Handwriting Recognition by Linear Classification

Max Ptasiewicz

February 23, 2022

1 Linear Classification

Linear classification is a technique used to determine differences between large sets of data, and can be used to predict large sets of unknown data. For this project, linear classification, or more specifically binary classification was used to identify the handwritten digits 0 through 9, as shown in figure 1. The first part of this project is to understand the necessary equations to use to identify mystery data. This is based on the following:

$$\begin{array}{cc} a^T X_{not0,1} + b = -1 & a^T X_{0,1} + b = 1 \\ a^T X_{not0,2} + b = -1 & a^T X_{0,2} + b = 1 \\ \vdots & \vdots \\ a^T X_{not0,n} + b = -1 & a^T X_{0,n} + b = 1 \end{array}$$

a^T represents the hyperplane that in this case, will separate a number from appearing as a 0 or not a 0. X represents any image, that has been converted from the matrix of pixel values into a vector containing the same data. b represents the constant coefficient of the hyperplane. These equations work by trying to have the hyperplane multiplication operation come out to a constant, either 1 or -1, where a resulting 1 says that the image in question is a 0, and a resulting -1 says that the image in question is not a 0. This operation would be repeated with hyperplanes representing digits 0 through 9. The problem can be reformatted into the following vector equation that can be applied to each image:

$$\begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + b = 1$$

This vector equation is assuming that the x vector is representing the same digit that the hyperplane represents. The resulting 1 will not be exactly 1, as this will be formatted as a least squares problem, and no single hyperplane will make every correct image equal to 1, and every incorrect image equal to -1. The problem is that the variables in this problem is not the x vector, but rather both the hyperplane and the coefficient are the variables. In its current state, the vector equation cannot be formed into a least squares problem, as there are variables in several different places, but this will be remedied in the next section.

2 Least Squares Formulation

In order to formulate this problem as a least squares problem, or better said as 10 least squares problems, it is necessary to combine the variables into one vector:

$$\begin{bmatrix} a_1 & a_2 & \cdots & a_n & b \end{bmatrix}$$

In the final formulation, the vector will be a column vector of dimensions 677 by 1. This is because each image comes in as a 28 by 28 matrix, but to reduce the size of each matrix and make the code more efficient, the less used pixels around the edge have been removed, as shown in figure 2. The resulting 26 by 26 matrix is again vectorized and has a total of 676 values. The last value is

then the coefficient b. With the previous reformulation of the problem, we can create the following matrix equation:

$$\begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} & 1 \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ b \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = 0$$

Each row of the A matrix represents an image in the set of training data, followed by a 1 appended to the end of it. This is so each image can be multiplied using an inner product with the hyperplane, and the constant can be added to the end of the sum. The B vector, which in this case is full of 1s, will actually only have 1s where an image is known to be a certain number, such as 0. Everything else in that vector will be a -1. Depending on which digit the hyperplane is being determined for, the location of the 1s changes to correspond to the images containing that digit. This B vector is updated for each hyperplane solution, by changing the desired image type's corresponding -1 to a 1. Then the process is simply to repeat this with each digit to produce 10 hyperplanes.

3 Doing the problem

3.1 Preprocessing

The data is imported as a matrix of 60000 training images containing 784 pixels, in a 60000 by 784 matrix. Each image also has a label showing which digit the image represents. A set of 10000 testing images is also imported with labels as well, in the same format. The preprocessing that needs to happen is removing the outside edge of each image, as those pixels are useless and will just slow down the algorithm. The removed pixels are shown in black in figure 2 Also, the data is in integer format, and needs to be changed to double type and normalized to be between 0 and 1. The following code makes all of these changes by running through each image and placing all of the desired pixels into a new vector. The code is written to be robust and dependent on the size of the image imported, so that theoretically less pixelated images with more data points could be run through the same algorithm and have no issues. The new vectors of trimmed data are then stacked up again in a new matrix.

```
% preallocate the new matrix to be filled with trimmed images
images_cut = zeros(size(images,1),(sqrt(size(images,2)) - 2)^2);

% trimming the edges of each image
for i = 1:size(images,1)
    % set counter variable to 0
    count = 1;
    % trimming the data, simply saving the desired parts of the matrix
    for j = 1:size(images,2)
        if j > sqrt(size(images,2)) && j < (size(images,2) - sqrt(size(images,2)))
            if mod(j,sqrt(size(images,2))) ~= 0 && mod(j,sqrt(size(images,2))) ~= 1
```

```
                images_cut(i,count) = images(i,j);
                count = count + 1;
            end
        end
    end
end

% normalize the data, it is already double type but it needs to be divided
images_cut_dub = images_cut./255;
```

The final matrix from all of this code is a double type made of the trimmed and normalized images ready for more processing. This next part of the preprocessing I discovered to be completely unnecessary, after writing the code, but I left it in anyway, as it still works. The thinking was that the data had to be ordered, so that the images that were 0 were first, followed by all the 1s, then 2s, and so on. This was done using the following code, and a lot of hard coding to deal with changing variable names:

```
    % preallocate matrices for each digit, to reorder
char0 = zeros(7000,size(images_cut_dub,2));
char1 = zeros(7000,size(images_cut_dub,2));
char2 = zeros(7000,size(images_cut_dub,2));
char3 = zeros(7000,size(images_cut_dub,2));
char4 = zeros(7000,size(images_cut_dub,2));
char5 = zeros(7000,size(images_cut_dub,2));
char6 = zeros(7000,size(images_cut_dub,2));
char7 = zeros(7000,size(images_cut_dub,2));
char8 = zeros(7000,size(images_cut_dub,2));
char9 = zeros(7000,size(images_cut_dub,2));

% set counter variables
count0 = 1;
count1 = 1;
count2 = 1;
count3 = 1;
count4 = 1;
count5 = 1;
count6 = 1;
count7 = 1;
count8 = 1;
count9 = 1;

% fill the matrices of each digit with the correct images
for k = 1:size(images_cut_dub,1)
    if labels(k) == 0
        char0(count0,:) = images_cut_dub(k,:);
        count0 = count0 + 1;
    elseif labels(k) == 1
```

```
        char1(count1,:) = images_cut_dub(k,:);
        count1 = count1 + 1;
    elseif labels(k) == 2
        char2(count2,:) = images_cut_dub(k,:);
        count2 = count2 + 1;
    elseif labels(k) == 3
        char3(count3,:) = images_cut_dub(k,:);
        count3 = count3 + 1;
    elseif labels(k) == 4
        char4(count4,:) = images_cut_dub(k,:);
        count4 = count4 + 1;
    elseif labels(k) == 5
        char5(count5,:) = images_cut_dub(k,:);
        count5 = count5 + 1;
    elseif labels(k) == 6
        char6(count6,:) = images_cut_dub(k,:);
        count6 = count6 + 1;
    elseif labels(k) == 7
        char7(count7,:) = images_cut_dub(k,:);
        count7 = count7 + 1;
    elseif labels(k) == 8
        char8(count8,:) = images_cut_dub(k,:);
        count8 = count8 + 1;
    elseif labels(k) == 9
        char9(count9,:) = images_cut_dub(k,:);
        count9 = count9 + 1;
    end
end

% get rid of all the extra rows made of zeros
char0(all(~char0,2),:) = [];
char1(all(~char1,2),:) = [];
char2(all(~char2,2),:) = [];
char3(all(~char3,2),:) = [];
char4(all(~char4,2),:) = [];
char5(all(~char5,2),:) = [];
char6(all(~char6,2),:) = [];
char7(all(~char7,2),:) = [];
char8(all(~char8,2),:) = [];
char9(all(~char9,2),:) = [];

% store the number of each digit we have in the each vector
each = zeros(10,1);
each(1) = size(char0,1);
each(2) = size(char1,1);
each(3) = size(char2,1);
```

```
each(4) = size(char3,1);
each(5) = size(char4,1);
each(6) = size(char5,1);
each(7) = size(char6,1);
each(8) = size(char7,1);
each(9) = size(char8,1);
each(10) = size(char9,1);
```

We are now left with matrices that are made of images representing a single digit; these will then be stacked and used to create the hyperplanes.

3.2 Training

The code is just doing the least squares operation with different B vectors to choose the number that the hyperplane represents. This will "train" the algorithm by creating hyperplanes, and then those trained hyperplanes will be used to determine the value of the digit of any mystery image.

```
    % preallocate matrices for creation of hyperplanes
% big_A is all of the images in order with a 1 added to the end of each
% image
big_A = [char0; char1; char2; char3; char4; char5; char6; char7; char8; char9];
big_A(:,size(images_cut_dub,2) + 1) = ones(size(images_cut_dub,1),1);
a = zeros(size(big_A,2),10);
check = 1;

% create hyperplanes
for m = 1:10
    % make a new B vector each time, as we want 10 unique hyperplanes. This
    % will adjust the locations of the positive 1s to the correct locations
    % for each digit
    B = -1*ones(size(images_cut_dub,1),1);
    B(check:(check - 1 + each(m))) = ones(each(m),1);
    check = check + each(m);
    % fill the hyperplane matrix using the least squares closed form
    % solution
    a(:,m) = pinv(big_A)*B;
end
```

3.3 Testing

The testing portion of this project involves performing the initial equations from section 1 using hyperplanes for each image, and then choosing the best fitting image. I did this by testing each image with each hyperplane, and then choosing the digit corresponding to the hyperplane that resulted in the number closest to 1. This was done instead of the positive versus negative distinction, as there could have been two positive choices or no positive choices, so using the closest value to the expected correct value was the right choice.

```
nums = zeros(size(images_cut_test_dub,1),1);

% calculating guesses for each test image
for k = 1:size(images_cut_test_dub,1)
    % reset a results vector to store resulting values of hyperplane
    % multiplication for each digit
    results = zeros(10,1);
    % calculate the result for each hyperplane
    for l = 1:10
        results(l) = a(1:(end - 1),l)'*images_cut_test_dub(k,:)' + a(end,l);
    end
    % reset values for difference values and guess values
    check_diff = 100;
    check_num = -5;
    % check how close to 1 each result value is and the closest will be the
    % guess of the number
    for m = 1:10
        if abs(results(m) - 1) < check_diff
            check_diff = abs(results(m) - 1);
            check_num = m - 1;
        end
    end
    % fill the number vector with the guess after each image is checked
    nums(k) = check_num;
end
```

This results in a vector of numbers that represent the guesses for each image, in the same order as the labels vector.

3.4 Analysis

The confusion matrices are the best way to see the accuracy of the prediction algorithm by outlining all of the false positives and negatives. The following code will create the confusion matrix for the testing data, and the code is identical for the training data.

```
% preallocate the test confusion matrix
confusion_testing = zeros(11);

% fill the confusion matrix with correct values, just increasing the
% correct loaction in the matrix if the guess is right or wrong
for p = 1:numel(labels_test_dub)
    if labels_test_dub(p) == nums(p)
        confusion_testing(nums(p) + 1,nums(p) + 1) =
        confusion_testing(nums(p) + 1,nums(p) + 1) + 1;
    else
        confusion_testing(labels_test(p) + 1,nums(p) + 1) =
        confusion_testing(labels_test(p) + 1,nums(p) + 1) + 1;
```

```
    end
end

% fill the outside edges of the confusion matrix with totals
for q = 1:10
    confusion_testing(end,q) = sum(confusion_testing(1:(end - 1),q));
    confusion_testing(q,end) = sum(confusion_testing(q,1:(end - 1)));
end

% fill the final value in the matrix
confusion_testing(end,end) = sum(confusion_testing(end,1:(end - 1)));
```

Following are the confusion matrices as calculated using my hyperplanes and checking algorithm for both testing and training data.

Prediction

Digit	0	1	2	3	4	5	6	7	8	9	*Total*
0	944	0	1	2	2	7	14	2	7	1	980
1	0	1107	2	2	3	1	5	1	14	0	1135
2	18	54	812	25	14	0	41	23	39	6	1032
3	4	18	23	878	5	17	9	25	21	10	1010
4	0	22	6	1	881	5	10	2	11	44	982
5	23	18	3	72	24	659	24	14	38	17	892
6	18	10	10	0	22	17	874	0	7	0	958
7	5	40	16	6	26	0	1	885	0	49	1028
8	14	46	11	30	27	39	15	13	759	20	974
9	16	11	2	17	80	1	1	75	4	802	1009
All	1042	1326	886	1033	1084	746	994	1040	900	949	10000

Prediction

Digit	0	1	2	3	4	5	6	7	8	9	*Total*
0	5678	7	20	15	25	43	64	5	60	6	5923
1	2	6547	40	15	19	31	14	14	54	6	6742
2	97	267	4787	149	107	11	237	92	192	19	5958
3	40	167	176	5156	32	128	56	116	134	126	6131
4	10	99	43	6	5203	50	39	23	60	309	5842
5	160	94	28	434	105	3986	191	40	240	143	5421
6	109	73	61	0	70	91	5475	0	36	3	5918
7	55	189	37	46	159	9	2	5452	11	305	6265
8	75	492	63	226	103	221	56	21	4417	177	5851
9	67	61	19	115	374	12	4	512	38	4747	5949
All	6293	7996	5274	6162	6197	4582	6138	6275	5242	5841	60000

Next, we can examine the most commonly used pixels for the determination of hyperplanes for different digits using a colormap. The colormap is shown in figure 3.

Finally, we can look at the effectiveness of the algorithm in its prediction of digits by looking at a histogram of the data, where the images assumed to be not the correct digit are shown in red

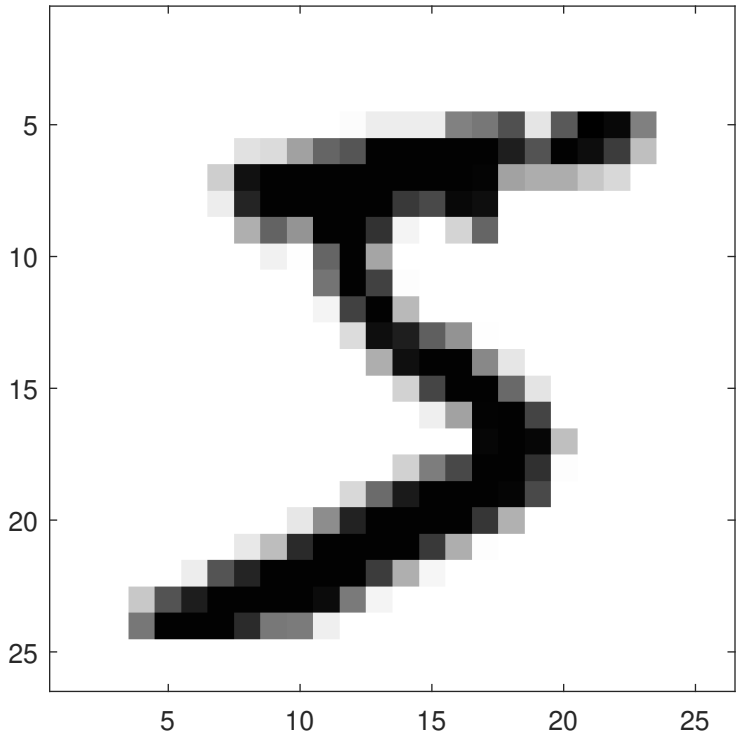

Figure 1: This is an example of a single image of data that will be identified using the prediction algorithm.

and the digits assumed to be correct are shown in blue. This can be seen in figure 4. Both the colormap and the histogram are created using the digit 0.

4 Code

Finally, all of the code used in this project, listed in order. The values in the hyperplanes can be seen by loading the saved data from the end of the training script:

```
clc
clear
close all

% load in the given data
load mnist.mat
```

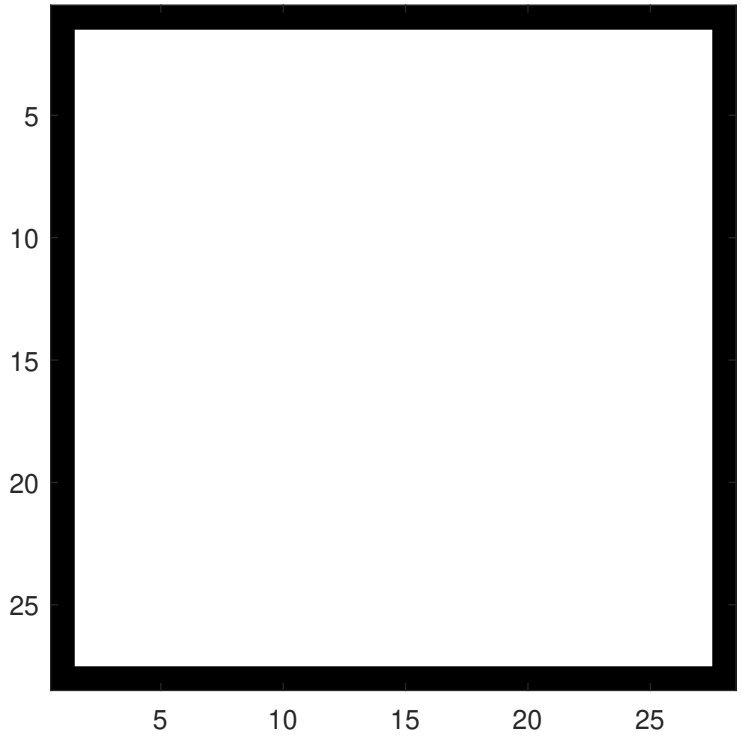

Figure 2: This is a representation of the pixels that get trimmed initially during preprocessing. These pixels were determined to be unnecessary and are removed to improve efficiency.

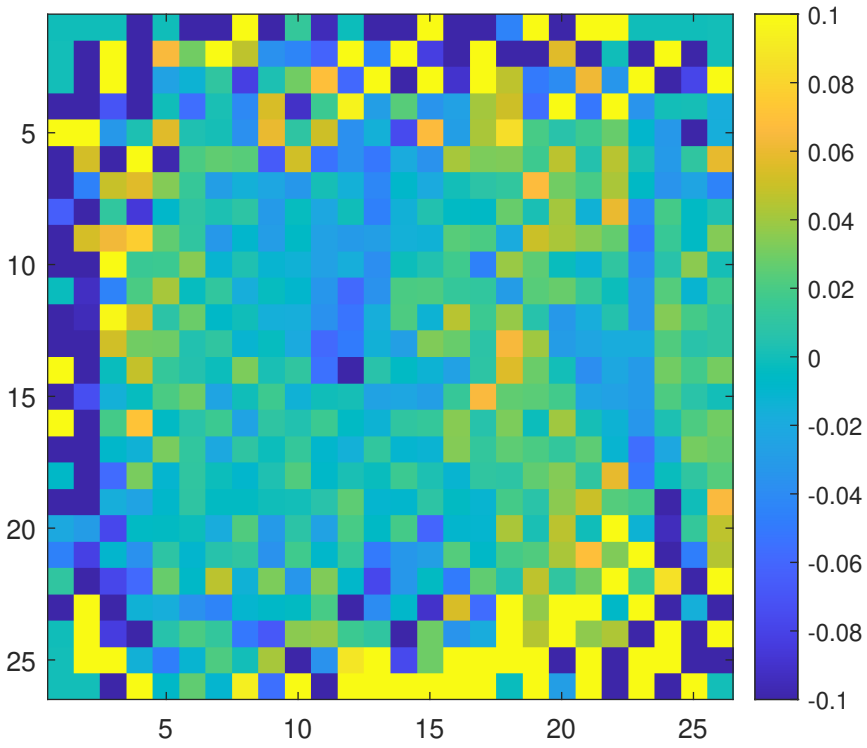

Figure 3: This figure is the representation of the most commonly used pixels in the determination of the digit. The further away from 0 that the value is, the more impactful it is on the final guess.

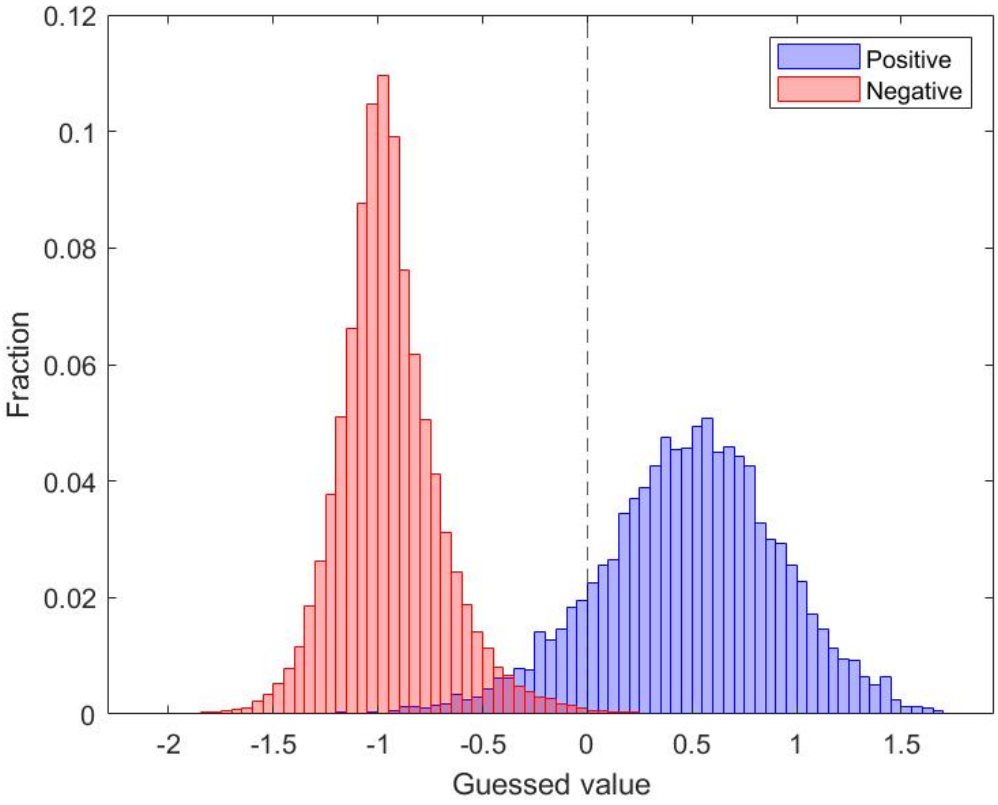

Figure 4: This is a histogram representing the distribution of guesses, and also illustrating the false positives and negatives that are a result of the prediction algorithm.

```
% preallocate the new matrix to be filled with trimmed images
images_cut = zeros(size(images,1),(sqrt(size(images,2)) - 2)^2);

% trimming the edges of each image
for i = 1:size(images,1)
    % set counter variable to 0
    count = 1;
    % trimming the data, simply saving the desired parts of the matrix
    for j = 1:size(images,2)
        if j > sqrt(size(images,2)) && j < (size(images,2) - sqrt(size(images,2)))
            if mod(j,sqrt(size(images,2))) ~= 0 && mod(j,sqrt(size(images,2))) ~= 1
                images_cut(i,count) = images(i,j);
                count = count + 1;
            end
        end
    end
end

% normalize the data, it is already double type but it needs to be divided
images_cut_dub = images_cut./255;

% preallocate matrices for each digit, to reorder
char0 = zeros(7000,size(images_cut_dub,2));
char1 = zeros(7000,size(images_cut_dub,2));
char2 = zeros(7000,size(images_cut_dub,2));
char3 = zeros(7000,size(images_cut_dub,2));
char4 = zeros(7000,size(images_cut_dub,2));
char5 = zeros(7000,size(images_cut_dub,2));
char6 = zeros(7000,size(images_cut_dub,2));
char7 = zeros(7000,size(images_cut_dub,2));
char8 = zeros(7000,size(images_cut_dub,2));
char9 = zeros(7000,size(images_cut_dub,2));

% set counter variables
count0 = 1;
count1 = 1;
count2 = 1;
count3 = 1;
count4 = 1;
count5 = 1;
count6 = 1;
count7 = 1;
count8 = 1;
count9 = 1;

% fill the matrices of each digit with the correct images
```

```
for k = 1:size(images_cut_dub,1)
    if labels(k) == 0
        char0(count0,:) = images_cut_dub(k,:);
        count0 = count0 + 1;
    elseif labels(k) == 1
        char1(count1,:) = images_cut_dub(k,:);
        count1 = count1 + 1;
    elseif labels(k) == 2
        char2(count2,:) = images_cut_dub(k,:);
        count2 = count2 + 1;
    elseif labels(k) == 3
        char3(count3,:) = images_cut_dub(k,:);
        count3 = count3 + 1;
    elseif labels(k) == 4
        char4(count4,:) = images_cut_dub(k,:);
        count4 = count4 + 1;
    elseif labels(k) == 5
        char5(count5,:) = images_cut_dub(k,:);
        count5 = count5 + 1;
    elseif labels(k) == 6
        char6(count6,:) = images_cut_dub(k,:);
        count6 = count6 + 1;
    elseif labels(k) == 7
        char7(count7,:) = images_cut_dub(k,:);
        count7 = count7 + 1;
    elseif labels(k) == 8
        char8(count8,:) = images_cut_dub(k,:);
        count8 = count8 + 1;
    elseif labels(k) == 9
        char9(count9,:) = images_cut_dub(k,:);
        count9 = count9 + 1;
    end
end

% get rid of all the extra rows made of zeros
char0(all(~char0,2),:) = [];
char1(all(~char1,2),:) = [];
char2(all(~char2,2),:) = [];
char3(all(~char3,2),:) = [];
char4(all(~char4,2),:) = [];
char5(all(~char5,2),:) = [];
char6(all(~char6,2),:) = [];
char7(all(~char7,2),:) = [];
char8(all(~char8,2),:) = [];
char9(all(~char9,2),:) = [];
```

```
% store the number of each digit we have in the each vector
each = zeros(10,1);
each(1) = size(char0,1);
each(2) = size(char1,1);
each(3) = size(char2,1);
each(4) = size(char3,1);
each(5) = size(char4,1);
each(6) = size(char5,1);
each(7) = size(char6,1);
each(8) = size(char7,1);
each(9) = size(char8,1);
each(10) = size(char9,1);

% preallocate matrices for creation of hyperplanes
% big_A is all of the images in order with a 1 added to the end of each
% image
big_A = [char0; char1; char2; char3; char4; char5; char6; char7; char8; char9];
big_A(:,size(images_cut_dub,2) + 1) = ones(size(images_cut_dub,1),1);
a = zeros(size(big_A,2),10);
check = 1;

% create hyperplanes
for m = 1:10
    % make a new B vector each time, as we want 10 unique hyperplanes. This
    % will adjust the locations of the positive 1s to the correct locations
    % for each digit
    B = -1*ones(size(images_cut_dub,1),1);
    B(check:(check - 1 + each(m))) = ones(each(m),1);
    check = check + each(m);
    % fill the hyperplane matrix using the least squares closed form
    % solution
    a(:,m) = pinv(big_A)*B;
end

% create A and b in the requested format
A = a(1:(end - 1),:)';
b = a(end,:)';

% save the generated hyperplanes
save Hyperplanes_Ptasiewicz A b
```

Finally, the testing data:

```
clc
clear
close all
```

```
% load given data and data that I created
load mnist.mat
load Hyperplanes_Ptasiewicz

% recreate the data in the way I developed my code
a = [A'; b'];

% preallocate the testing images matrix
images_cut_test = zeros(size(images_test,1),(sqrt(size(images_test,2)) - 2)^2);

% trim the data using the same algorithm as before
for i = 1:size(images_test,1)
    count = 1;
    for j = 1:size(images_test,2)
        if j > sqrt(size(images_test,2)) &&
        j < (size(images_test,2) - sqrt(size(images_test,2)))
            if mod(j,sqrt(size(images_test,2))) ~= 0 &&
            mod(j,sqrt(size(images_test,2))) ~= 1
                images_cut_test(i,count) = images_test(i,j);
                count = count + 1;
            end
        end
    end
end

% normalize the data and preallocate a vector for guesses
images_cut_test_dub = images_cut_test./255;
nums = zeros(size(images_cut_test_dub,1),1);

% calculating guesses for each test image
for k = 1:size(images_cut_test_dub,1)
    % reset a results vector to store resulting values of hyperplane
    % multiplication for each digit
    results = zeros(10,1);
    % calculate the result for each hyperplane
    for l = 1:10
        results(l) = a(1:(end - 1),l)'*images_cut_test_dub(k,:)' + a(end,l);
    end
    % reset values for difference values and guess values
    check_diff = 100;
    check_num = -5;
    % check how close to 1 each result value is and the closest will be the
    % guess of the number
    for m = 1:10
        if abs(results(m) - 1) < check_diff
            check_diff = abs(results(m) - 1);
```

```
            check_num = m - 1;
        end
    end
    % fill the number vector with the guess after each image is checked
    nums(k) = check_num;
end

% change data type of labels
labels_test_dub = double(labels_test);

% preallocate the test confusion matrix
confusion_testing = zeros(11);

% fill the confusion matrix with correct values, just increasing the
% correct loaction in the matrix if the guess is right or wrong
for p = 1:numel(labels_test_dub)
    if labels_test_dub(p) == nums(p)
        confusion_testing(nums(p) + 1,nums(p) + 1) =
        confusion_testing(nums(p) + 1,nums(p) + 1) + 1;
    else
        confusion_testing(labels_test(p) + 1,nums(p) + 1) =
        confusion_testing(labels_test(p) + 1,nums(p) + 1) + 1;
    end
end

% fill the outside edges of the confusion matrix with totals
for q = 1:10
    confusion_testing(end,q) = sum(confusion_testing(1:(end - 1),q));
    confusion_testing(q,end) = sum(confusion_testing(q,1:(end - 1)));
end

% fill the final value in the matrix
confusion_testing(end,end) = sum(confusion_testing(end,1:(end - 1)));

% now do the confusion matrix for the training data, requires some more
% preprocessing

% preallocate the training images matrix
images_cut_training = zeros(size(images,1),(sqrt(size(images,2)) - 2)^2);

% trim the data using the same algorithm as before
for r = 1:size(images,1)
    count_training = 1;
    for s = 1:size(images,2)
        if s > sqrt(size(images,2)) &&
        s < (size(images,2) - sqrt(size(images,2)))
```

```
            if mod(s,sqrt(size(images,2))) ~= 0 &&
            mod(s,sqrt(size(images,2))) ~= 1
                images_cut_training(r,count_training) = images(r,s);
                count_training = count_training + 1;
            end
        end
    end
end

% normalize the data and preallocate a vector for guesses
images_cut_training_dub = images_cut_training./255;
nums_training = zeros(size(images_cut_training_dub,1),1);

% calculating guesses for each training image
for t = 1:size(images_cut_training_dub,1)
    % reset a results vector to store resulting values of hyperplane
    % multiplication for each digit
    results_training = zeros(10,1);
    % calculate the result for each hyperplane
    for u = 1:10
        results_training(u) = a(1:(end - 1),u)'*images_cut_training_dub(t,:)'
        + a(end,u);
    end
    % reset values for difference values and guess values
    check_diff_training = 100;
    check_num_training = -5;
    % check how close to 1 each result value is and the closest will be the
    % guess of the number
    for v = 1:10
        if abs(results_training(v) - 1) < check_diff_training
            check_diff_training = abs(results_training(v) - 1);
            check_num_training = v - 1;
        end
    end
    % fill the number vector with the guess after each image is checked
    nums_training(t) = check_num_training;
end

% change data type of labels
labels_training_dub = double(labels);

% preallocate the test confusion matrix
confusion_training = zeros(11);

% fill the confusion matrix with correct values, just increasing the
% correct loaction in the matrix if the guess is right or wrong
```

```
for w = 1:numel(labels_training_dub)
    if labels_training_dub(w) == nums_training(w)
        confusion_training(nums_training(w) + 1,nums_training(w) + 1) =
        confusion_training(nums_training(w) + 1,nums_training(w) + 1) + 1;
    else
        confusion_training(labels(w) + 1,nums_training(w) + 1) =
        confusion_training(labels(w) + 1,nums_training(w) + 1) + 1;
    end
end

% fill the outside edges of the confusion matrix with totals
for c = 1:10
    confusion_training(end,c) = sum(confusion_training(1:(end - 1),c));
    confusion_training(c,end) = sum(confusion_training(c,1:(end - 1)));
end

% fill the final value in the matrix
confusion_training(end,end) = sum(confusion_training(end,1:(end - 1)));

% image like 14.1
% generate matrix for plotting
img141 = zeros(28);
img141(1,:) = 1;
img141(end,:) = 1;
img141(:,1) = 1;
img141(:,end) = 1;

% plot the created image that illustrates trimming
figure(1)
imagesc(img141)
colormap(flipud(gray(256)))
axis square

% create example image of a number
imgtest = reshape(images_cut_training(1,:),[26, 26]);
figure(2)
imagesc(imgtest')
colormap(flipud(gray(256)))
axis square

% create an image similar to 14.3 using imagesc
% create a mesh to make the colormap
x = linspace(-10,10,26);
[X, Y] = meshgrid(x);
a1 = a(1:(end - 1),3);
a2 = reshape(a1,26,26,[]);
```

```
figure(3)
imagesc(a2,[-.1 .1])
colorbar
axis square

% create a histogram, similar to 14.4

% initialize the histogram I want to create
find_num = 0;

% preallocate vectors for data for the histogram and counter values
histgood = zeros(7000,1);
histbad = zeros(size(images_cut_training_dub,1),1);
countgood = 1;
countbad = 1;

% store each hyperplane result in the correct vector
for ii = 1:size(images,1)
    % calculate the hyperplane result for each image using the hyperplane
    % associated with the data in find_num
    histdata = a(1:(end - 1),find_num + 1)'*images_cut_training_dub(ii,:)'
    + a(end,find_num + 1);
    if labels(ii) == find_num
        histgood(countgood) = histdata;
        countgood = countgood + 1;
    else
        histbad(countbad) = histdata;
        countbad = countbad + 1;
    end
end

% remove the extra zeros at the end of each vector
histgood(all(~histgood,2),:) = [];
histbad(all(~histbad,2),:) = [];

% generate the histogram
figure(4)
histogram(histgood,'Binwidth',.05,'Normalization','probability',
'FaceAlpha',.3,'EdgeColor','b','FaceColor','b')
hold on
histogram(histbad,'Binwidth',.05,'Normalization','probability',
'FaceAlpha',.3,'EdgeColor','r','FaceColor','r')
xline(0,'--')
legend('Positive','Negative')
xlabel('Guessed value')
ylabel('Fraction')
```