Assignment 3 - Image Analysis

Theodora Gaiceanu

1 Your own classifier

The selected classifier is KNN, and I chose to implement one nearest neighbour, as we only need one output value. The classification data is just a matrix constructed by concatenating the X features and the Y outputs. The concatenation was done in such a manner that the matrix has the features on the first rows and the class (-1 or 1) on the last row. Therefore, the training function has the following code.

```
function classification_data = class_train(X, Y)
    classification_data = [X; Y];
end
```

Listing 1: training function

For KNN, the basic idea is to compute the Euclidean distance for each x from the classification data set and each x from the training data set, respectively testing data set. Then the distances are summed. For every column in the classification data, we store the sum of distances. Afterwards, the array with the sum of distances is sorted, and the index of the smallest value is extracted. Lastly, the corresponding class of the index is stored in the output variable. The code for the KNN algorithm can be observed below.

```
function y = classify(x, classification_data)
    % 1 KNN implementation
    [m,n] = size(classification_data);
    [mx,nx] = size(x);
    dist_sum = ones(1,n);
    %for each column in the classification data
    %and each column in x
    %compute the euclidian distance between the x values
    for i = 1:n
        sum = 0;
        for j = 1:mx
            %euclidian distance
            dist = sqrt((x(j,1)-classification_data(j,i))^2);
            %sum the distances
            sum = sum + dist;
        end
        %construct the array with the sum of distances
        %it will have the same number of columns as the
    classification data
        dist_sum(1,i) = sum;
    end
    %sort the array with the sum of distances
    [min_dist,index] = sort(dist_sum,'ascend');
    %use KNN wth only one neighbour
    k = 1;
    %extract the index of the mnmal value
    index = index(1:k);
    %select the corresponding result for the index
    y_coresp = classification_data(m,index);
    y= y_coresp;
end
```

Listing 2: classify function

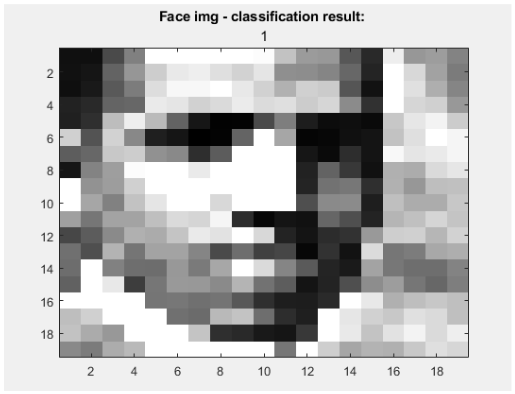
The error rates can be observed below. The algorithm has a testing accuracy of 87.80% and a training accuracy of 100.00%. Therefore, the model overfits, but that was expected, as it is known that KNN with 1 neighbour tends to overfit.

```
mean_err_rate_test =

    0.1220

mean_err_rate_train =

    0.0000
```

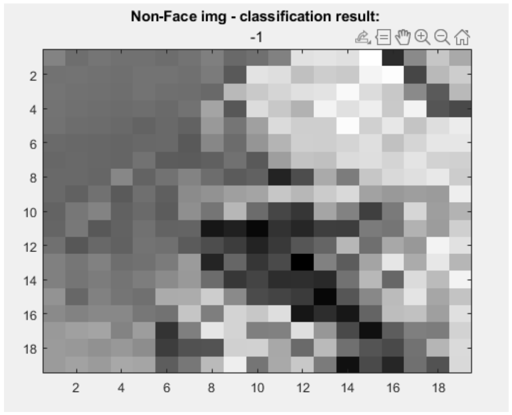
Listing 3: Error rates

The results of the algorithm can be observed visually also. Two images from the dataset were selected, one showing a face, and the other showing a non-face. As it can be noticed in the text of the image, the model managed to predict correctly the output.

Also, the code for displaying the images and the results can be observed below.



(a) Face image



(b) Non-face image

Figure 1: Results when testing the model on two images

```
for i = 1:length(Y_test)
    if(Y_test(:,i)==1)
        figure;
        colormap(gray);
        imagesc(X_reshape(:,:,1,i));
        res = classify(X_test(:, i), classification_data);
        title('Face img - classification result: ',res);
        break;
    end
end
for i = 1:length(Y_test)
    if (Y_test(:,i)==-1)
        figure;
        colormap(gray);
        imagesc(X_reshape(:,:,1,i));
        res = classify(X_test(:, i), classification_data);
        title('Non-Face img - classification result: ',res);
        break;
    end
end
```

Listing 4: Code for the images

2 Use pre-coded machine learning techniques

```
mean_err_rate_test =

    0.1220    0.1578    0.0510    0.1465


mean_err_rate_train =

         0    0.0099         0         0
```

Listing 5: Error rates

The errors correspond from left to right to the following models: the KNN from task 1, the Regression Tree Classifier, the SVM, and the built-in KNN. As it can be observed, the mean error rates for the Regression Tree Classifier and Nearest Neighbours Classifier are quite similar for the testing data set. But both have very good performances on the training data. This means that the models are a little overfitted. This is expected in a way, because Regression Trees usually overfit if there is no constraint for the depth of the tree. Also, Nearest Neighbour Classifier with small k tends to overfit (k is 1 in this case). The best accuracy is achieved when using SVM. SVM uses only a part of the data set to make the prediction (the support vectors).
Moreover, when comparing the mean error rates for the built-in KNN model and the KNN model from the last exercise, it can be observed that they have pretty similar performances. This was expected also, as they both implement the same algorithm.

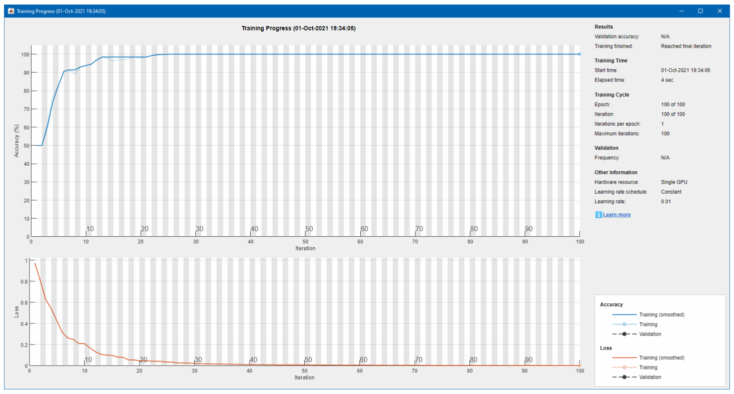
3 Testing a simple CNN model

```
mean_err_rate_test =

    0.0512


mean_err_rate_train =

     0
```

Listing 6: Error rates

As it can be noticed from the mean error rate for testing (100 trials), the accuracy of the CNN model is very high also for the testing data set. Therefore, the model is not overfitted. This is a substantial improvement from the cases where Regression Tree and Nearest Neighbours Classifiers (both KNN models) were used. But the performance of the CNN model is very similar to the SVM model.

Figure 2 illustrates the training process when using only 1 trial.



Results

Validation accuracy:	N/A
Training finished:	Reached final iteration

Training Time

Start time:	01-Oct-2021 19:34:05
Elapsed time:	4 sec

Training Cycle

Epoch:	100 of 100
Iteration:	100 of 100
Iterations per epoch:	1
Maximum iterations:	100

Validation

Frequency:	N/A

Other Information

Hardware resource:	Single GPU
Learning rate schedule:	Constant
Learning rate:	0.01

Learn more

Figure 2: Training process

4 Line fit

Both Least Squares and Total Least Squares are line fitting methods. But when using the Least Squares model, it is assumed that the error is only in the y-direction. Also, the line cannot be vertical because this would lead to big values of the error. Therefore, sometimes one may not obtain the best fitting line using a Least Squares model. The Total Lest Squares method aims to reduce these disadvantages, by minimising the distance between the point and the line. The solution for Total Least Squares is obtained by solving an eigenvalue problem and by checking two orthogonal solutions [1].

4.1 The Least Squares approach

The implementation for the Least Squares method is simple. First, the A matrix needs to be formed. It contains the x coordinates in the first column and values of 1 in the second column. Then, the slash function is used.

```
A = ones(length(x),2);
A(:,1) = x;
```

```
p_ls = A\y
```

Listing 7: Least Squares code

4.2 The Total Least Squares approach

For Total Least Squares, one must first compute the A matrix using the equations in [1]. After that, it is known that finding the solution for the Total Lest Squares is actually an eigenvalue problem. Consequently, the eigenvalues of A were computed using the Matlab *eig* function. This gave two sets of a, b, c values. The best line is the line that has the minimal value for the sum of squares of the distance [1] from equation (1).

$$\sum_i (ax_i + by_i + c)^2 \tag{1}$$

After comparing the results from both sums, the first one was chosen as it had the smallest value. Next, the line was put in the form $y = kx + m$.

```
sum_x = 0;
sum_x_sqrd =0;
sum_y = 0;
sum_xy = 0;
sum_y_sqrd = 0;
for i = 1:length(x)
    sum_x_sqrd  = sum_x_sqrd + x(i)*x(i);
    sum_x = sum_x + x(i);
    sum_xy = sum_xy + x(i)*y(i);
    sum_y = sum_y + y(i);
    sum_y_sqrd = sum_y_sqrd +y(i)*y(i);
end
A = [sum_x_sqrd - sum_x*sum_x/length(x), sum_xy - sum_x*sum_y
   /length(x); sum_xy - sum_x*sum_y/length(x) sum_y_sqrd-
   sum_y*sum_y/length(x)]
[eig_val,eig_array] = eig(A)
a1 = eig_val(1,1)
b1 = eig_val(1,2)
c1 = -(a1*sum_x+b1*sum_y)/length(x)

a2 = eig_val(2,1)
b2 = eig_val(2,2)
c2 = -(a2*sum_x+b2*sum_y)/length(x)

sum_1 = sum((a1*x+b1*y+c1).^2)
```

```
sum_2 = sum((a2*x+b2*y+c2).^2)

k_tls = -a1/b1;
m_tls = -c1/b1;
p_tls = [k_tls; m_tls];
```

Listing 8: Total Least Squares code

As it can be noticed from Figure 3, the Total Lest Squares line approximate points the best.

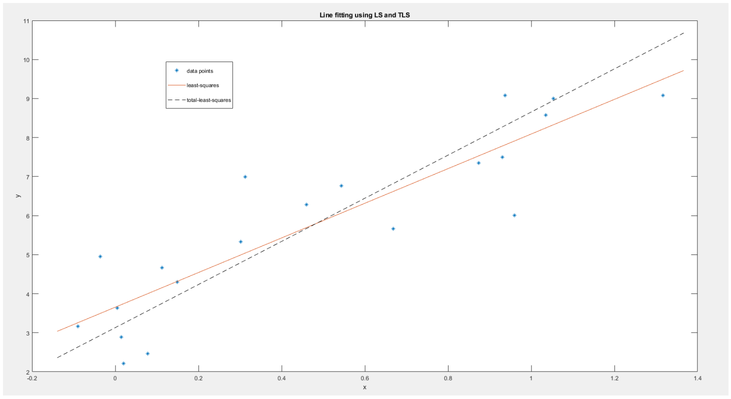


Figure 3: Line fitting

The errors for the two lines can be computed using the equations (2) and (3):

$$\frac{|ax + by + c|}{\sqrt{a^2 + b^2}} \tag{2}$$

$$\sqrt{|y - y_est|^2} \tag{3}$$

Therefore, the equations were implemented for each line.

```
%%LS line
y_est_ls = p_ls(1) * x + p_ls(2);
error_dif = (y - y_est_ls);
err_ls = norm(error_dif,2).^2

d_ls = abs(p_ls(1)*x + (-1)*y + p_ls(2))./sqrt(p_ls(1)*p_ls
    (1)+1);
err_ls_2 = norm(d_ls,2).^2

```

```
%TLS line
y_est_tls = p_tls(1) * x + p_tls(2);
error_dif_tls = (y - y_est_tls);
err_tls = norm(error_dif_tls,2).^2

d = abs(a1*x+ b1*y + c1)./sqrt(a1*a1+b1*b1);
err_tls_2 = norm(d,2).^2

%%all results
Res = [err_ls err_ls_2; err_tls err_tls_2]
```

Listing 9: Errors for the two lines - code

The results can be observed below:

```
Res =

   19.5072    0.9433
   24.1030    0.7649
```

Listing 10: Errors for the two lines - results

As it can be observed, the LS error for the Least Squares line is smaller than the LS error for the Total Least Squares Line (19.5072 vs 24.1030), but the TLS error for the Least Squares line is bigger than the TLS error for the Total Least Squares line (0.9433 vs 0.7649). Summarising, the best LS error is achieved when using the Least Squares line, and the best TLS error is achieved when using Total Least Squares line. Anyway, as it can be notice from both the Figure 3 and the errors, the difference is not so big.

Completion after feedback The fact that the best LS error was achieved by the Least Squares line is normal. This is because the Least Squares line was actually computed by solving a LS problem. This means that the optimal parameters were found by minimising the squared vertical residuals [2]. So, the Least Squares line already had the optimal parameters for the LS error. Similarly, it is normal that the best TLS error was achieved by the Total Least Squares line. This is due to the fact that the Total Lest Squares line was computed by solving a TLS problem. The optimal parameters were found by minimising a set of homogeneous squared errors, in all directions [2]. Therefore, the Total Least Squares line had before the optimal parameters for the TLS error.

5 OCR system construction and system testing

For this exercise, the classifier used is the KNN model from task 1. The hitrate obtained for the OCR-system on short1 dataset is shown below.

```
>> inl3_test_and_benchmark
Hitrate = 50%
```

Listing 11: Results - short1 directory

The hitrate obtained for the OCR-system on home1 dataset can be noticed below.

```
>> inl3_test_and_benchmark
Hitrate = 23.7%
```

Listing 12: Results - home1 directory

As it can be noticed, the hitrate for the home1 directory is significantly lower than the one obtained for the short1 directory. This is explainable, because the examples from home1 directory have more noise than the ones from short1 directory. So the segmentation part should be further improved.

Completion after feedback But I also improved the *segment2features* function in order to get the above performances. First of all, I normalised the Area and the Perimeter by dividing them by $m*n$, where m, n are the size of the image. Also, I normalised the Centroid, by dividing the centroid components by n, respectively m. I did the normalisation because I wanted to have the majority of the features' values between 0 and 1. Moreover, I thought that adding the *Eccentricity* to the feature vector may be a good idea, as it could be very helpful for differentiating between line segments (eccentricity=1) and circle segments (eccentricity 0). The eccentricity is defined as the ratio of the distance between the focus points of the ellipse and its major axis length [3]. In addition, I have used *MajorAxisLength* and *MinorAxisLength* as it could offer a perspective over the size of a digit. Major, respectively Minor Axis Length is a scalar that represents the length (in pixels) of the major/ minor axis of the ellipse having the same normalized second central moments as the region [3]. Also, I have computed the number of holes as the number of objects from the connected components in the complemented image - 1.

Lastly but not least, I noticed that some digits from the home1 directory had separated segments. Therefore, in this case, the number of connected components per digit would have been bigger than 1. So I put a condition, whenever the number of the connected components per digit is larger than 1, a closing operation is applied, so that the digit is continuous. Then the number of connected components is computed again using the closed image.

The rest of the features were the same as in Assignment 2. Some properties of the image regions given by the Matlab function *regionprops* have been taken into account. The digits have different sizes and different number of white pixels. Therefore, it seemed a good idea to include the area and the perimeter in the feature vector.

Also, some digits have holes inside (e.g. 0, 6, 8, 9). In this context, the Euler number might be a good property, as it is defined as the difference between the total number of objects in the image and the total number of holes in the objects [2]. Therefore, negative and 0 values of the Euler Number should be expected for the digits with holes.

The centroid may also be useful, as it is the average of the pixels forming a shape. So, depending on the shape of a digit, the centroid has different coordinates.

In addition, the Matlab function *imdilate* does a dilation operation on the image. In this way, a bigger digit should be obtained, with filled holes. Consequently, another way to differentiate between the digits with holes and the digits without holes may be to compute the difference between the area of the dilated digit and the area with the original digit. In this way, the difference would be smaller for the digits without holes and bigger for the digit with holes.

Lastly, the number of edges and corners may be slightly different from one digit to another. The edges of an image can be obtained easily, by using the Matlab function *edge*. Then, these can be counted by using the function *nnz*, which counts the number of non-zero elements of a matrix. As far as the corners are concerned, the corner points can be obtained by using the Matlab function *detectHarrisFeatures* and the number of corners is the length of the array with the corner points.

The new code for *segment2features* can be observed below.

```matlab
function features = segment2features(I)
    CC =  bwconncomp(I,8);
    if CC.NumObjects >1
        se = strel('disk',10);
        I = imclose(I,se);
        CC =  bwconncomp(I,8);
    end
    [m,n]=size(I);
    holes = bwconncomp(~I);
    holes_no = holes.NumObjects -1;
    A = regionprops(CC,'Area');
    P = regionprops(CC,'Perimeter');
    EN = regionprops(CC,'EulerNumber');
    C = regionprops(CC, 'Centroid');
    E = regionprops(CC, 'Eccentricity');
    edges = edge(I,'canny');
    countEdges = nnz(edges); %number of nonzero matrix
   elements
    se = strel('sphere',5);
    I_filled = imdilate(I, se);
    [m2,n2] = size(I_filled);
    CC_filled =  bwconncomp(I_filled,8);
    A_filled = regionprops(CC_filled,'Area');
    diff_filled = A_filled.Area/(m2*n2) - A.Area/(m*n);
    corneres = detectHarrisFeatures(I);
    noCorners = length(corneres);
    MajA = regionprops(CC,'MajorAxisLength');
    MinA = regionprops(CC,'MinorAxisLength');
    features = [A.Area/(m*n) P.Perimeter/(m*n) EN.EulerNumber
    C.Centroid(1)/n C.Centroid(2)/m diff_filled countEdges
   noCorners E.Eccentricity MajA.MajorAxisLength MinA.
   MinorAxisLength holes_no];
end
```

Listing 13: New code for extracting features

References

[1] M. Oskarsson, Image Analysis Course (2021), Lund University.
[2] R. Szelisky, Computer Vision: Algorithms and Applications (2010), Springer.
[3] Mathworks, https://www.mathworks.com/help/images/ref/regionprops.html