AMATH 563 HW 1

Promoting Sparsity in Overdetermined Linear Systems

Zachary McNulty
zmcnulty, ID: 1636402

Abstract: Linear systems are fundamental problems found in many applications of data science. However, due to the nature of data itself these systems are typically highly overdetermined or underdetermined. In these cases, it usually not obvious which solution best captures the dynamics of the system of interest. In this paper, we explore a range of available techniques to classification on the MNIST database of handwritten digits.

University of Washington
April 2019

1 Introduction and Overview

The linear systems that often arise in data science are typically highly over or underdetermined. In the latter case, there are likely infinitely many solutions and in the former there are likely none. As a result, some metric must be defined to decide which of the infinitely many solutions is the best one in the underdetermined case and which of the infinitely many non-solutions is the "closest" in the overdetermined case. These choices we make to sift through solution-space often are defined based on desirable properties we wish the solution to have. One common quality to have is the idea of sparsity: a good solution with as many zeros as possible. This adopts the idea that the simplest explanation is often the best as it generates a model with as few relevant variables as possible. Not only does sparsity help develop simpler models that are easier to interpret, but it also tends to choose parameter values in a way that is robust in the presence of noise. In this paper, we will explore some of the numerous techniques available and highlight their assorted properties. We will use the MNIST database which contains tens of thousands of handwritten digits as our case study and attempt to build a linear classifier. As we will see, this problem can be condensed into a simple overdetermined linear system.

2 Theoretical Background

Consider a linear system $Ax = b$. There will likely be no solution in the overdetermined case: for any assignment of the variables we look at there will always be some amount of error between the predicted value and the true value. Naturally, the goal will be to find a solution that minimizes this error. How we measure this error is another question. Suppose we have some system $f(x \mid \theta)$ parameterized by θ. Furthermore, when we measure this system to collect our data, our measurements are noisy. Assuming zero-mean Gaussian noise, our measurements y_i will be $y_i = f(x_i \mid \theta) + error_i$ with $error_i \sim \mathcal{N}(0, \sigma^2)$. This implies $y_i \sim \mathcal{N}(f(x_i \mid \theta), \sigma^2)$. Thus, our data is a series of pairs $\{(x_i, y_i)\}$ and our goal is to determine the Maximum Likelihood Estimate of the parameter θ. Let ϕ be the Gaussian pdf:

$$P(y_1, y_2, .., y_n \mid x_1, x_2, ..., x_n, \theta) \sim L_n(\theta) = \prod_{i=1}^{n} \phi(y_i \mid f(x_i \mid \theta), \sigma^2) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-(y_i - f(x_i;\theta))^2/(2\sigma^2)}$$

$$log(L_n(\theta)) = \sum_{i=1}^{n} \left(log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{(y_i - f(x_i \mid \theta))^2}{2\sigma^2} \right) = constant + \frac{1}{2\sigma^2} \sum_{i=1}^{n} -(y_i - f(x_i \mid \theta))^2$$

Thus, maximizing the probability of our data from some underlying distribution is the same as maximizing $\sum_{i=1}^{n} -(y_i - f(x_i;\theta))^2$ or equivalently minimizing the sum of the squared errors. From this, we see the L_2-norm $||Ax - b||_2$ is a powerful measure of the error assuming zero-mean, Gaussian distributed noise.

In our linear system $Ax = b$ we have a series of linear equations of the form:

$$a_{i1}x_1 + a_{i2}x_2 + ... + a_{in}x_n = b_i \qquad a_{ij} \in \mathbb{R}$$

While the L_2 norm provides great error properties, it incentivizes the use of every parameter possible. This can be an issue because it is not often the case that the exact set of predictors is known coming into a problem. Sometimes a predictor is used that is actually useless or redundant. Regardless, given an extra knob to turn the L_2 norm might be able use this predictor to lower the error, even if it only gives the predictor a value slightly away from zero. This can muddy the interpretation of the model by overestimating the number of seemingly relevant parameters. To lower the number of parameters used (promote sparsity) we will have to add regularizaton.

$$\min_{x \in \mathbb{R}^n} ||Ax - b||_2 + \lambda g(x)$$

Now instead of simply minimizing the error, we are penalizing undesirable qualities in the solution with the $g(x)$ function. If we want to promote sparsity, we can choose $g(x) = ||x||_1$, the L_1 norm. By penalizing non-zero predictors by a factor of λ, we encourage the solution to have as few non-zero terms as possible. As a result, these nonzero predictors are often interpretable as key features of the system of interest. In actuality, this is just a proxy for the L_0 norm, but optimization using L_0 is much more difficult so the L_1 norm is often used as a reasonable approximation.

3 Algorithm Implementation and Development

The MNIST data is loaded by the load_MNIST_file function. This flattens each image and stores it as a row in our A matrix. For each label $i \in \{0, 1, .., .9\}$, we store the label as a row in B using one hot encoding

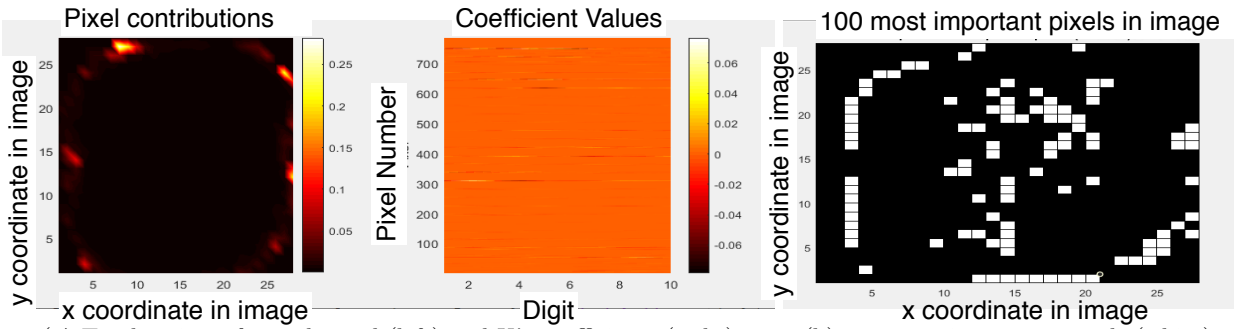
(as the vector $[0, 0, ..., 0, 1, 0, ...0]$ where only the ith entry is nonzero, the last entry in the case of $i = 0$). This generates a labeling system where each label is orthogonal rather than just using the digit as the label where there is a lot more room for ambiguity and it is less obvious how the system weights produce them. In the end, we have a system $AX = B$ where all are matrices. Under this arrangement, we can see that $A_{row1}X = B_{row1} \rightarrow (image_1)X = label_1$. In this framework, X is naturally interpretable: X_{ij} represents how important pixel i is in determining the image represents digit j. To determine to contribution of each pixel, we summed across the absolute value of each entry in the corresponding row in X. This prevented negative values from possibly interfering with the value associated to each pixel (Appendix B: Main code lines 20 - 32).

We down-sampled from 60000 to 10000 training samples because some of the sparsity promoting techniques had difficulty running on the full dataset. From here, we ran several $AX = B$ solvers: MATLAB's backlash (QR decomposition), pseudoinverse, and Least-Squares with L_1 regularization. For the L_1 regularization, we used the CVX optimization package and determined the optimal λ by slowly raising the λ value until there was a significant increase in the error rate. We choose $\lambda = 10$ as it consistently led to low error and a seemingly meaningful choice of important pixels (Appendix B: Main code lines 70 - end). After solving for X using one of these techniques/conditions, we checked the performance of X on the testing dataset using the function predict_labels. For each test image, it vectorizes it and multiplies it by X. Then, it will guess the digit whose entry is the highest in resulting product. From there, it calculates the error rate by comparing these responses against the true labels. Lastly, it plots some information about the coefficients in X such as where the nonzero entries lie and which pixels have the largest coefficients. From this information, we could determine which pixels were most important in classification, calculating the contribution/value of each pixel as discussed earlier. We used two metrics to determine importance: absolute value of a pixel's contribution and the absolute value distance from the mean. From there, we simply chose the top 100 pixels that had the highest value according to this metric. We found the latter worked best, more often choosing central pixels likely to be part of the digit rather than the pixels that are consistently background, so we ultimately used that. Using only the top 100 pixels determined to be most important, we redid the classification after setting all rows in X not associated to one of these 100 pixels to zero. From there, we analyzed the results and calculated the error rate (Appendix B: Predict Labels lines 80-105).

Lastly, rather than training our predictors on the entire dataset, we selected only the images corresponding to a specific digit. From here, we repeated the whole process to analyze the performance under these new conditions. In particular, we were interested in how this change would effect the subset of pixels determined most important (Appendix B: Main Program lines 39-55).

4 Computational Results

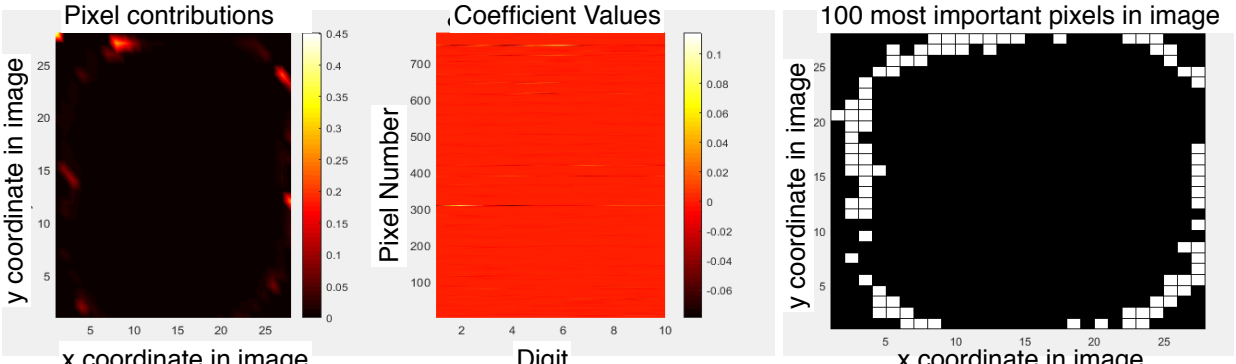
Applying this procedure to the Moore-Penrose pseudoinverse, a true least squares solution, we get a fairly dense representation of the system. As we see in **Figure 1a** there appears to be only a few pixels and coefficients in X that take on significant values. However, on closer inspection all of these 784 of these pixels and 7840 of these coefficients in X take on nonzero values, many incredibly smalls ($\sim 10^{-10}$). As shown in **Figure 1b**, some of the most important pixels lie oddly on the exterior of the image where the digit likely does not lie. Regardless, the representation generated is highly accurate with an error rate of only 16% on 10000 images in the testing data set. However, when only the top 100 most important pixels were used as classifiers, this error rate jumped to 80%, suggesting these exterior pixels are likely an artifact of the least squares procedure and do not help store a representation of digitspace.



(a) Total activity for each pixel (left) and X's coefficients (right)

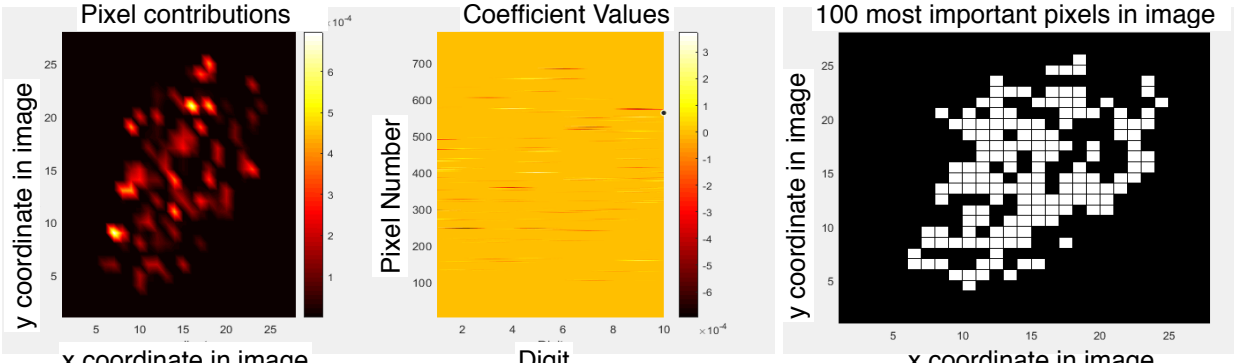
(b) 100 most important pixels (white)

Figure 1: Mapping from pixel space to digit space using Moore-Penrose pseudoinverse



(a) Total activity for each pixel (left) and X's coefficients (right) (b) 100 most important pixels (white)

Figure 2: Mapping from pixel space to digit space using MATLAB's backslash (QR decomposition)



(a) Total activity for each pixel (left) and X's coefficients (right) (b) 100 most important pixels (white)

Figure 3: Mapping from pixel space to digit space using an L_1 regularizer on X

Applying this procedure to MATLAB's backslash which relies on a QR decomposition (and thus is not truely a least squares solver) we get a sparser representation of the system. As we see in **Figure 2a**, there appears to be only a few pixels and coefficients in X that take on significant values, but a closer inspection shows that 591/784 of the pixels and 5910/7840 of the coefficients in X take on nonzero values, many of which are incredibly small ($\sim 10^{-10}$). As shown in **Figure 1b**, almost all of the most important pixels lie on the exterior of the image where the digit likely does not lie. Regardless, the representation generated is highly accurate with an error rate of only 16% on 10000 images in the testing data set. Similar to the pseudoinverse however, when only the top 100 most important pixels were used as classifiers this error rate jumped to 70%, suggesting these exterior pixels likely do not help store a representation of digitspace.

Lastly, we applied this procedure to a least squares solution regularized by an $L1$ norm on X:

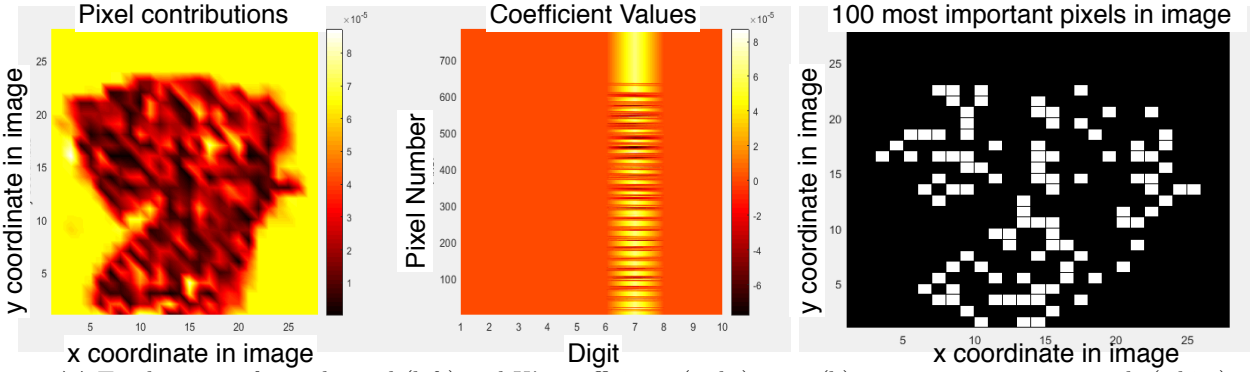
$$||AX - B||_{fro} + \lambda||X||$$

.

With a $\lambda = 10$ we found a much sparser representation of pixelspace was generated. As we can see in **Figure 3a** there are significantly more high valued pixels using the L_1 regularizer than in the previous two techniques. Furthermore, as we see in **Figure 3b** the 100 most important pixels are much more reasonable, capturing a space where the digit is likely to lie. Additionally, on closer inspection we see only 412/784 pixels and 4120/7840 coefficients in X are nonzero, much fewer than in the previous two cases. While this technique performed slightly worse in initial classification, achieving around 20% accuracy, its top 100 pixels were much better classifiers than before. With them alone, a 55% accuracy was achieved. This suggests the L_1 norm is capturing the fundamental structure of the underlying pixelspace better than either of the two previous techniques.

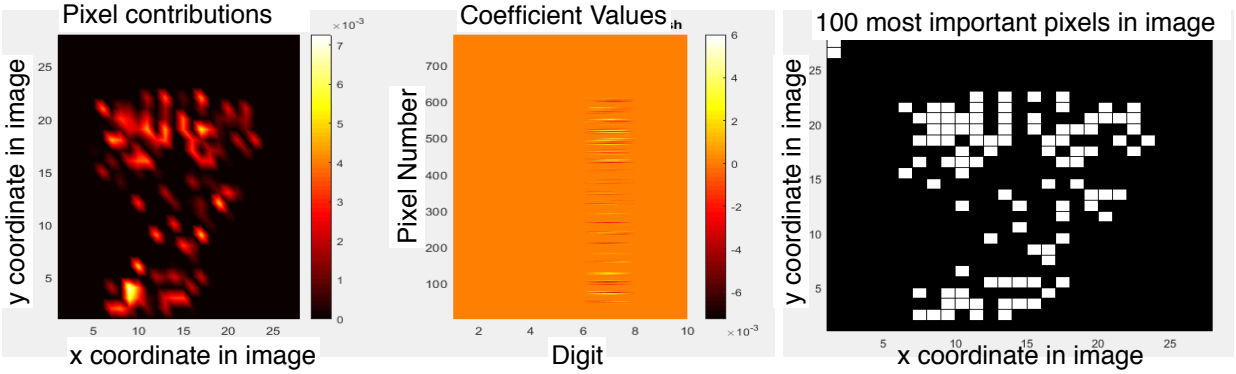
We can repeat this procedure training and testing only on a single digit (seven in this case). **Figures 4,5,** and **6** below show the results of this. As is clear from **4b,5b,6b**, we can see all techniques generate at least a rough representation of the appearance of a seven. Interestingly, the most important pixels extracted from using backslash and pseudoinverse are primarily relevant pixels rather the exterior pixels we observed in **Figures 1,2**. We noticed that each technique had a consistently low error rate, typically less than a 10% error, across all ten possible digits. This is likely because there is a lot less variation when training over just a single digit and the coefficients in X can be more finely tuned towards that digit. Furthermore, in the right images of **Figures**

4a, 5a, 6a we can see clear localization of the coefficients in X: almost all of the coefficients not associated to the digit 7 are near zero in all three methods. As expected, the pixel localization is much more refined in the L_1 norm case, as visible in the left image of **4a,5a,6a**, than in the other two cases. On closer inspection, we saw that 784/784 of the pixels and 784/7840 of the coefficients in X were nonzero in the pseudoinverse case, 116/784 and 116/7840 using MATLAB's backslash, and 432/784 and 432/7640 in the L_1 regularization case. Interestingly, MATLAB's backslash generated a more sparse representation in this case than the L_1 norm. However, on closer inspection we can again see many nonzero values in this case were near machine precision, 10^{-16}, and this could have interfered with the counting of the number of nonzero entries. Lastly, we redid the classification using only the most important pixels and saw roughly similar results in all three cases: the error rate increased to around 40%, but that is still an improvement on the previous case where all digits were used. A further exploration of the most important pixels for each digit can be found in Appendix C



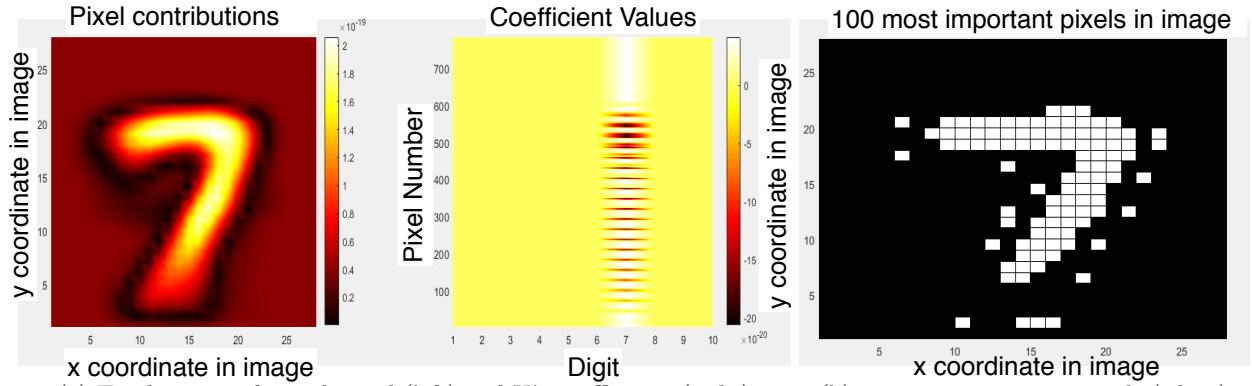
(a) Total activity for each pixel (left) and X's coefficients (right) (b) 100 most important pixels (white)

Figure 4: Mapping from pixel space to digit space using Moore-Penrose pseudoinverse trained on only 7's.



(a) Total activity for each pixel (left) and X's coefficients (right) (b) 100 most important pixels (white)

Figure 5: Mapping from pixelspace to digitspace using MATLAB's backslash trained on only 7's.



(a) Total activity for each pixel (left) and X's coefficients (right) (b) 100 most important pixels (white)

Figure 6: Mapping from pixel space to digit space using L_1 norm regularizer trained on only 7's.

5 Summary and Conclusions

In summary, we were able to show that promoting sparsity through L_1 norm regularization can help generate more meaningful representations of the system we are working with. In this case, it helped us generate a better low-rank approximation of the system by picking out more relevant pixels than either MATLAB's backslash or pseudoinverse. As a result of this, we could generate relatively accurate predictions of the digit using only 100 pixels in the image, just over an eighth of the total image. We showed that with an appropriately chosen regularization parameter λ we can introduce this idea of sparsity with little cost to the accuracy of the classifier. However, one downside of the L_1 norm that became apparent during this project was that it is much slower to utilize due to slower optimization techniques. For this reason, we had to downsize our training set to a sixth of its total size. While the L_1 regularization outperformed on this smaller dataset, MATLAB's backslash and pseudoinverse could easily handle the full 60,000 training images. This larger sample sized allowed them to get much higher accurracies than L_1 regularization could on the smaller dataset. This slowdown is likely caused by the optimization package (CVX) we used which is meant to be used in a more general setting. Using more specific L_1 regularization optimizations may improve this performance. Overall however, L_1 regularization provided a much more interpretable model due to its sparsity and could offer key insights if its results were used alongside another technique.

6 Appendix A

Below is a brief summary of the MATLAB functions I used during this project and their functions.

pinv(A) : Calculates the Moore-Penrose pseudoinverse of the matrix A. Using this to solve the system $Ax = b$ is equivalent to finding the least squares solution x.

swap_bytes(): Given a series of bytes, this function swaps between two possible formattings: Big-endian and little-endian values. This is used to reformat the MNIST data files into the format that MATLAB uses.

cvx_begin and cvx_end: CVX is an optimization toolbox generated for MATLAB. These commands specify the start/end of a call to CVX and the parameters in between specify the optimization parameters, objective function, and constraints.

pcolor: Provides a image representation of a matrix, coloring each entry in a grid based on the value in the corresponding entry in the matrix.

load_MNIST_file: loads in images and labels from MNIST files and puts them in the format described in the implementation section.

predict_labels: Given a mapping X and a set of test images/labels, it calculates the error rate of the mapping and plots some information about the coefficients in the mapping X such as the location of nonzero values and the relative magnitude of the coefficients.

7 Appendix B

7.1 Main Code

```
% Zachary McNulty
% AMATH 563: Inferring Structure of Complex Systems
% HW 1

% NOTES:

% invert = 0, mean_subtract = 0 works well for choosing the best pixels from the individual
% digits using "sort by distance from mean" metric

%% Part 0: Loading the MNIST Data
clear all; close all; clc;

invert = 1; % invert image pixel colors or not; 0 = white digit on black background, 1=black on white
mean_subtract = 1; % mean subtract from images; 1 = yes, 0 = no
num_pixels = 100; % number of pixels to use in low rank approximation

% load_MNIST_file is a helper function that helps deal with the formatting
% of these data files. I take the tranpose to make images/labels in the
% rows
A_train_images = load_MNIST_file('input_files/train-images-idx3-ubyte','image', invert, mean_subtract).';
B_train_labels = load_MNIST_file('input_files/train-labels-idx1-ubyte', 'label', invert, 0).';
A_test_images = load_MNIST_file('input_files/t10k-images-idx3-ubyte','image', invert, mean_subtract).';
B_test_labels = load_MNIST_file('input_files/t10k-labels-idx1-ubyte', 'label', invert, 0).';

% smaller datasets for testing
train_size = 10000; % max is 60000
test_size = 10000; % max is 10000
A_train_images = A_train_images(1:train_size, :);
B_train_labels = B_train_labels(1:train_size, :);
A_test_images = A_test_images(1:test_size, :);
B_test_labels = B_test_labels(1:test_size, :);


% Part 4: Analysis on single digits at a time

% skip running this section or set digit_of_interest < 0 if you want to
% use all digits in training/test set.

digit_of_interest = 9;

if digit_of_interest >= 0
% accounts for indexing of labels
    if digit_of_interest == 0
       digit_of_interest = 10;
    end

    train_rows = B_train_labels(:, digit_of_interest) == 1;
    test_rows = B_test_labels(:, digit_of_interest) == 1;

    A_train_images = A_train_images(train_rows, :);
    B_train_labels = B_train_labels(train_rows, :);
```

```
52      A_test_images = A_test_images(test_rows, :);
53      B_test_labels = B_test_labels(test_rows, :);
54  end
55
56  % test code for ensuring all images are from digit of interest
57  % for k = 1:100
58  %     imshow(uint8(reshape(A_train_images(k,:), [28,28]).'));
59  % end
60
61
62
63  %% Part 1: Mapping between images and digits.
64  close all; clc; clearvars -except A_test_images A_train_images B_test_labels
        B_train_labels num_pixels
65
66  % Standard MATLAB backslash: AX = b
67
68  % X is our classifier. Note that as each row of A is an image and each row
69  % of B its corresponding label, we have (image i) X = (label i) so X tells
70  % us how to weight weight each pixel. Furthermore, (image i) (colum X_j) =
71  % B_ij --> tells us how much image i resembles digit j
72  X_train = A_train_images \ B_train_labels;
73
74  [predicted_labels_backslash, error_backslash] = ...
75      predict_labels(X_train, A_test_images, B_test_labels, 'backslash',
            num_pixels);
76  error_backslash
77
78  %% matrix L2 Norm (psuedoinverse)
79  close all; clc; clearvars -except A_test_images A_train_images B_test_labels
        B_train_labels num_pixels
80
81  X = pinv(A_train_images)*B_train_labels;
82  [predicted_labels_2norm, error_2norm] = ...
83      predict_labels(X, A_test_images, B_test_labels, '2 norm', num_pixels);
84  error_2norm
85
86
87  %% Part 2: Promoting Sparsity & Dimensionality Reduction
88
89  % % Lasso
90  % close all; clc; clearvars -except A_test_images A_train_images B_test_labels
        B_train_labels num_pixels
91  %
92  % % lasso on each column separately? Column i of X will generate column i in
93  % % B given A
94  % X = zeros(28^2,10);
95  %
96  % for col = 1:10
97  % %     X(:, col) = lasso(A_train_images, B_train_labels(:, col), 'Lambda',
        lambda);
98  %     [temp, fitInfo] = lasso(A_train_images, B_train_labels(:, col));
99  %     X(:, col) = temp(:, end);
100 % end
101 %
102 % [predicted_labels_lasso, error_lasso] = ...
103 %     predict_labels(X, A_test_images, B_test_labels, 'lasso', num_pixels);
104 % error_lasso
105
106
107
```

```
%% L1 norm across all elements
close all; clc; clearvars -except A_test_images A_train_images B_test_labels
    B_train_labels num_pixels
clear cvx_problem;

% Frobenius norm is the sum of the squared entries in the matrix vs the
% matrix 2-norm (i.e. norm(X, 2)) which gives the max singular value. This
% just tries to minimize L1 norm of ALL coefficients, not necessarily
% pixels. How can we do it over pixels?


lambda = 10;
m = size(A_train_images, 2);
n = size(B_train_labels, 2);
cvx_begin
    variable X(m,n)
    minimize norm(A_train_images*X - B_train_labels, 'fro') + lambda*sum(sum(abs
        (X)))
%     minimize norm(A_train_images*X - B_train_labels, 2) + lambda*sum(sum(abs(X
    )))
cvx_end

[predicted_labels_1norm, error_1norm] = ...
    predict_labels(X, A_test_images, B_test_labels, '1 norm', num_pixels);
error_1norm
```

7.2 Load MNIST File

```
function result = load_MNIST_file(file, file_type, invert, mean_subtract)

% https://stackoverflow.com/questions/24127896/reading-mnist-image-database-binary-file-in-matlab

% pixels are organized row-wise (when we read them in we get the pixels in
% row 1 from left to right, pixels in row 2 from left to right, etc...)
% Thus the first 28 pixels in each column of A are the first row.


% file_type = 'image' or 'label' tells whether these files correspond to
% images or labels
% file = path to file from current working directory.
% invert = 1 if you want to invert the pixel colors (i.e. to be white digit
% on black background)

%//Open file
fid = fopen(file, 'r');

%//Read in magic number
next = fread(fid, 1, 'uint32');
magicNumber = swapbytes(uint32(next));

%//Read in total number of images or labels
next = fread(fid, 1, 'uint32');
numElements = swapbytes(uint32(next));

if file_type == 'image'
    %//Read in number of rows
    next = fread(fid, 1, 'uint32');
    numRows = swapbytes(uint32(next));

    %//Read in number of columns
    next = fread(fid, 1, 'uint32');
    numCols = swapbytes(uint32(next));

    % stores a column for each image
    result = zeros(28^2,numElements);
else
    % stores a column for each image label.
    % See the homework specification for the format of these labels
    result = zeros(10, numElements);
    numRows = 1;
    numCols = 1;
end

for k = 1 : numElements
    %//Read in numRows*numCols pixels at a time
    next = fread(fid, numRows*numCols, 'uint8');
    if file_type == 'image'

        if invert
            result(:, k) = imcomplement(uint8(next)); % take image complement
        else
            result(:, k) = uint8(next);
        end
%         imshow(uint8(reshape(result(:,k), numCols, numRows)).');
%         pause(1)
    else
```

```
59          next = uint8(next);
60          if next == 0
61              next = 10;
62          end
63          result(next, k) = 1;
64      end
65  end
66
67  %//Close the file
68  fclose(fid);
69
70  if mean_subtract && file_type == 'image'
71      result = result - mean(mean(result));
72  end
73
74  end
```

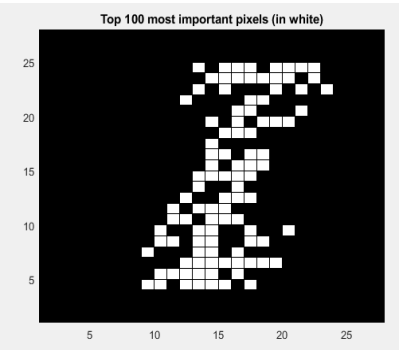
7.3 Predict Labels

```
function [predicted_labels, error_rate] = predict_labels(X_train, A_test_images,
    B_test_labels, method_name, num_pixels)

% X_train are the parameters trained using the given model
% A_test_images are the images in the testing data set and B_test_labels
% are their corresponding labels.
% method name is the name of the AX = B solver being used; only effects
% plot labeling.
% num_pixels = number of pixels to use in low rank approximation


new_labels = A_test_images * X_train;
predicted_labels = zeros(size(A_test_images,1),10);
for k = 1:size(predicted_labels, 1)
   index = find(new_labels(k,:) == max(new_labels(k,:)),1);
   predicted_labels(k, index) = 1;
%    predicted_labels(k, :) = predicted_labels(k, :) == max(predicted_labels(k,
    :));
end

% need the 2 in the bottom as if a label is incorrect both the true
%% position and the misclassified position will have the wrong number
% figure (1)
% subplot(121)
% error_rate = sum(sum(predicted_labels ~= B_test_labels)) / (2*size(
    predicted_labels, 1));
% pcolor(flipud(X_train)), shading interp;
% colormap('hot')
% ylabel('Pixel')
% xlabel('Digit')
% title(method_name)
% colorbar;
%
% subplot(122)
% plot(reshape(X_train, [1, numel(X_train)]), 'r.', 'markersize', 5)


% tranpose is because reshape fills the columns of the new matrix first,
% but X is ordered by rows --> X = [image_row1.'; image_row2 .'; ...]
figure(2)
subplot(121)
pixel_preferences = reshape(sum(abs(X_train), 2), [28,28]).';
pcolor(flipud(pixel_preferences)), shading interp;
colormap('hot')
xlabel('x coordinate')
ylabel('y coordinate')
title(strcat("Pixel preferences for ", method_name))
colorbar;

subplot(122)
error_rate = sum(sum(predicted_labels ~= B_test_labels)) / (2*size(
    predicted_labels, 1));
pcolor(flipud(X_train)), shading interp;
colormap('hot')
ylabel('Pixel')
xlabel('Digit')
title(strcat("Coefficient values for ",method_name))
colorbar;
% plot(reshape(pixel_preferences, [1, numel(pixel_preferences)]), 'r.', '
```

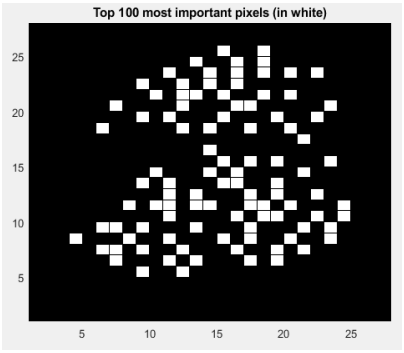
```
    markersize', 5)
% ylabel('Coefficient Value')
% xlabel('Pixel Number')


figure(3)
subplot(121)
pcolor(flipud(pixel_preferences ~= 0))
nonzero_pixels = sum(sum(pixel_preferences ~= 0))
colormap(gray(2))
title('Nonzero pixel values (in white)')
xlabel('x coordinate')
ylabel('y coordinate')


subplot(122)
pcolor(flipud(X_train ~= 0)), shading interp;
nonzero_coefficients = sum(sum(X_train ~= 0))
colormap(gray(2))
title('Nonzero coefficient values (in white)')
ylabel('Pixel')
xlabel('Digit')


% Part 3: Find the most important pixels in this image
figure(4)

% sort by value
 [sorted, I] = sort(abs(reshape(pixel_preferences, [1, 28^2])), 'descend');

% sort by distance from mean
% [sorted, I] = sort(abs(reshape(pixel_preferences, [1, 28^2]) - mean(mean(
    pixel_preferences))), 'descend');
best_pixels = zeros(1, 28^2);
best_pixels(I(1:num_pixels)) = 1;

pcolor(flipud(reshape(best_pixels, [28,28])));
colormap(gray(2));
title("Top 100 most important pixels (in white)")

X_lowrank = X_train .* (best_pixels.'); % zero out all but the most important
    pixels

new_labels_lowrank = A_test_images * X_lowrank;
predicted_labels_lowrank = zeros(size(A_test_images,1),10);
for k = 1:size(predicted_labels, 1)
    index = find(new_labels_lowrank(k,:) == max(new_labels_lowrank(k,:)),1);
    predicted_labels_lowrank(k, index) = 1;
%    predicted_labels(k, :) = predicted_labels(k, :) == max(predicted_labels(k,
    :));
end

error_rate_lowrank = sum(sum(predicted_labels_lowrank ~= B_test_labels)) / (2*
    size(predicted_labels_lowrank, 1))
end
```

8 Appendix C

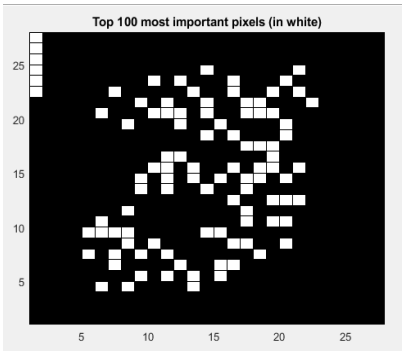
In this section, only MATLAB's backslash was used to generate the subset of pixels deemed most important for each digit. This was because the L_1 regularization was too slow to run repeatedly on a large dataset and from **Figures 5,6** we saw that both generate roughly similar representations of these pixels. The digit 7 was excluded from this analysis as it was already shown in the results section above. Interestingly, we see that the exterior pixels are consistently ignored across all ten digits, contrary to what we found when all digits were trained together. It is possible that the larger sample size used when the digits were trained together influenced these features. Regardless, for each digit there is a clear pattern in the most important digits. All seem to capture the general shape of their respective digits. Interestingly, in all cases the interior of the digits seem to be used as a reference point rather than the borders or some other kind of edge detection.



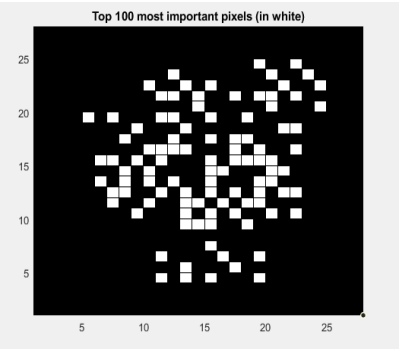
(a) one (1)



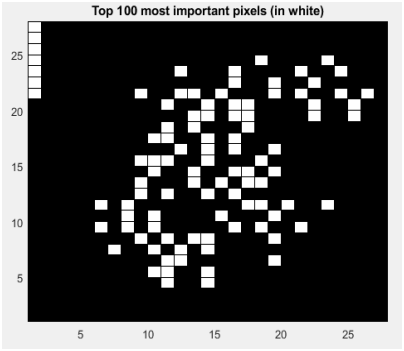
(b) two (2)



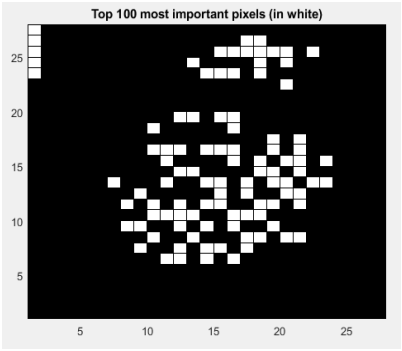
(c) three (3)



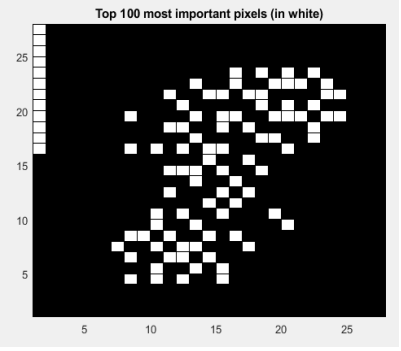
(a) four (4)



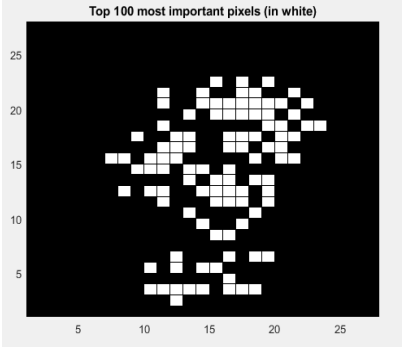
(b) five (5)



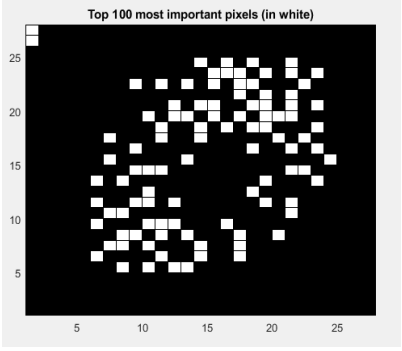
(c) six (6)



(a) eight (8)



(b) nine (9)



(c) zero (0)

Figure 9: 100 most important pixels from MATLAB's backslash (QR decomposition) on training sets including only a single digit. Above, the results are shown for all the digits. Notice that exterior pixels are consistently ignored.