Assignment 3

Maks Epsteins

October 18, 2021

Exercise 1

1.1 Code for the Classify Function (Gaussian Bayes)

```
function y = classify(x, classification_data)

    nrfeatures = size(x,1);
    nrsamples = size(x,2);
    y = zeros(1,nrsamples);

    for s = 1:nrsamples
        y1_total = 0;
        y2_total = 0;

        for f = 1:nrfeatures

            mean1f = classification_data.class1(1,f);
            stdv1f = classification_data.class1(2,f);

            mean2f = classification_data.class2(1,f);
            stdv2f = classification_data.class2(2,f);

            y1 = normpdf(x(f,s), mean1f, stdv1f);
            y2 = normpdf(x(f,s), mean2f, stdv2f);

            y1_total = y1_total + log(y1);
            y2_total = y2_total + log(y2);

        end

        score1 = y1_total + log(classification_data.class1(:,nrfeatures + 1));
        score2 = y2_total + log(classification_data.class2(:,nrfeatures + 1));

        if score1 >= score2
            y(s) = -1;
        elseif score2 > score1
```

```
            y(s) = 1;
        end

    end

end
```

1.2 Code for the Class Train Function

```
function classification_data = class_train(X, Y)

    nrfeatures = size(X,1);
    nrsamples = size(Y,2);

    data = struct('class1', zeros(2,nrfeatures),'class2', zeros(2,nrfeatures));

    class1Y = [find(Y(1,:)==-1)];
    class1X = X(:,class1Y);

    class2Y = [find(Y(1,:)==1)];
    class2X = X(:,class2Y);

    for f = 1:nrfeatures
        param1 = class1X(f,:);
        param2 = class2X(f,:);
        data.class1(:,f) = transpose([mean(param1), std(param1)]);
        data.class2(:,f) = transpose([mean(param2), std(param2)]);
    end


    apriori1 = size(class1X,2)/nrsamples;
    apriori2 = size(class2X,2)/nrsamples;

    data.class1(:,nrfeatures+1) = apriori1;
    data.class2(:,nrfeatures+1) = apriori2;

    classification_data = data;

end
```

1.3 Mean Error Rates for the Data

```
mean_err_rate_test =

    0.1698         0         0         0


mean_err_rate_train =

    0.1289         0         0         0
```

Figure 1: Error rates for task 1

The error rate seems pretty adequate for both the test and the training data.

1.4 Image of face and nonface

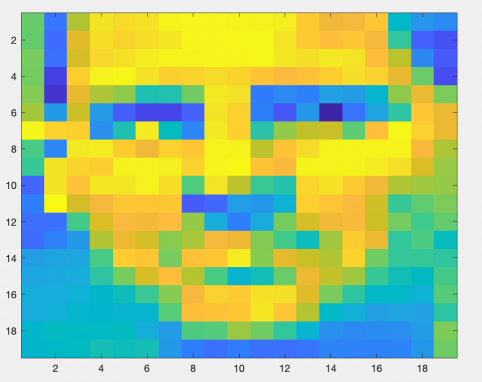

Figure 2: Example image of a face which was classified correctly.

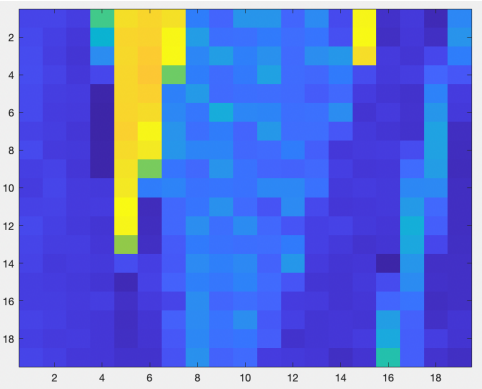

Figure 3: Example image of a non-face which was also classified correctly

Exercise 2

2.1 Error Rates

```
mean_err_rate_test =

    0.1725    0.1507    0.0430    0.1517


mean_err_rate_train =

    0.1259    0.0114         0         0
```

Figure 4: Error rates for task 2

As expected the SVM classifier gave the best results for both the test and training data since it is the more advanced of the algorithms and also works well for more complex data sets.

It also makes sense that the error rate for the training data for the NN classifier is so low since the training data for NN will never have any error, but it also makes sense that the error rate for the test set is quite high since NN is one of the more "primitive" classifier algorithms and the training data seems to be quite chaotic, which again explains why the SVM performed so much better than the other classifiers.

Our Gaussian Bayes classifier did not work so well, in fact it performed the worse but also makes sense since GB is more sensitive to large variance in data, but I'm suprised that it still performed worse than NN since NN should be more sensitive. However it could have been by sheer chance that NN worked better due to the inconsistency of that algorithm.

The tree classifier results I believe are very close to NN since they are kind of similar algorithms.

Exercise 3

```
mean_err_rate_test =

    0.0473         0         0         0


mean_err_rate_train =

     0     0     0     0
```

Figure 5: Error rates

The error rates for the CNN is similar to the error rate for the SVM classifier. This I assume is due to the fact that same as with SVM CNN is better suited for complex data sets than the other classifiers. I would expect the CNN however to have a better result than SVM due to the large amount of features for our data.

Exercise 4

4.1 Least Squares

4.1.1 Code for Least Squares

```
function leastsquares = ls(x,y)

    npoints = max(size(x));
    xsquared = x;

    for i = 1:npoints
        xsquared(i) = x(i)^2;
    end

    xy = x;

    for i = 1:size(x)
        xy(i) = x(i)*y(i);
    end

    sumx = sum(x);
    sumy = sum(y);
    sumxsquared = sum(xsquared);
    sumxy = sum(xy);

    k = (npoints*sumxy - sumx*sumy)/(npoints*sumxsquared - sumx^2);
    m = (sumy - k*sumx)/npoints;

    leastsquares = [k,m];

end
```

4.1.2 Code for Least Squares Error

```
function error = lserror(k,m,x,y)

    npoints = max(size(x));
    error = 0;

    for i = 1:npoints
        error = error + (y(i) - (k*x(i) + m))^2;
    end
end
```

4.2 Total Least Squares

4.2.1 Code for Total Least Squares

```
function totalleastsquares = tls(x,y)
%TLS Summary of this function goes here
%   Detailed explanation goes here

    npoints = max(size(x));

    xbar = mean(x);
    ybar = mean(y);

    w = 0;

    for i = 1:npoints
        w = w + (y(i)-ybar)^2 - (x(i) - xbar)^2;
    end

    r = 0;

    for i = 1:npoints
        r = r + 2*(x(i) - xbar)*(y(i) - ybar);
    end

    k = (w + sqrt(w^2 + r^2))/r;

    m = ybar - k*xbar;


    totalleastsquares = [k,m];
end
```

4.2.2 Code for Total Least Square Error

```
function error = tlserror(k,m,x,y)
    npoints = max(size(x));
    error = 0;

    for i = 1:npoints
        error = error + (abs(-k*x(i) + y(i) - m)/sqrt((-k)^2 + 1))^2;
    end

end
```

4.3 TLS and LS Regressions Plots

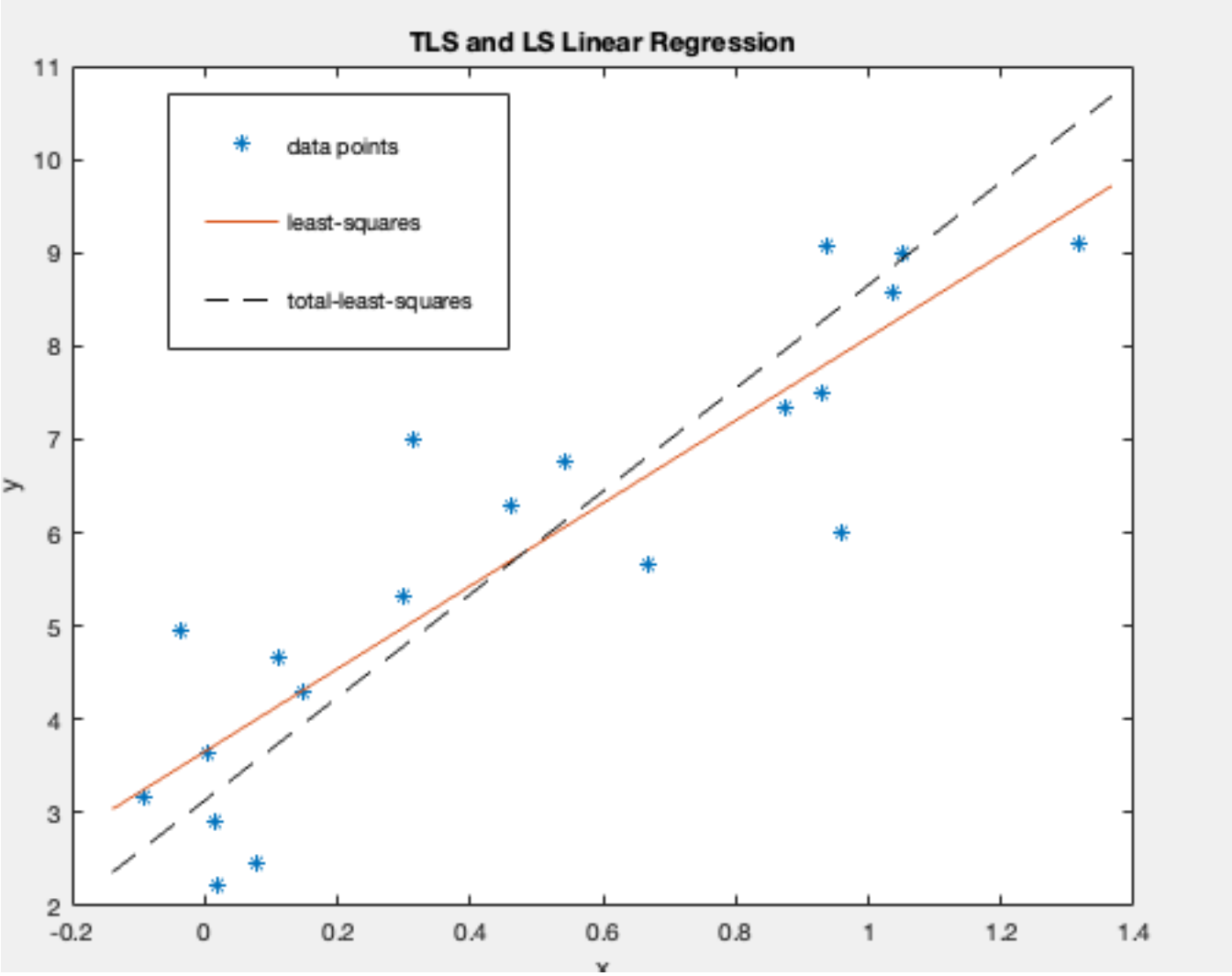

Figure 6: Plot for the TLS and LS Regression Lines

4.4 Results for TLS and LS

	LS	TLS
LS Error	19.5072	24.1030
TLS Error	0.9433	0.7649

The errors I believe are quite consistent with reality and fully make sense. The LS error will be smaller for LS than for TLS since we are the LS regression line minimizes the vertical distance, which the TLS regression line does not. And of course by the same logic the TLS error is better for the TLS regression than for LS. The reason why the TLS error is so much smaller for both LS and TLS is because when we calculate the TLS error we add the sum of the squares of the euclidean shortest distance between the regression line and our point. This value will always be less than or equal to the vertical distance regardless if its an LS or TLS regression.

4.5 Difference Between the Two Methods

Both of the methods are used to calculate the approximate solutions for overdetermined systems, in our case to approximate a linear regression. The difference between the two methods is that LS attempts to minimize the vertical distance between the line and each datapoint, while TLS attempts to minimize the euclidean distance.

Exercise 5

5.1 Code and Explanation for class_train Function

The Training algorithm loops through each sample in our data set which corresponds to each class and for said class generates a Gaussian distribution which represents each feature.

```
%The code was modified from the first exercise to accommodate for any number of
%classes

function classification_data = class_train(X, Y)
nrfeatures = size(X,1);
nrsamples = size(Y,2);
nrclasses = 10;

data = struct();

%Iterate through all the classes
for c=1:nrclasses
    %Find all the data in our dataset that has class c
    classY = [find(Y(1,:) == c)];
    index = strcat('c', num2str(c));

    %Initialize data struct with empty array with space for mean and deviation
    %for each feature and for each class.
    data.(index) = zeros(2, nrfeatures);

    %Iterate through each feature.
    for f = 1:nrfeatures
        %Get each data point for the current feature
        classX = X(:,classY);
        parameters = classX(f,:);
        %Add the mean and deviation to our data array
        data.(index)(:,f) = transpose([mean(parameters), std(parameters)]);
    end

    %Calculate our apriori and add it to our data set.
    apriori = size(classY,2)/nrsamples;
    data.(index)(:,nrfeatures+1) = apriori;
end

classification_data = data;
end
```

5.2 Code and Explanation for features2class Function using Gaussian Bayes

Loops through each sample and generates scores which represent the probability that our sample is within one of our 10 classes. This is done by iterating through the features of each

sample and taking the sum of the logarithm of the likelihood that said feature corresponds to said class. The class that has the highest sum will be the class that the sample is assigned to.

```
function y = features2class(x,classification_data)

nrfeatures = size(x,1);
nrsamples = size(x,2);
y = zeros(1, nrsamples);
nrclasses = 10;

%Iterate through each sample in the data set
for s = 1:nrsamples

    %Initialize the scores for each class to 0
    scores = zeros(10,1);

    %Iterate through each class (1-10)
    for c=1:nrclasses
        prob_total = 0;
        %The data is stored in a struct where the keys are written as 'cn'
        %where n is the number to which the class corresponds
        index = strcat('c', num2str(c));

        %Iterate through each feature
        for f = 1:nrfeatures
            %Get the calculated mean and standard deviation for each class
            mean = classification_data.(index)(1,f);
            stdv = classification_data.(index)(2,f);

            %If the standard deviation is 0 we only need to provide the mean to
            %normpdf
            if stdv ~= 0
                prob = normpdf(x(f,s), mean, stdv);
            elseif stdv == 0
                prob = normpdf(x(f,s), mean);
            end
            %Zero check to prevent error if the probability is 0 since we
            %are using the log function.
            if prob ~= 0
               prob_total = prob_total + log(prob);
            end

        end

        %We assign the calculated score into our score array and add the apriori
        %probability
        scores(c,1) = prob_total + log(classification_data.(index)(1,nrfeatures+1));
    end
```

```
    %The index variable will correspond to the class with the highest probability
    [~, index] = max(scores);
    y(1,s) = index;
end
end
```

5.3 Classification Results

```
>> inl3_test_and_benchmark
Hitrate = 50%
>> inl3_test_and_benchmark
Hitrate = 37.5%
```

Figure 7: Hitrates for short1 and home1 datasets

5.4 Made modifications

The majority of the modifications that were made were in the segment2features function which I pretty much redid most of. Also had to change the threshold in the im2segment function which made the segmentation worse, but kept more of the features (the holes in the numbers were particularly sensitive).

The features that were added were the following:

```
%Nbr of holes in the image, 1 is subtracted to not count the background.
%Works well to separate our numbers into at least 3
%classes based on how many holes the numbers has, for example
%8 has 2 while 0 has 1 and 7 has 0.
nbrholes = max(max(bwlabel(not(number)))) - 1;

%Density of white pixels in the image.
%The less "feature dense" numbers will have less
%pixels such as 1 while 8 which has "more features" will
%have a higher value.

density = sum(sum(number)) / numel(number);

%The proportion of white pixels between the top and bottom regions
%of our images. For example 6 will have a higher lowerhalfdensity
%value and therefore the proportional density will be negative,
%while 2 has more pixels in the top region and will therefore
%have a positive value.
upperhalf = number(1:y_mid,:);
lowerhalf = number(y_mid:end,:);

upperhalfdensity = sum(sum(upperhalf)) / numel(upperhalf);
lowerhalfdensity = sum(sum(lowerhalf)) / numel(lowerhalf);
```

```
proportionaldensity = upperhalfdensity - lowerhalfdensity;

%Density of white pixels in the middle 5 columns.
%I believe that this works well because the middle of the
%numbers is where a lot of the distinct features
%appear. For example 3,8 will have similar results while
%perhaps 7,1 will have similar depending on how they are written.
midcolumndensity = number(:,x_mid-2:x_mid+2);
midcolumndensity = sum(sum(midcolumndensity))/numel(midcolumndensity);

%Density of white pixels in the middle 5 rows.
%same reasoning as above, could for example help for
%in differentiating between numbers 3,8 which the above
%will give similar results for.
midrowdensity = number(y_mid-2:y_mid+2,:);
midrowdensity = sum(sum(midrowdensity))/numel(midrowdensity);

%Feature extraction kernel in order to find horizontal features in the image.
%Calculates the density of horizontal features in the image.
%Works well to separate numbers with a lot of horizontal
%features for example 2 from numbers such as 7,1.
hozkernel = 1/15*[0,0,0,0,0;
                  1,1,1,1,1;
                  1,1,1,1,1;
                  1,1,1,1,1;
                  0,0,0,0,0];

hozfeatures = conv2(number,hozkernel,'same');
hozfeatures = sum(sum(hozfeatures))/sum(sum(number));

%Feature extraction kernel in order to find vertical features in the image.
%Again same reasoning as above, just for vertical features
%instead of horizontal.

vertkernel = 1/15*[0,1,1,1,0;
                   0,1,1,1,0;
                   0,1,1,1,0;
                   0,1,1,1,0;
                   0,1,1,1,0];

vertfeatures = conv2(number,vertkernel,'same');
vertfeatures = sum(sum(vertfeatures))/sum(sum(number));
```