viola Jones rectangles identifying specific faces in the G20 photo. Significantly this enabled us to use various classification methods and compare our results against the pre-determined expected outputs.

Our first step was to use the pre-determined Viola Jones rectangles to read and store the specific faces from the G20 photos. Listing 4 shows how we used the rectangles to extract appropriate faces from the image and store them in the *g20data* matrix. We defined additional variables {*workingImSz, nRGB*} which allowed us to fine-tune the classifier in Requirement 4 at a later stage for efficiency and success rate.

```
% Get G20 data to classify against
nRects = size(rects, 1);
g20data = zeros(nRects, (workingImSz * workingImSz * nRGB));

% Use VJ rectangles to get the G20 test faces
for i = 1:nRects
    imi = im(rects(i, 1):rects(i, 3), rects(i, 2):rects(i, 4), :);
    % Resize for efficiency
    imf = imresize(imi, [workingImSz workingImSz]);
    g20data(i, :) = imf(:);
end;
```

Listing 4: Reading G20 Faces

We now had the faces which we were required to classify with a particular name – for instance, *Obama* or *Erdogan* – stored in *g20data*. The second step was to gather the training data from the provided *facedata* image, containing 32 image examples of 20 people. We wrote a new function *ReadImageData* to allow us to collect the 640 images from the single *facedata* file and store them in a 20×32 matrix.

```

function [ allData ] = ReadImageData( dataIm, nPeople, nExamples,
    workingImSz)

    % Define image size in large image to read
    imSz = 64;
    allData = cell(nPeople, nExamples);

    for i=1:nPeople
        for j = 1:nExamples

            % Get the next image's location in the larger image
            x = imSz * i;
            y = imSz * j;
            img = dataIm(x-(imSz-1):x, y-(imSz-1):y, :);
            img = imresize(img, [workingImSz workingImSz]);
            img = mean(img, 3);

            % Add to our final matrix
            allData{i,j} = img;
        end
    end
```

Listing 5: Initial ReadImageData Function

Listing 5 shows the initial *ReadImageData* function used in Requirement 3 to iterate through the images (*Note:* We adjust this function during Requirement 4). A separate variable *imSz* is defined with a constant size of 64. The dimensions of the example images in *facedata* are read in at their source size 64×64 and later resized to the user-defined *workingImSz*. During requirement 3 we set this to 32, and set the main and face images to grayscale as specified by the requirement.

The third and final step was the nearest neighbour classification algorithm itself. Listing 6 shows how we used the two datasets we collected above to classify the G20 leaders. This involved iterating through the 640 images held in *validationData* and finding the euclidean distance between each image and a leader extracted via the Viola Jones rectangles. The best result was stored in the variable *closestMatch* and a label was returned after the 640-image dataset had been traversed. This label corresponded to the nearest neighbour for each leader – or where *closestMatch* was nearest to 0.

```
1
2       for i = 1:nOfTestFaces
3           closestMatch = -1; % We want this closest to 0
4           personIndex = -1;
5           g20face = g20(:, i, :);
6
7           for j = 1:nPeople
8               for k = 1:nExamples
9                   candidateImg = validationData{j,k};
10                  matchStrength = norm(candidateImg(:) - g20face);
11
12                  % Update closest match if closer to the G20 face
13                  % Or get initial match if it hasn't yet been set
14                  if matchStrength < closestMatch || closestMatch ==
    -1
15                      personIndex = j;
16                      closestMatch = matchStrength;
17                  end
18              end
19          end
20
21          % Assign the closest/nearest neighbour prediction
22          testPred(i) = personIndex;
23
24      end
```

Listing 6: Excerpt From NearestNeighbourClassification Function

Figure 12: Nearest Neighbour Classification Results

The vector *testPred* is used with the supplied *EvaluateClassification* function, which validates our results against the ground-truth for this image. The basic Nearest Neighbour Classifer with no optimisations returns a result of 4/18 faces correctly identified, as shown in Figure 12. The following section explores how we improved the basic classification rate of 4/18 to 8/18 using a variety of

techniques. Future directions for improving the classifier beyond this coursework are suggested.

4 Requirement #4: Improved Classifier

With a basic classification rate of 4/18 in place, we set out to improve and optimise our classifier. Our first approach was to use normalisation to standardise the images from *facedata* when we were reading them in. For an input image X:

$Y = \frac{X - \bar{X}}{\sigma}$

which translated to lines 17-19 of Listing 7.

```
1
2  function [ allData ] = ReadImageData( dataIm, nPeople, nExamples,
       workingImSz)
3
4      % Define image size in large image to read
5      imSz = 64;
6      allData = cell(nPeople, nExamples);
7
8      for i=1:nPeople
9          for j = 1:nExamples
10
11             % Get the next image's location in the larger image
12             x = imSz * i;
13             y = imSz * j;
14             img = dataIm(x-(imSz-1):x, y-(imSz-1):y, :);
15
16             % Normalise the image
17             img = img - mean(img(:));
18             img = img / std(img(:));
19
20             img = imresize(img, [workingImSz workingImSz]);
21
22             % Add to our final matrix
23             allData{i,j} = img;
24         end
25     end
```

Listing 7: Optimised ReadImageData Function

Initially we resized the image before normalisation, which yielded an improved return of 5/18. While this performed well in terms of speed, scaling the number of pixels down inevitably resulted in a loss of quality, having a negative effect on the normalisation process. Instead, we resized the image after normalising the original 64×64 image. At the dimensions and dataset sizes we were working at, this had a negligible effect on speed and culminated in an additional two faces being identified. With larger datasets, and larger image dimensions, normalising before resizing would have to be considered carefully in terms of the

performance-success ratio. Nevertheless, this increased our success rate to 7/18 (Figure 13).

Figure 13: Nearest Neighbour Classification with Normalisation

We extended our Face Classifier to use RGB images, rather than converting to grayscale initially. We adjusted this through the *grayscaleFilter* and *nRGB* configurable variables which we had set up during Requirement 3. Using just the red channel had no effect on the results, but using the green and blue channels reduced the success rate to 5/18 and 3/18 respectively – these channels do not lend themselves to highlighting facial features. Using all 3 channels identified an additional face, bringing our success rate to 8/18 as shown in Figure 14.

Figure 14: Nearest Neighbour Classification with RGB Normalisation

Future Work

We had planned to use a Support Vector Machine (SVM) to classify images on the basis of a hyperplane to separate classes (faces) within our G20 image. However, due to the overlap with final year project, we were not able to complete this in time. Nevertheless, we wrote the code to split the data from *facedata* into training and test sets. This can be toggled with the configurable variables at the top of *FaceClassify* in Listing 8.

```
% Config vars
NNClassification = 0;
SVMClassification = 1;
trainingSetSize = 25;
grayscaleFilter = 0;

...

```

```
if SVMClassification == 1
    % Decide sizes of test and training sets
    testSetSize = nExamples - trainingSetSize;

    % Now split this into separate arrays
    trainingSet = allData(1:trainingSetSize);
    testSet = allData(trainingSetSize+1:nExamples);
end
```

Listing 8: Splitting into Testing and Training Data

The purpose of splitting the images into two distinct sets is so that the training set can help to construct the SVM prediction algorithm. The test set can then be used on the SVM model to give a prediction of real-world performance, given that the images in this set are fresh and were not included as part of the training set. A third set – the validation set – is often used to tune particular parameters of the model, and indeed gives an early indication of how well the SVM algorithm performs given the training data.

SVM is relatively sensitive to new images in the datasets, and so the split of training and test data (e.g 60/40, 80/20) should be carefully considered and empirically tested. The SVM algorithm can be extended to a highly-dimensional space, which is suitable for the 18 faces to be classified here. The aim of the SVM algorithm is to identify a hyperplane which has the maximum separation between the training observations (the margins) across the higher-dimensional space. These observations can then be separated into classes according to the hyperplane, and each class provides a label for evaluation.

A separate approach could have been explored altogether. Using a classification algorithm such as Naive Bayes may have enabled a smaller training set to be allocated[2], given appropriate feature selection. This would be advantageous in terms of performance generalisability – shorter training times for the classification model would be required. Furthermore, an eigenvector-based approach could have been investigated for feature extraction.

Smaller experiments could have also been performed on the existing nearest neighbour algorithm. For instance:

```
matchStrength = norm(candidateImg(:) - g20face);
```

returns the euclidean distance between the example face from *faceData* and the G20 face. This was the only form of distance which we considered. Experimenting with different styles of distance between the two images may have provided some varied results – for instance, match strength based on the Manhattan distance and the Mahalanobis distance.

Additionally, we could have concentrated on the assignment problem – that is, uniquely assigning a label to each G20 face. In our algorithm, labels are not

[2]Wasikowski, M., & Chen, X. W. (2010). Combating the small sample class imbalance problem using feature selection. IEEE Transactions on knowledge and data engineering, 22(10), 1388-1400.

unique and can be assigned multiple times to leaders in the G20 image. By generating an array of {*label, matchStrength*} for each G20 face, an implementation of the *Hungarian Algorithm*[3] could have been developed to match the strongest, unique label to each person in the G20 image.

Conclusion

This report has detailed the design and development of a face detector and face classifier. Requirement 1 described the sliding normalisation process we used to achieve a detection rate of 21.43%. In Requirement 2 we took a breadth-based approach to explore edge detection, averaging kernels, and edge thinning through an optimal suppression distance. We achieved a 67.92% success rate for our improved face detector.

The second half of this report focused on the design and development of a face classifier. Our basic Nearest Neighbour algorithm correctly classified 4/18 faces, and our improvements in Requirement 4 brought this to 8/18. Finally, we discussed how we would extend this project through SVM classification. Nearest Neighbour distance experiments and an implementation of the Hungarian Algorithm to tackle the assignment problem were pinpointed as directions for future research.

[3]Kuhn, H.W., 2010. The hungarian method for the assignment problem. 50 Years of Integer Programming 1958-2008, pp.29-47.