Linear Fitting to Classify Hand Drawn Digits

Ashley Batchelor

Abstract

I used five different methods of linear fitting to map images of hand drawn digits: LASSO, robustfit (least squares), QR decomposition, Moore-Penrose pseudoinverse, and ridge regression. I compared the characteristics of each fit. For each method, I determined the pixels which represented 90% of the total pixel weightings (summed over each image), and created a sparse fit. I then determined the pixels which represented 90% of the total pixel weightings for each individual digit and created as sparse fit.

Sec. I. Introduction and Overview

The hand drawn digits came from the MNIST training data set which includes 60,000 labeled images of hand drawn digits 0 through 9.[1]

Each of the digits may be represented by a 10-component column vector $\boldsymbol{y_i}$, corresponding to the numerals 1, 2, 3, 4, 5, 6, 7, 8, 9, 0.

$$\boldsymbol{y_1} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \boldsymbol{y_2} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \ldots \boldsymbol{y_0} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

The labels may be represented as a matrix $\boldsymbol{B}$, where the columns are the respective $\boldsymbol{y_i}$:

$$\boldsymbol{B} = [\boldsymbol{y_1}\ \boldsymbol{y_2} \ldots \boldsymbol{y_0}]$$

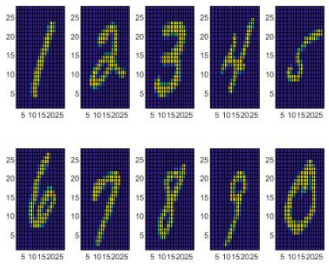

Figure 1 - Examples of MNIST digit images.

The images of the hand drawn digits may be represented as vectors $\boldsymbol{a_i}$ formed from successive rows of pixel values. The set of images may be represented as a matrix $\boldsymbol{A}$, where the columns are the respective $\boldsymbol{a_i}$:

$$\boldsymbol{A} = [\boldsymbol{a_1}\ \boldsymbol{a_2} \ldots \boldsymbol{a_n}]$$

[1] http://yann.lecun.com/exdb/mnist/

The images map by a matrix $\boldsymbol{X}$ to the label matrix $\boldsymbol{B}$, by a simple linear matrix multiplication where:

$$\boldsymbol{AX} - \boldsymbol{B} = \boldsymbol{0}$$

The MNST training data includes 60,000 28x28 pixel grayscale images of hand written digits (784 total pixels). Therefore, the linear relation above may be understood as an over-determined linear system, which may be solved by many different linear regression algorithms for $\boldsymbol{X}$ to determine the mapping.

Sec. II. Theoretical Background

A least squares (i.e. L_2 norm) fit minimizes an error E according to:

$$\min E = \text{argmin}_{\boldsymbol{X}} \|\boldsymbol{AX} - \boldsymbol{B}\|_2$$

In general, a linear fit for an over-determined system may be chosen to minimize the error E according to:

$$\min E = \text{argmin}_{\boldsymbol{X}} \|\boldsymbol{AX} - \boldsymbol{B}\|_2 + \lambda_1 \|\boldsymbol{X}\|_1 + \lambda_2 \|\boldsymbol{X}\|_2$$

The λ_1 and λ_2 terms respectively specify the penalization of the L_1 and L_2 norms. A LASSO algorithm solves this for $\lambda_1 > 0$ and $\lambda_2 = 0$. A Moore Penrose pseudo-inverse algorithm solves this for $\lambda_1 = \lambda_2 = 0$. A ridge regression algorithm solves this for $\lambda_1 = 0$ and $\lambda_2 > 0$.[2]

The matlab robustfit function fits data using a weighted least squares algorithm. The matlab backslash "\" function fits data using a QR decomposition algorithm.

Sec. III. Algorithm Implementation and Development

In order to find the set of pixels that represent 90% of the weighting, I summed each row of $\boldsymbol{X}$, used a maxk command to find a set of k maximum row sums, and then varied k until I found a sum of rows that represented 90% of the weighting. I then made a mask corresponding to $\boldsymbol{X}$ and populated it with zeros and inserted ones corresponding to the rows in the set of k maximum row sums. I then multiplied this mask by $\boldsymbol{X}$ to determine a sparse matrix $\boldsymbol{X_{sparse}}$.

I then found the product $\boldsymbol{AX_{sparse}} = \boldsymbol{B_{sparse}}$ and compared the result to the matrix $\boldsymbol{B}$. To compare the result, I rounded the values of $\boldsymbol{B_{sparse}}$ to integers and calculated the difference $\boldsymbol{B_{diff}} = \boldsymbol{B_{sparse}} - \boldsymbol{B}$. I then counted the number of zero values in the $\boldsymbol{B_{diff}}$ matrix to determine a percentage accuracy.

In MATLAB, the LASSO, ridges, and robustfit algorithms solve for a vector value (as opposed to a matrix) and require a vector value for $\boldsymbol{B}$. Therefore, in order to solve for the 784x10 matrix $\boldsymbol{X}$, I iterated over 10 loops, using a column from $\boldsymbol{B}$ each time to solve for a column of $\boldsymbol{X}$. I constructed the matrix $\boldsymbol{X}$ from those solutions.

I did a similar computation for each individual digit. I used a maxk command on each individual row of $\boldsymbol{X}$ and summed the absolute values of the top k weightings from each pixel, and then varied k until I found a sum of pixels that represented 90% of the weighting. I then made a mask corresponding to each row of $\boldsymbol{X}$ and populated it with zeros and inserted ones corresponding to the pixels in the set of k maximum row sums. I then multiplied this mask by $\boldsymbol{X}$ to determine a sparse matrix $\boldsymbol{X_{sparse}}$.

[2] Class notes

For the QR decomposition, I simply computed a backslash command in MATLAB. For the robustfit algorithm I set const='off' to avoid adding a column of 1s to $\boldsymbol{B}$, and thus to get the right size of matrix for $\boldsymbol{X}$.

For the LASSO algorithm, I used $\lambda_1 = 0.1$. A value of 0.01 gave a solution with the weightings heavily spread out over the pixels of each digit, whereas a value of 0.05 gave a solution with the weightings very strongly concentrated in just two or three pixels, which may introduce alignment sensitivity of the digits.

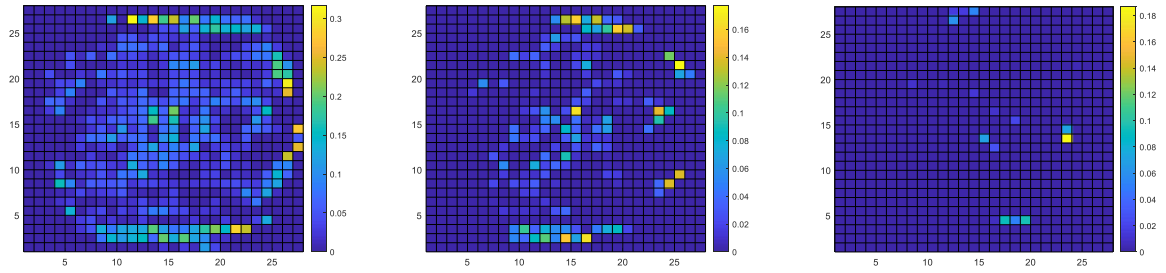

Figure 2 Summed rows of weightings for LASSO fit with $\lambda_1 = 0.01, 0.05, 0.1$

For the Moore-Penrose pseudo-inverse, I used the default parameters.

For the ridge regression algorithm, I used $\lambda_2 = 0.05$. I tried values spanning from 0.0001 to 1, all of which yielded 90% of the weighting within the same number of rows. For comparison with the LASSO algorithm, I chose the same coefficient.

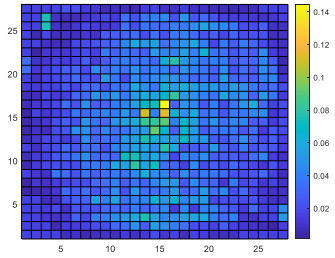

Figure 3 Summed rows of weightings for ridge regression fit with $\lambda_2 = 0.05$

In order to measure the accuracy of each fit, I calculated the predicted $\boldsymbol{B}$ matrix and for each column of the matrix, i.e. the data label, I found the maximum value and checked if it matched the index of the 1 in the corresponding column in the original $\boldsymbol{B}$ matrix to determine if this was an accurate predicton for the digit. For each method, I used the total count of accurate predictions to compute a "percentage true." I also calculated a mean square accuracy, but this is not the best metric. If a model for $\boldsymbol{X}$ simply predicts a $\boldsymbol{B}$ matrix which is entirely zeroes, then it would still match 90% of the values. I looked at how well it actually predicts which label goes with the corresponding data.

Sec. IV. Computational Results

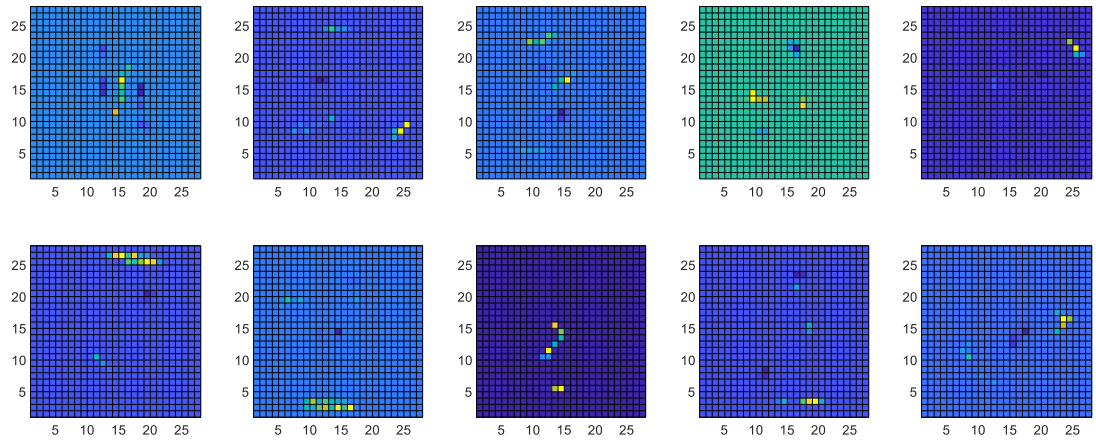

Figure 4 Individual weightings for LASSO fit. Dark blue – negative, yellow – positive.

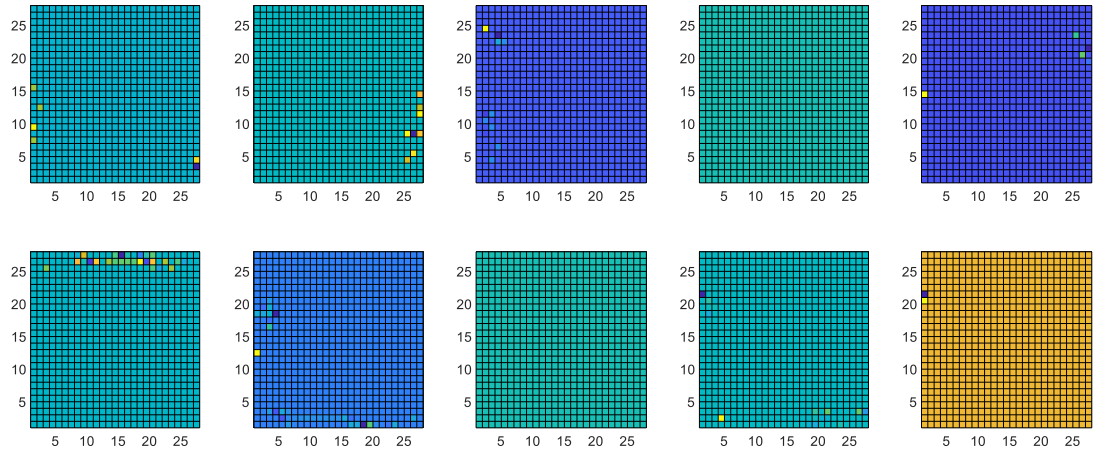

Figure 5 Individual weightings for robustfit. Dark blue – negative, yellow – positive.

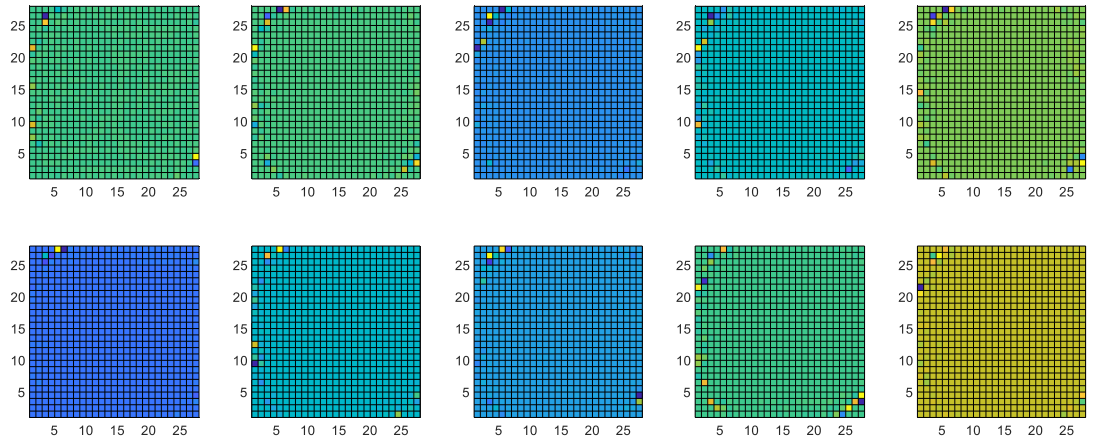

Figure 6 Individual weightings for QR decomposition fit. Dark blue – negative, yellow – positive.

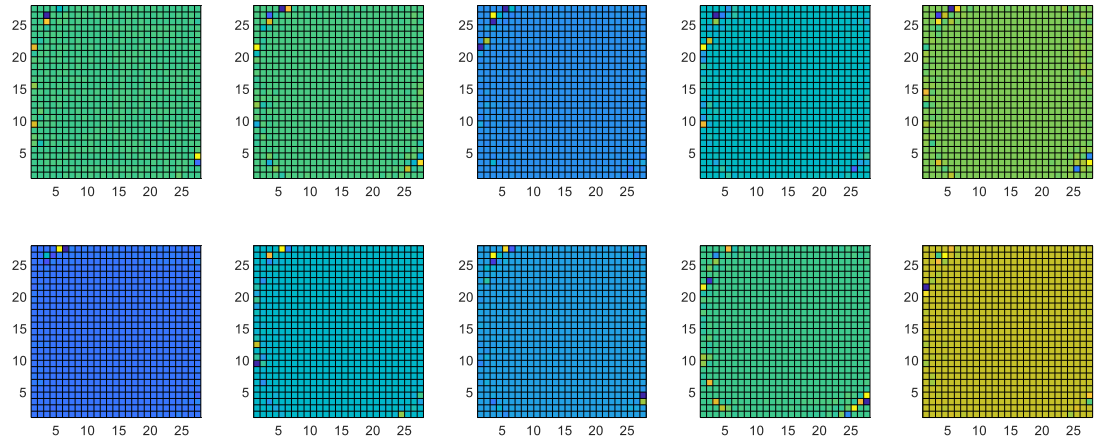

Figure 7 Individual weightings for pinv fit. Dark blue – negative, yellow – positive.

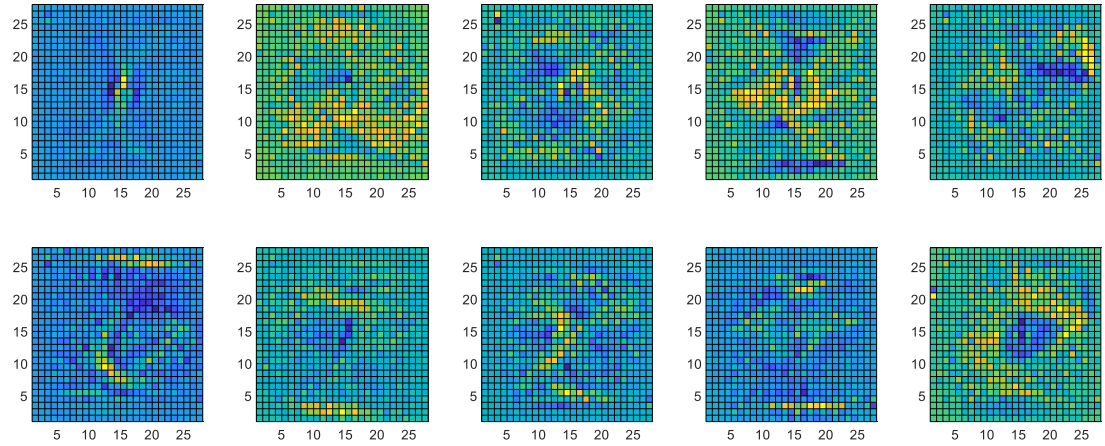

Figure 6 Individual weightings for ridge regression fit. Dark blue – negative, yellow – positive.

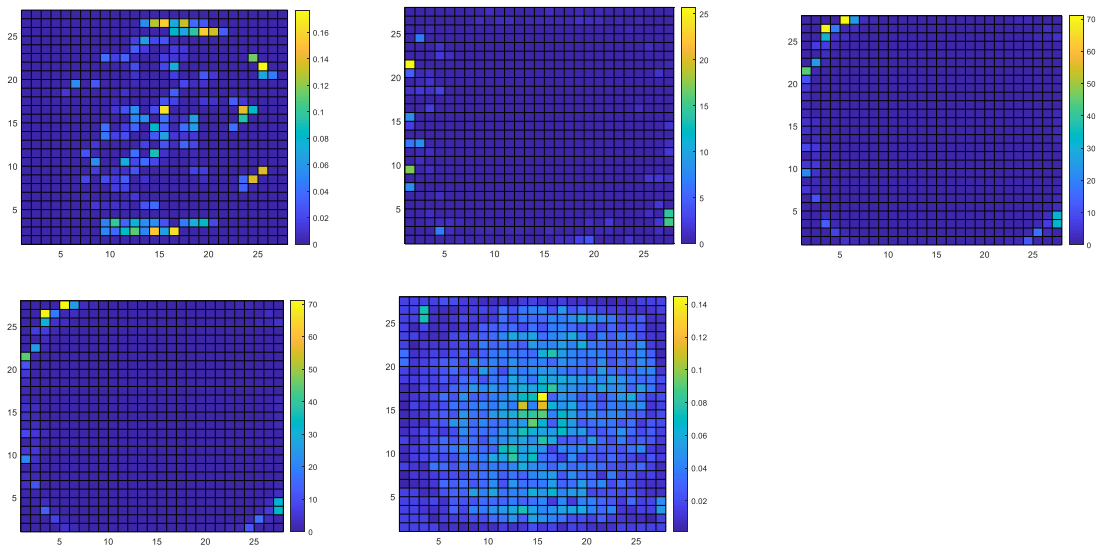

Figure 7 Summed rows of weightings for LASSO, robustfit, QR decomposition, pinv, and ridge regression.

	Composite Mask for Sparsity				
Method	Pixels for 90% Weight	MSE (Sparse)	MSE (Full)	Percent True (Sparse)	Percent True (Full)
LASSO	950	0.0735	0.0721	61.72%	65.06%
robustfit	610	0.0953	0.0901	31.69%	33.50%
QR	2010	0.0957	0.0391	30.85%	85.19%
pinv	2050	0.0957	0.0391	28.92%	85.19%
ridges	5690	0.0724	0.0724	77.95%	78.00%
	Individual Pixel Mask for Sparsity				
Method	Pixels for 90% Weight	MSE (Sparse)	MSE (Full)	Percent True (Sparse)	Percent True (Full)
LASSO	123	0.0736	0.0721	62.39%	65.06%
robustfit	87	0.0934	0.0901	31.61%	33.50%
QR	1531	0.0782	0.0391	46.83%	85.19%
pinv	1568	0.0778	0.0391	45.27%	85.19%
ridges	4638	0.0723	0.0724	77.70%	78.00%

Figure 10 Comparison of accuracies and sparsity using a mask based on summed rows and a mask based on individual pixels.

Sec. V. Summary and Conclusions

QR Decomposition and pinv gave the highest accuracy (percentage true) results for the full fit, but gave very poor accuracy for the sparse fits. Robustfit gave the best sparsity to produce comparable accuracy, especially for the individual digit pixel masks. Ridges required by far the most pixels for 90% of the weightings. Robustfit only required 87 pixels, whereas ridges required 4638. However, ridges gave much better accuracy. Using individual digit pixel masks generally gave better accuracy.

Appendix A MATLAB functions used and brief implementation explanations

A\B – This applies a QR decomposition fit.

lasso() – This applies a lasso fit, with specified parameters.

loadMNISTImages() and loadMNISTLabels() – These are special utilities for loading the images and labels of the MNIST data sets into MATLAB.[3]

maxk() – This returns the maximum k values of an array.

pinv() – This applies a Moore-Penrose pseudoinverse fit, with specified parameters.

ridge() – This applies a ridge regression fit, with specified parameters.

robustfit() - This applies a least squares fit.

Appendix B MATLAB codes

Code for fittings that accept matrix input

```
clear all; close all; clc;
%http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset
images = loadMNISTImages('train-images.idx3-ubyte');
labels = loadMNISTLabels('train-labels.idx1-ubyte');
A = images;
B=zeros(10,60000);
for i = 1:60000
    currentVector = zeros(1,10);
    currentIndex = mod(labels(i),10);
    if currentIndex==0
        currentIndex=10;
    end
    currentVector(currentIndex) = 1;
    B(:,i)=currentVector;
end

%Solve for X
A = A.';
B=B.';


%QR Decomposition
X=A\B;

%Pseudoinverse
%X=pinv(A)*B;
%Plot histogram of loading values


rowSum = zeros(784,1);
%Sum and rank the rows of X
for j=1:784
    rowSum(j)=sum(abs(X(j,:)));
```

[3] http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset

```
end

%pcolor of rowSum
%compositeImage=reshape(rowSum,[28,28]);
%pcolor(abs(compositeImage)), shading interp, colormap(hot)


%Sort and display top rows
topCount=201;
sortedRows=sort(abs(rowSum),'descend');
bar(sortedRows/sum(sortedRows(:)));
[topRows,I] = maxk(rowSum,topCount);
I=I.';

maskMatrix = zeros(784,10);
for k = I
    maskMatrix(k,:)=1;
end
Xsparse=maskMatrix.*X;

%pcolor of loading values
%h = pcolor(XSparse);
%set(h, 'EdgeColor', 'none');

%Sort and display top rows
topCount=569;
sortedRows=sort(abs(rowSum),'descend');
bar(sortedRows/sum(sortedRows(:)));
[topRows,I] = maxk(rowSum,topCount);
I=I.';

maskMatrix = zeros(784,10);
for k = I
    maskMatrix(k,:)=1;
end
Xsparse=maskMatrix.*X;


%Make a sparse B from A and X
Bsparse=mtimes(A,Xsparse);
Bfull=mtimes(A,X);

error = immse(Bsparse,B);
errorFull=immse(Bfull,B);

accurateCountFull=0;
for j=1:60000
[rowMaxPredicted,Ipredicted] = max(Bfull(j,:));
[rowMaxActual,Iactual] = max(B(j,:));
if Ipredicted==Iactual
  accurateCountFull = accurateCountFull + 1;
end
end
percentTrueFull=100*accurateCountFull/60000
```

```
accurateCountSparse=0;
for j=1:60000
[rowMaxPredicted,Ipredicted] = max(Bsparse(j,:));
[rowMaxActual,Iactual] = max(B(j,:));
if Ipredicted==Iactual
  accurateCountSparse = accurateCountSparse + 1;
end
end
percentTrueSparse=100*accurateCountSparse/60000
```

Code for fittings requiring vector input

```
clear all; close all; clc;
%http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset
images = loadMNISTImages('train-images.idx3-ubyte');
labels = loadMNISTLabels('train-labels.idx1-ubyte');
A = images;
B=zeros(10,60000);
for i = 1:60000
    currentVector = zeros(1,10);
    currentIndex = mod(labels(i),10);
    if currentIndex==0
        currentIndex=10;
    end
    currentVector(currentIndex) = 1;
    B(:,i)=currentVector;
end

%Solve for X
A = A.';
B=B.';
X=[];

for j=1:10
%Make B a vector
Bsingle=B(:,j);

%Backslash
%Xsingle=A\Bsingle;

%Pseudoinverse
%Xsingle=pinv(A)*Bsingle;

%LASSO
%lambda=0.1;
%Xsingle=lasso(A,Bsingle,'Lambda',lambda);

%Ridges
Xsingle=ridge(Bsingle,A,0.1);

%Robustfit
%Xsingle=robustfit(A,Bsingle,[],[],'off');

X=[X,Xsingle];
```

```
end

%Plot loading values of X with pcolor

rowSum = zeros(784,1);
%Sum and rank the rows of X
for j=1:784
    rowSum(j)=sum(abs(X(j,:)));
end


%compositeImage=reshape(rowSum,[28,28]);
%pcolor(abs(compositeImage)), shading interp, colormap(hot)



%Sort and display top rows
topCount=569;
sortedRows=sort(abs(rowSum),'descend');
bar(sortedRows/sum(sortedRows(:)));
[topRows,I] = maxk(rowSum,topCount);
I=I.';

maskMatrix = zeros(784,10);
for k = I
    maskMatrix(k,:)=1;
end
Xsparse=maskMatrix.*X;


%Make a sparse B from A and X
Bsparse=mtimes(A,Xsparse);
Bfull=mtimes(A,X);

error = immse(Bsparse,B);
errorFull=immse(Bfull,B);S

accurateCountFull=0;
for j=1:60000
[rowMaxPredicted,Ipredicted] = max(Bfull(j,:));
[rowMaxActual,Iactual] = max(B(j,:));
if Ipredicted==Iactual
  accurateCountFull = accurateCountFull + 1;
end
end
percentTrueFull=100*accurateCountFull/60000



accurateCountSparse=0;
for j=1:60000
[rowMaxPredicted,Ipredicted] = max(Bsparse(j,:));
[rowMaxActual,Iactual] = max(B(j,:));
if Ipredicted==Iactual
  accurateCountSparse = accurateCountSparse + 1;
```

```
end
end
percentTrueSparse=100*accurateCountSparse/60000



exampledigit = [4,6,8,10,12,14,16,18,20,22];
for j = 1:10
  testImage = A(exampledigit(j),:);
  testImage = reshape(testImage,[28,28]);
  testImage = rot90(testImage,2);
  testImage = flip(testImage);
  testImage = rot90(testImage,2);
  subplot(1,10,j), pcolor(testImage);
end

%Generate images of weightings
for j = 1:10
  testImage = X(j,:);
  testImage = reshape(testImage,[28,28]);
  subplot(2,5,j), pcolor(testImage);
end
```

Code for Single Digit Sparsity

```
clear all; close all; clc;
load('LassoX.mat')
images = loadMNISTImages('train-images.idx3-ubyte');
labels = loadMNISTLabels('train-labels.idx1-ubyte');

A = images;
B=zeros(10,60000);
for i = 1:60000
    currentVector = zeros(1,10);
    currentIndex = mod(labels(i),10);
    if currentIndex==0
        currentIndex=10;
    end
    currentVector(currentIndex) = 1;
    B(:,i)=currentVector;
end

%Solve for X
A = A.';
B=B.';

Xsparse=[];
maskMatrix=[];
for i = 1:10
%Find value for 90% of energy of weighting
Xslice=X(:,i);
sortedXslice = sort(abs(Xslice),'descend');
for j=1:784
    topCount=j;
    percent = sum(sortedXslice(1:j))/sum(sortedXslice);
    if percent > 0.9
        break
```

```
    end
end
[topPixels,I] = maxk(abs(Xslice),topCount);

maskVector = zeros(784,1);
for k = I
    maskVector(k)=1;
end
Xslicesparse=maskVector.*Xslice;
Xsparse=[Xsparse,Xslicesparse];
maskMatrix=[maskMatrix,maskVector];
end

%Sort and display top rows
topCount=569;
sortedRows=sort(abs(rowSum),'descend');
bar(sortedRows/sum(sortedRows(:)));
[topRows,I] = maxk(rowSum,topCount);
I=I.';

maskMatrix = zeros(784,10);
for k = I
    maskMatrix(k,:)=1;
end
Xsparse=maskMatrix.*X;


%Make a sparse B from A and X
Bsparse=mtimes(A,Xsparse);
Bfull=mtimes(A,X);

error = immse(Bsparse,B);
errorFull=immse(Bfull,B);


accurateCountFull=0;
for j=1:60000
[rowMaxPredicted,Ipredicted] = max(Bfull(j,:));
[rowMaxActual,Iactual] = max(B(j,:));
if Ipredicted==Iactual
  accurateCountFull = accurateCountFull + 1;
end
end
percentTrueFull=100*accurateCountFull/60000


accurateCountSparse=0;
for j=1:60000
[rowMaxPredicted,Ipredicted] = max(Bsparse(j,:));
[rowMaxActual,Iactual] = max(B(j,:));
if Ipredicted==Iactual
  accurateCountSparse = accurateCountSparse + 1;
end
end
percentTrueSparse=100*accurateCountSparse/60000
```