Handwriting Recognition using Advanced Binary Classification

Max Ptasiewicz

April 19, 2022

1 Introduction

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. For our initial handwriting recognition problem, we used a least squares formulation as opposed to an SVM (Support-Vector-Machine) process. The SVM formulation allows us several advantages, all of which will result in a more accurate algorithm. The specific techniques will be explained later, but both the least squares formulation and the SVM formulation are based on the following basic equations, but we will modify them to allow for a more effective optimization process:

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

a^T represents the hyperplane that in this case, will separate a number from appearing as a 0 or a 1. 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 a 1. This operation would be repeated with hyperplanes representing combinations of digits 0 through 9, following the pattern of 0 vs 1, 0 vs 2, 0 vs 3, and so on until we get to 8 vs 9.

We take these equations, and allow for the right side of the equation to vary slightly, so that we make our formulation more robust by essentially telling our equations that they do not have to exactly equal a 1 or a -1, as long as they are only off by a small amount. We do this by creating a second set of equations with the addition of a slack variable, in this case a unique u for each image of lower numerical value and a unique v for each image of higher numerical value. This is the power of SVM formulation, which allows us to have these support vectors u and v:

$$\begin{array}{cc} a^T X_{0,1} + b = 1 - u_1 & a^T X_{1,1} + b = -(1 - v_1) \\ a^T X_{0,2} + b = 1 - u_2 & a^T X_{1,2} + b = -(1 - v_2) \\ \vdots & \vdots \\ a^T X_{0,n} + b = 1 - u_n & a^T X_{1,n} + b = -(1 - v_n) \end{array}$$

2 Formulation

This set of equations can be applied to each combination of digits 0 through 9, and we can formulate these equations into a matrix and vector set of equations to use a solver like MATLAB's linprog to solve it. However in this problem, we will be using a solver from cvx.com, which allows for access to multiple solvers of convex optimization problems, and significantly easier formulation. All we need to do is convert this to a comprehensive set of variables, cost function, and linear

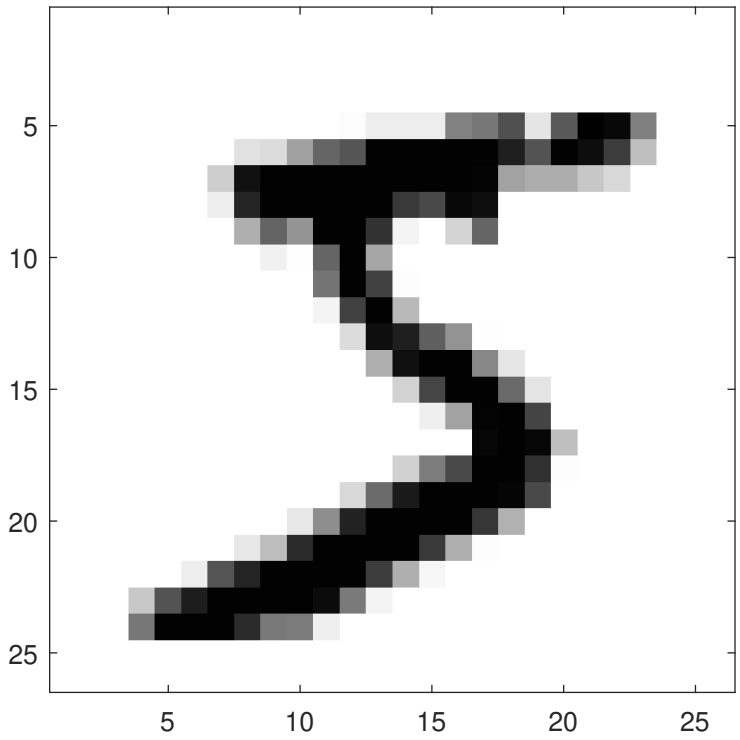

Figure 1: An example of the type of data that we will be using.

constraints. As it happens, I tend to be used to formulating problems with inequality constraints, specifically less than or equal to type inequality constraints. This results in the following variables:

1. A_{solve} which represents the "slopes" of each hyperplane between digit combinations, and has the same size as each of the number of features in this problem, which happens to be 676. Initially, there are 784 features, but through assuming that the outside edge of each image contains features that have no effect on the overall value of the image, and then removing them, we reduce the feature size to 676. These are being called "slopes", as if this were a 2-dimensional problem, these numbers would be a single number, and would be a slope as we commonly envision one, and it would represent a line, which is essentially a 2-dimensional hyperplane.

2. b_{solve} which represents the "y-intercept" of each hyperplane. Again, this is an analogous value to the y-intercept that we normally think of, but since this hyperplane is in multiple dimensions we do not have an effective way to visualize this, unless we imagine a similar 2-dimensional situation. The size of this variable is the same as a 2-dimensional y-intercept, or a single value.

3. u which represents the slack allowed for the image of lower numerical value, and is a unique value for each number of images of each digit, which is about 6000 long, but varies for each digit.

4. v which represents the slack allowed for the image of higher numerical value, and is again a unique value for each number of images of each digit, which is about 6000 long, but varies for each digit. Has the same function as u, but needs to have a different name to differentiate and organize results.

In addition to the variables, we need to create a cost function. The point of the cost function in this problem is to maximize the distance between the hyperplane equation greater than or equal to 1 and the hyperplane equation less than or equal to -1. This is done to make the formulation more robust against outliers, as the larger the margin is, the closer to the actual limit of the desired data set. This is best explained using a 2-dimensional analogy once again, and taking the theoretical hyperplanes separating blue and red dots, named $ax - b \leq -1$ and $ax - b \geq 1$:

The margin is represented algebraically by $\frac{2}{||a||}$, so minimizing $||a||$ will maximize the margin. We also want to minimize the slack variables, so we will add the sum of each of these to the minimization function. In MATLAB, specifically within the CVX format, the cost function is as follows:

```
minimize(norm(A_solve,2) + gamma*(sum(u) + sum(v)))
```

There is a previously unmentioned addition to this formulation, a γ variable that will let us choose the weight difference between the maximization of the margin and the minimization of the slack variables. This value is set before the CVX solver begins, but different values will be tested to find the most effective one.

Finally, we have to create constraints, which have already been mostly created, as they strongly resemble the starting equations. CVX will really take any form of a constraint, but I chose to formulate them as less than or equal to constraints, as I am used to that type of formulation, but really it was just me doing extra work. The constraints used in the CVX formulation are as follows:

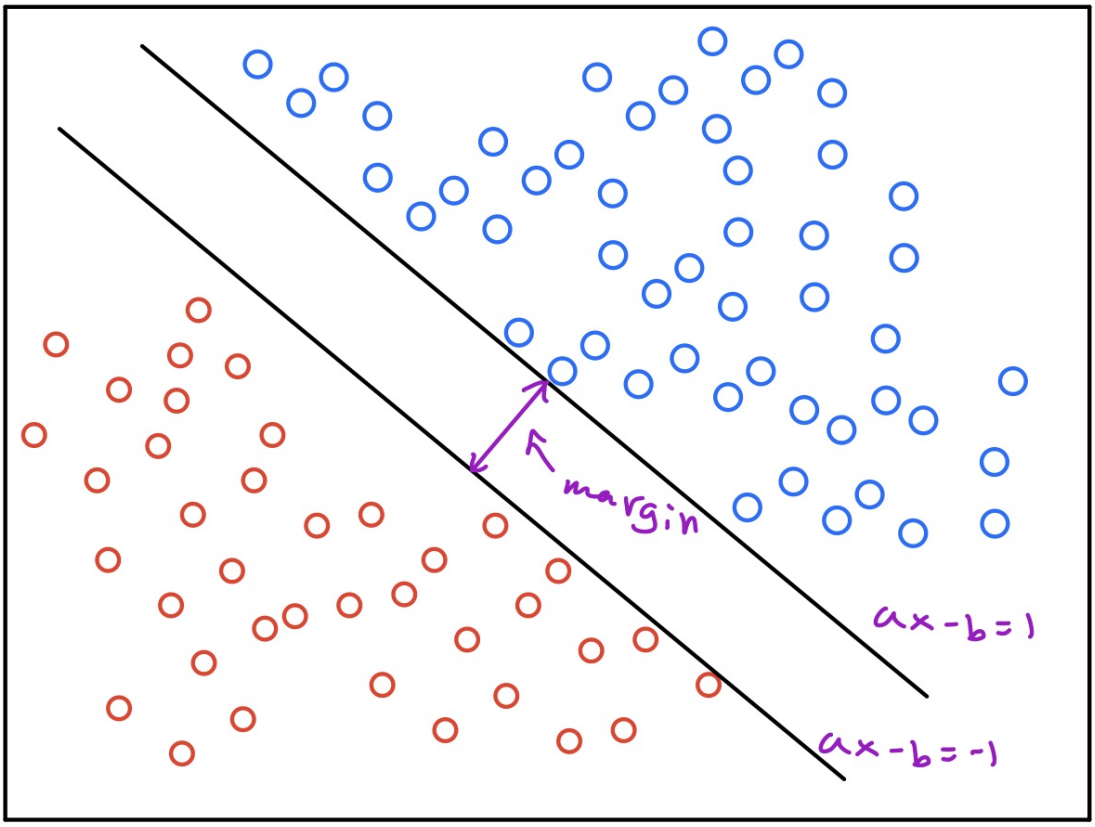

Figure 2: Demonstration of the concept of a margin applied to a 2-dimensional data set.

$$-xa * A_{solve} + b_{solve} - u \leq -ones$$

$$xb * A_{solve} - b_{solve} - v \leq -ones$$

$$u \geq 0$$

$$v \geq 0$$

This is the last part of the formulation for use in CVX, except for the choice of solver to use. This was done simply by testing the different solvers available in CVX, and choosing the fastest one. This ended up being the mosek solver, which converges to a resulting set of variables in just a few seconds, for any single hyperplane. The final CVX formulation follows:

```
cvx_begin
    cvx_solver mosek
    variables A_solve(size(big_A,2)) b_solve u(each(m)) v(each(n))
    minimize(norm(A_solve,2) + gamma*(sum(u) + sum(v)))
    subject to
        -xa*A_solve + b_solve - u <= -ones(each(m),1)
        xb*A_solve - b_solve - v <= -ones(each(n),1)
        u >= 0
        v >= 0
cvx_end
```

3 Analysis

After the CVX formulation is complete, we need to run it the correct amount of times with the correct data sets, and check several different values of γ to determine the best one. This was done by first sorting the training data into sets that contained only one type of a number, about 6000 of each. These were then sent into a for loop that found the hyperplane between 0 and 1, 0 and 2, 0 and 3, all the way until between 8 and 9, using the CVX formulation with the mosek solver shown above. For each hyperplane, 9 different values of γ were tested, from 0.0001 to 10000, by powers of 10, as the size of γ needed to be adjusted drastically to have any affect on the outcome of the solver. Each of the results of these hyperplane creations were then tested for accuracy with the training data used to create the hyperplanes, and the most accurate hyperplanes were chosen. Additionally, a run was done that found each hyperplane with a single value of γ, instead of using several and finding the best one. The γ used here was 0.01, which was determined after testing several single γ values. Each of these sets of hyperplanes were tested individually, on both the testing and training data, and this yielded very interesting results, discussed in the next section.

The testing for each hyperplane was done using a Directed Acyclic Graph, or DAG, which looks a little bit like a plinko board, where a given image travels down from the top, moving left or right given the outcome of the specific hyperplane multiplication operation. At the top, we test every image using the hyperplane between 0 and 9. If it is determined to be 0, the only thing we can actually say about the image is that it is not a 9. If we say that the image is not a 9, this means that the hyperplane multiplication resulted in a positive number, sending the image to the right. If the hyperplane multiplication results in a negative number, we say that the image is not a 0, and send it to the left. There are then two hyperplane nodes below the 0 vs 9 hyperplane

node, the one on the left is a 0 vs 8 and the one on the right is a 1 vs 9. The same process is repeated, sending the image to the left or right appropriately. This repeats until we have a row of 9 hyperplanes, at the bottom, from the left separating 0 vs 1, then 1 vs 2, then 2 vs 3, until 8 vs 9. At this point, each hyperplane is separating two consecutive numbers, and results in the final determination of the mystery digit. This was done in MATLAB in an extremely inefficient manner, using 45 individual functions representing the 45 nodes in the DAG, and the 45 hyperplanes. Each node takes in an image and all of the hyperplanes, and then outputs by calling the next node in the chain, depending on the sign of the resulting hyperplane multiplication. At the end, the 9 node functions at the bottom output simply a number, representing the final guess. Given is an example of a node function, node 9, which happens to be the top node using the hyperplane between 0 and 9:

```
function [next] = node9(img,A,b)

if img*A{1,10} - b{1,10} > 0
    next = node8(img,A,b);
else
    next = node17(img,A,b);
end

end
```

4 Discussion

The overall goal of this project was to improve the accuracy of the initial project, which can be seen in several ways. Three sets of confusion matrices will be included, as well as three sets of accuracy metrics, which are simply a reported value of each of the percentage of correct guesses for each test. First, the confusion matrix for the testing data from the first project, with an accuracy of 86.01%:

				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

Secondly, the confusion matrix for the training data from the first project, with an accuracy of 85.75%:

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 have the confusion matrix for the testing data of the runs where the γ was chosen for each specific hyperplane, based on the accuracy of testing the hyperplane with the training data, with an overall accuracy of 91.66%:

Prediction

Digit	0	1	2	3	4	5	6	7	8	9	*Total*
0	938	3	7	18	13	15	6	7	13	6	1026
1	2	1105	9	15	10	10	9	7	12	3	1182
2	3	1	925	12	10	16	4	12	9	5	997
3	4	3	6	953	7	17	13	12	10	4	1029
4	3	0	6	18	938	18	5	8	6	9	1011
5	2	3	5	13	13	794	2	12	8	5	857
6	0	1	8	10	5	12	899	11	11	4	961
7	4	1	8	16	14	12	4	958	13	4	1034
8	7	2	8	12	9	21	3	7	861	4	934
9	0	3	13	17	4	21	2	11	7	891	969
All	963	1122	995	1084	1023	936	947	1045	950	935	10000

Next, we have the confusion matrix for the training data of the runs where the γ was chosen for each specific hyperplane, based on the accuracy of testing the hyperplane with the training data, with an overall accuracy of 98.47%:

Prediction

Digit	0	1	2	3	4	5	6	7	8	9	*Total*
0	5923	0	4	24	6	13	4	15	9	13	6011
1	0	6742	10	24	8	25	3	18	13	13	6856
2	0	0	5870	13	6	13	5	10	13	11	5941
3	0	0	7	5972	10	17	2	12	14	11	6045
4	0	0	5	18	5798	22	5	16	8	9	5881
5	0	0	10	14	7	5288	2	11	14	11	5357
6	0	0	8	18	11	15	5920	11	9	15	6007
7	0	0	6	20	4	24	4	6205	13	18	6294
8	0	0	4	22	11	11	1	18	5667	10	5744
9	0	0	5	24	15	13	1	10	14	5782	5864
All	5923	6742	5929	6149	5876	5441	5947	6326	5774	5893	60000

Now we have the confusion matrices for the data where a single γ of 0.01 was used for all of the hyperplanes, starting with the testing data with an accuracy of 94.50%:

Prediction

Digit	0	1	2	3	4	5	6	7	8	9	*Total*
0	968	3	4	7	5	4	4	2	8	4	1009
1	3	1123	5	10	7	8	6	4	11	7	1184
2	2	1	970	7	12	7	4	5	6	3	1017
3	4	0	5	963	8	7	7	5	7	5	1011
4	5	0	5	5	940	13	4	5	3	7	987
5	4	0	5	8	4	808	2	10	7	5	853
6	2	1	7	7	4	5	933	4	7	3	973
7	2	3	5	7	7	9	5	975	10	3	1026
8	4	3	2	8	5	10	3	8	896	5	944
9	1	5	8	7	5	16	5	7	5	937	996
All	995	1139	1016	1029	997	887	973	1025	960	979	10000

Finally the confusion matrix for the training data, using a γ value of 0.01, resulting in an accuracy of 95.15%:

					Prediction						
Digit	0	1	2	3	4	5	6	7	8	9	*Total*
0	5813	10	26	48	26	48	24	33	31	31	6090
1	11	6621	37	39	33	60	34	44	40	35	6954
2	11	22	5625	31	25	42	22	32	39	40	5889
3	9	16	26	5755	32	62	30	26	35	28	6019
4	12	19	24	33	5657	51	11	30	40	24	5901
5	13	15	41	25	40	5085	23	27	26	27	5322
6	10	12	28	39	40	51	5782	25	31	31	6049
7	13	17	30	37	37	57	16	6057	30	35	6329
8	11	13	21	39	39	37	18	30	5427	23	5658
9	13	14	30	37	29	47	22	26	32	5539	5789
All	5916	6759	5888	6083	5958	5540	5982	6330	5731	5813	60000

Now overall, the the accuracy was significantly increased when comparing the results of the first project to the results of the new project, but the results of the new project are somewhat confusing. When iterating through several γ values and then testing the accuracy of each using the training data, the resulting hyperplanes actually performed poorly on the testing data, with only about 92% accuracy. However this γ testing was done with the training data, and when this data was run through the DAG, the accuracy shot up to close to 99%, which is extremely accurate, however it is expected. The mystery is why the testing data accuracy is so much worse. Presumably, if the testing data were used to train the γ values, then the accuracy when run through the DAG would be much higher, however this does not exactly follow the rules, in terms of training with training data and testing with testing data. Moreover, the accuracy of both the training and testing data when run through the DAG after the hyperplanes were trained with a single value of γ equal to 0.01 equalized to both be around 95%. This was unexpected, as the accuracy without changing the γ value for the testing data is higher than implementing a "smart" algorithm. Additionally, creating hyperplanes using a single value of γ is significantly faster than testing several values, about 10 times as fast when testing with 9 γ values, which makes sense. This could be a possible advantage to using only a single value of γ instead of trying to make a "smart" algorithm. No matter what, however, the implementation of an SVM formulation always proved to be more effective than using a least squares formulation, as in every case the accuracy improved by at least 7%.

5 Conclusion

As stated before, this project was a success in terms of achieving an accuracy of 95%, which was acheived with a γ of 0.01 for every hyperplane. This shows the effectiveness of the SVM algorithm over the least squares algorithm, and additionally showcases the power of the CVX solvers. Using CVX allowed for significantly easier formulation of the problem, as well as fast solutions using the mosek solver. Something that would have improved this accuracy for the testing data would be to choose a γ for each hyperplane based on the accuracy of each γ value for the testing data, as opposed to the training data, but using testing data in a training algorithm breaks the rules of training a machine learning algorithm. Something that could be done to possibly remedy this is

to have three unique sets of data, one for creating a hyperplane, a second for testing the accuracy of each hyperplane for each γ, and a third to use as normal testing data. Furthermore, there are several issues with my implementation of the DAG, not that it is inaccurate, just that it is time consuming to create. Upon its creation, I realized I could do all 500ish lines of the DAG in about 20, using a recursively defined function. However, my inefficient DAG had already been created, and had proven to be effective. Next, feature engineering could help either the speed at which this algorithm runs, through reducing feature size by averaging the value of several pixels to create a larger one representative of, for example, a nine pixel square. While increasing the speed, this would presumably also decrease the accuracy, as we would lose out on some information. Something extremely similar could be done using a process to do the opposite, where a procedure that would increase the feature size, by maybe turning one pixel into nine, and then assigning those values based on the gradient of the pixels around it could help create a less pixelated image that would presumably increase the accuracy while slowing down the algorithm. Lastly, this algorithm with the right training data could be used to separate not just numbers, but also letters, upper and lowercase, as well as special characters, and when paired with an image processing algorithm could be used to interpret entire sentences. All that is missing is more training data.

6 Code

First, the training algorithm: The one shown is the version that check several γ values and chooses the best resulting hyperplanes. To switch to a single γ value of 0.01, simply switch the comment from line 134 to line 133. (several lines of code are placed on multiple lines to avoid interfering with the margins).

```
clc
clear
close all

tic

% load in the given data
load mnist.mat

% 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);

neweach = zeros(10,1);
ind_count = 0;
for l = 1:numel(each)
    neweach(l) = ind_count + 1;
    ind_count = ind_count + each(l);
end
neweach(11) = 60001;

big_A = [char0; char1; char2;
char3; char4; char5;
char6; char7; char8;
char9];

A = cell(10);
b = cell(10);

for m = 1:9
    for n = (m + 1):10
        A_int = cell(1,1);
        b_int = cell(1,1);
        gammas = [1e-4, 1e-3, 1e-2, 1e-1,
        1e0, 1e1, 1e2, 1e3, 1e4];
        %gammas = 1e-2;
        next = 1;
        for gamma = gammas
            xa = big_A((neweach(m):(neweach(m + 1) - 1)),:);
            xb = big_A((neweach(n):(neweach(n + 1) - 1)),:);
                cvx_begin
                    cvx_solver mosek
                    variables A_solve(size(big_A,2)) b_solve
                    u(each(m)) v(each(n))
                    minimize(norm(A_solve,2) +
                    gamma*(sum(u) + sum(v)))
                    subject to
                        -xa*A_solve + b_solve - u <=
                        -ones(each(m),1)
                        xb*A_solve - b_solve - v <=
                        -ones(each(n),1)
                        u >= 0
                        v >= 0
```

```
                cvx_end
            A_int{next} = A_solve;
            b_int{next} = b_solve;
            next = next + 1;
        end
        percentage_check = zeros(numel(A_int),1);
        xcheck = [xa; xb];
        for c = 1:numel(A_int)
            right_check = 0;
            for p = 1:(size(xcheck,1))
                guess_check = xcheck(p,:)*A_int{c} - b_int{c};
                if guess_check > 0 && p <= size(xa,1)
                    right_check = right_check + 1;
                elseif guess_check < 0 && p > size(xa,1)
                    right_check = right_check + 1;
                end
            end
            percentage_check(c) =
            right_check/(size(xcheck,1));
        end
        check_ind = max(percentage_check);
        for f = 1:numel(percentage_check)
            if percentage_check(f) == check_ind
                use_ind = f;
            end
        end
        A{m,n} = A_int{use_ind};
        b{m,n} = b_int{use_ind};
    end
end

save('DAG_Ptasiewicz','A','b')

beep

toc
```

Finally, we have the testing algorithm, which is extremely long due to the DAG. Again, some lines have been separated into multiple as to not affect the margins.

```
clc
clear
close all

load('sumhypemosek.mat')
load('mnist.mat')
```

```
% preallocate the new matrix to be filled with trimmed images
images_cut_training =
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_training = 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_training(i,count_training) =
                images(i,j);
                count_training = count_training + 1;
            end
        end
    end
end

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

% preallocate matrices for each digit, to reorder
char0_training = zeros(7000,size(images_cut_dub_training,2));
char1_training = zeros(7000,size(images_cut_dub_training,2));
char2_training = zeros(7000,size(images_cut_dub_training,2));
char3_training = zeros(7000,size(images_cut_dub_training,2));
char4_training = zeros(7000,size(images_cut_dub_training,2));
char5_training = zeros(7000,size(images_cut_dub_training,2));
char6_training = zeros(7000,size(images_cut_dub_training,2));
char7_training = zeros(7000,size(images_cut_dub_training,2));
char8_training = zeros(7000,size(images_cut_dub_training,2));
char9_training = zeros(7000,size(images_cut_dub_training,2));

% set counter variables
count0_training = 1;
count1_training = 1;
count2_training = 1;
count3_training = 1;
count4_training = 1;
```

```
count5_training = 1;
count6_training = 1;
count7_training = 1;
count8_training = 1;
count9_training = 1;

% fill the matrices of each digit with the correct images
for k = 1:size(images_cut_dub_training,1)
    if labels(k) == 0
        char0_training(count0_training,:) =
        images_cut_dub_training(k,:);
        count0_training = count0_training + 1;
    elseif labels(k) == 1
        char1_training(count1_training,:) =
        images_cut_dub_training(k,:);
        count1_training = count1_training + 1;
    elseif labels(k) == 2
        char2_training(count2_training,:) =
        images_cut_dub_training(k,:);
        count2_training = count2_training + 1;
    elseif labels(k) == 3
        char3_training(count3_training,:) =
        images_cut_dub_training(k,:);
        count3_training = count3_training + 1;
    elseif labels(k) == 4
        char4_training(count4_training,:) =
        images_cut_dub_training(k,:);
        count4_training = count4_training + 1;
    elseif labels(k) == 5
        char5_training(count5_training,:) =
        images_cut_dub_training(k,:);
        count5_training = count5_training + 1;
    elseif labels(k) == 6
        char6_training(count6_training,:) =
        images_cut_dub_training(k,:);
        count6_training = count6_training + 1;
    elseif labels(k) == 7
        char7_training(count7_training,:) =
        images_cut_dub_training(k,:);
        count7_training = count7_training + 1;
    elseif labels(k) == 8
        char8_training(count8_training,:) =
        images_cut_dub_training(k,:);
        count8_training = count8_training + 1;
    elseif labels(k) == 9
        char9_training(count9_training,:) =
```

```
        images_cut_dub_training(k,:);
        count9_training = count9_training + 1;
    end
end

% get rid of all the extra rows made of zeros
char0_training(all(~char0_training,2),:) = [];
char1_training(all(~char1_training,2),:) = [];
char2_training(all(~char2_training,2),:) = [];
char3_training(all(~char3_training,2),:) = [];
char4_training(all(~char4_training,2),:) = [];
char5_training(all(~char5_training,2),:) = [];
char6_training(all(~char6_training,2),:) = [];
char7_training(all(~char7_training,2),:) = [];
char8_training(all(~char8_training,2),:) = [];
char9_training(all(~char9_training,2),:) = [];

% store the number of each digit we have in the each vector
each_training = zeros(10,1);
each_training(1) = size(char0_training,1);
each_training(2) = size(char1_training,1);
each_training(3) = size(char2_training,1);
each_training(4) = size(char3_training,1);
each_training(5) = size(char4_training,1);
each_training(6) = size(char5_training,1);
each_training(7) = size(char6_training,1);
each_training(8) = size(char7_training,1);
each_training(9) = size(char8_training,1);
each_training(10) = size(char9_training,1);

big_A_training =[char0_training;char1_training;
char2_training;char3_training; char4_training;
char5_training;char6_training; char7_training;
char8_training;char9_training];

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

% trimming the edges of each image
for i = 1:size(images_test,1)
    % set counter variable to 0
    count = 1;
    % trimming the data, simply saving the desired parts of
    the matrix
    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(i,count) = images_test(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_test(k) == 0
        char0(count0,:) = images_cut_dub(k,:);
        count0 = count0 + 1;
    elseif labels_test(k) == 1
        char1(count1,:) = images_cut_dub(k,:);
        count1 = count1 + 1;
```

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

big_A = [char0; char1; char2;
         char3; char4; char5;
         char6; char7; char8;
         char9];

% data sorted

guesses = zeros(sum(each),1);

for p = 1:(sum(each))
    img = big_A(p,:);
    guesses(p) = node9(img,A,b);
end

key = sort(labels_test);

num_right = 0;
for q = 1:numel(guesses)
    if guesses(q) == key(q)
        num_right = num_right + 1;
    end
end

percentage = num_right/numel(guesses);

guesses_training = zeros(sum(each_training),1);

for p = 1:(sum(each_training))
    img = big_A_training(p,:);
    guesses_training(p) = node9(img,A,b);
end

key_training = sort(labels);

num_right_training = 0;
for q = 1:numel(guesses_training)
    if guesses_training(q) == key_training(q)
        num_right_training = num_right_training + 1;
    end
end
```

```
percentage_training =
num_right_training/numel(guesses_training);


labels_testing_dub = double(key);
labels_training_dub = double(key_training);

% 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 w = 1:numel(labels_testing_dub)
    if labels_testing_dub(w) == guesses(w)
        confusion_testing(guesses(w) + 1,
        guesses(w) + 1) =
        confusion_testing(guesses(w) + 1,
        guesses(w) + 1) + 1;
    else
        confusion_testing(labels_test(w) + 1,
        guesses(w) + 1) =
        confusion_testing(labels_test(w) + 1,
        guesses(w) + 1) + 1;
    end
end

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

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

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

% fill the confusion matrix with correct values, just
increasing the
```

```
% correct location in the matrix if the guess is right or
wrong
for w = 1:numel(labels_training_dub)
    if labels_training_dub(w) == guesses_training(w)
        confusion_training(guesses_training(w) + 1,
        guesses_training(w) + 1) =
        confusion_training(guesses_training(w) + 1,
        guesses_training(w) + 1) + 1;
    else
        confusion_training(labels(w) + 1,
        guesses_training(w) + 1) =
        confusion_training(labels(w) + 1,
        guesses_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)));
```

```
function [next] = node1(img,A,b)

if img*A{1,2} - b{1,2} > 0
    next = 0;
else
    next = 1;
end

end

function [next] = node2(img,A,b)

if img*A{1,3} - b{1,3} > 0
    next = node1(img,A,b);
```

```
else
    next = node10(img,A,b);
end

end

function [next] = node3(img,A,b)

if img*A{1,4} - b{1,4} > 0
    next = node2(img,A,b);
else
    next = node11(img,A,b);
end

end

function [next] = node4(img,A,b)

if img*A{1,5} - b{1,5} > 0
    next = node3(img,A,b);
else
    next = node12(img,A,b);
end

end

function [next] = node5(img,A,b)

if img*A{1,6} - b{1,6} > 0
    next = node4(img,A,b);
else
    next = node13(img,A,b);
end

end

function [next] = node6(img,A,b)

if img*A{1,7} - b{1,7} > 0
    next = node5(img,A,b);
else
    next = node14(img,A,b);
end

end
```

```
function [next] = node7(img,A,b)

if img*A{1,8} - b{1,8} > 0
    next = node6(img,A,b);
else
    next = node15(img,A,b);
end

end

function [next] = node8(img,A,b)

if img*A{1,9} - b{1,9} > 0
    next = node7(img,A,b);
else
    next = node16(img,A,b);
end

end

function [next] = node9(img,A,b)

if img*A{1,10} - b{1,10} > 0
    next = node8(img,A,b);
else
    next = node17(img,A,b);
end

end

function [next] = node10(img,A,b)

if img*A{2,3} - b{2,3} > 0
    next = 1;
else
    next = 2;
end

end

function [next] = node11(img,A,b)

if img*A{2,4} - b{2,4} > 0
    next = node10(img,A,b);
else
    next = node18(img,A,b);
```

```
end

end

function [next] = node12(img,A,b)

if img*A{2,5} - b{2,5} > 0
    next = node11(img,A,b);
else
    next = node19(img,A,b);
end

end

function [next] = node13(img,A,b)

if img*A{2,6} - b{2,6} > 0
    next = node12(img,A,b);
else
    next = node20(img,A,b);
end

end

function [next] = node14(img,A,b)

if img*A{2,7} - b{2,7} > 0
    next = node13(img,A,b);
else
    next = node21(img,A,b);
end

end

function [next] = node15(img,A,b)

if img*A{2,8} - b{2,8} > 0
    next = node14(img,A,b);
else
    next = node22(img,A,b);
end

end

function [next] = node16(img,A,b)
```

```
if img*A{2,9} - b{2,9} > 0
    next = node15(img,A,b);
else
    next = node23(img,A,b);
end

end

function [next] = node17(img,A,b)

if img*A{2,10} - b{2,10} > 0
    next = node16(img,A,b);
else
    next = node24(img,A,b);
end

end

function [next] = node18(img,A,b)

if img*A{3,4} - b{3,4} > 0
    next = 2;
else
    next = 3;
end

end

function [next] = node19(img,A,b)

if img*A{3,5} - b{3,5} > 0
    next = node18(img,A,b);
else
    next = node25(img,A,b);
end

end

function [next] = node20(img,A,b)

if img*A{3,6} - b{3,6} > 0
    next = node19(img,A,b);
else
    next = node26(img,A,b);
end
```

```
end

function [next] = node21(img,A,b)

if img*A{3,7} - b{3,7} > 0
    next = node20(img,A,b);
else
    next = node27(img,A,b);
end

end

function [next] = node22(img,A,b)

if img*A{3,8} - b{3,8} > 0
    next = node21(img,A,b);
else
    next = node28(img,A,b);
end

end

function [next] = node23(img,A,b)

if img*A{3,9} - b{3,9} > 0
    next = node22(img,A,b);
else
    next = node29(img,A,b);
end

end

function [next] = node24(img,A,b)

if img*A{3,10} - b{3,10} > 0
    next = node23(img,A,b);
else
    next = node30(img,A,b);
end

end

function [next] = node25(img,A,b)

if img*A{4,5} - b{4,5} > 0
    next = 3;
```

```
else
    next = 4;
end

end

function [next] = node26(img,A,b)

if img*A{4,6} - b{4,6} > 0
    next = node25(img,A,b);
else
    next = node31(img,A,b);
end

end

function [next] = node27(img,A,b)

if img*A{4,7} - b{4,7} > 0
    next = node26(img,A,b);
else
    next = node32(img,A,b);
end

end

function [next] = node28(img,A,b)

if img*A{4,8} - b{4,8} > 0
    next = node27(img,A,b);
else
    next = node33(img,A,b);
end

end

function [next] = node29(img,A,b)

if img*A{4,9} - b{4,9} > 0
    next = node28(img,A,b);
else
    next = node34(img,A,b);
end

end
```

```
function [next] = node30(img,A,b)

if img*A{4,10} - b{4,10} > 0
    next = node29(img,A,b);
else
    next = node35(img,A,b);
end

end

function [next] = node31(img,A,b)

if img*A{5,6} - b{5,6} > 0
    next = 4;
else
    next = 5;
end

end

function [next] = node32(img,A,b)

if img*A{5,7} - b{5,7} > 0
    next = node31(img,A,b);
else
    next = node36(img,A,b);
end

end

function [next] = node33(img,A,b)

if img*A{5,8} - b{5,8} > 0
    next = node32(img,A,b);
else
    next = node37(img,A,b);
end

end

function [next] = node34(img,A,b)

if img*A{5,9} - b{5,9} > 0
    next = node33(img,A,b);
else
    next = node38(img,A,b);
```

```
end

end

function [next] = node35(img,A,b)

if img*A{5,10} - b{5,10} > 0
    next = node34(img,A,b);
else
    next = node39(img,A,b);
end

end

function [next] = node36(img,A,b)

if img*A{6,7} - b{6,7} > 0
    next = 5;
else
    next = 6;
end

end

function [next] = node37(img,A,b)

if img*A{6,8} - b{6,8} > 0
    next = node36(img,A,b);
else
    next = node40(img,A,b);
end

end

function [next] = node38(img,A,b)

if img*A{6,9} - b{6,9} > 0
    next = node37(img,A,b);
else
    next = node41(img,A,b);
end

end

function [next] = node39(img,A,b)
```

```
if img*A{6,10} - b{6,10} > 0
    next = node38(img,A,b);
else
    next = node42(img,A,b);
end

end

function [next] = node40(img,A,b)

if img*A{7,8} - b{7,8} > 0
    next = 6;
else
    next = 7;
end

end

function [next] = node41(img,A,b)

if img*A{7,9} - b{7,9} > 0
    next = node40(img,A,b);
else
    next = node43(img,A,b);
end

end

function [next] = node42(img,A,b)

if img*A{7,10} - b{7,10} > 0
    next = node41(img,A,b);
else
    next = node44(img,A,b);
end

end

function [next] = node43(img,A,b)

if img*A{8,9} - b{8,9} > 0
    next = 7;
else
    next = 8;
end
```

```
end

function [next] = node44(img,A,b)

if img*A{8,10} - b{8,10} > 0
    next = node43(img,A,b);
else
    next = node45(img,A,b);
end

end

function [next] = node45(img,A,b)

if img*A{9,10} - b{9,10} > 0
    next = 8;
else
    next = 9;
end

end
```