Comparing Machine Learning Algorithms Using MNIST Data Sets

Elliot Jennis

March 2021

Abstract

Machine learning algorithms were tested against the MNIST data set of hand drawn numerical digits to compare the accuracy of different classification methods. Linear Differential Analysis (LDA), Support Vector Machines (SVM), and Classification Trees were all used. Principle Component Analysis was also conducted to determine the highest energy modes. These modes helped group the data into clusters, allowing for the relative difficulty of classification between pairs of digits to be calculated. Direct comparisons between all three classification methods were made on the hardest and easiest pairs of digits. It was found that SVM produced the most accurate data classification with LDA being the second best. Classification trees did not perform well possibly due to a lack of pruning and an unrestricted number of branches.

Introduction and Overview

In the last few decades, the technological progress made in data storage that has brought modern computing from floppy disks, to hard drives, and then to solid state storage simultaneously increased access to a new kind of data analysis tool, machine leaning. Almost anyone can store the amount of data needed to use machine learning algorithms on their home computer. This increased access has also increased the pace of progress within the field of machine learning. The MNIST database of handwritten digits is an online combination of digitized handwritten digits from 0 - 9 capturing the handwriting of over two hundred and fifty people. The images are preprocessed and centered to allow the user to spend most of his or her energy on the analysis and learning algorithms rather than prepossessing. This data set was used to compare the accuracy of three different machine learning algorithms, Linear Differential Analysis (LDA), Support Vector Machines (SVM), and Classification Trees.

Theoretical Background

Discriminant analysis was developed in the 1930s. There are multiple types of discriminant analysis including Quadratic Discriminant Analysis (QDA) however for the scope of this paper, the focus will be on LDA. LDA is a supervised learning algorithm requiring a training

input and a test input. The linear modifier refers to the function of LDA. Similar to PCA, LDA finds a linear combination of vectors to characterize or separate different classes within a data set[1]. Two important variables are calculated to create this linear combination, the variance (Equation 1) and the covariance (Equation 2) of each class [1].

$$var(X) = \frac{1}{n_2} \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{1}{2}(x_i - x_j)^2 \tag{1}$$

$$cov(X, Y) = \frac{1}{n_2} \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{1}{2}(x_i - x_j)(y_i - y_j) \tag{2}$$

LDA optimizes to ensure that the resulting linear combination it creates to represent the data maximizes the covariance between classes while minimizing the variance within a class. There are a few inherent assumptions LDA uses to simplify this task. The covariances are assumed to be equal between the classes (QDA does not make this assumption), and the distribution of data within a class is normal[1].

Support Vector Machines (SVM) were developed in the 1990s and can classify data with nonlinear class boundaries [1]. The math underlying SVM relies on a hyperplane that separates groups of data. A hyperplane is subspace of of dimension $n-1$ within n-dimensional space. In two dimensions a hyperplane would be a line, in three dimensions it would be a plane, and so on. The use of a hyperplane negates the need to do data dimensionality reduction that other classifiers rely on. SVM attempts to find the optimal hyperplane to separate the classes within the data set it is given. The optimal hyperplane is defined as one with the maximum marginal distance between the closest data points to the hyper plane boundary (Figure 6 Appendix A). These hyperplanes can be linear, quadratic, and even radial by changing the kernal (K) in the support vector classifier (Equation 3) allowing for the SVM to classify data with non linear boundaries. [1].

$$f(x) = \beta_0 + \sum \alpha_i K(x, x_i) \tag{3}$$

Classification trees are a type of regression tree analysis that uses a recursive binary branching process to grow the tree. They consist of branches and nodes at the junction of branches (Figure 7 in Appendix A). The function that drives the growth of the tree is the 'classification error rate' [1] comparing the label the tree gives a particular data point and the most commonly occurring value in that region of data. An additional variable used to guide the tree branching is called the 'Gini Index' (equation 4)[1]. The Gini index is a way of measuring the variance across the K classes within a data set. The variable p_{mk} represents the proportion of training data observations in region m of class type k[1].

$$G = \sum_{k=1}^{K} p_{mk}(1 - p_{mk}) \tag{4}$$

G will be small for p values close to 1 and 0 which is a good indicator that a particular node in the tree has accurately classified the data in that region[1].

Algorithm Implementation and Development

There were five steps required to fully explore these different algorithms.

1. Load and sort the data
2. Singular Value Decomposition (SVM) and modal truncation
3. Determine easiest and hardest digit pairs to separate, confirm using LDA
4. Run SVM and Classification Trees on entire data set
5. Compare SVM, LDA, and Classification Trees on hardest and easiest pairs of digits

The MNIST data set uses a non traditional file format that cannot be directly imported into MATLAB. To alleviate this problem, a function file written by Nathan Kutz was used to alter the file format. It is included in Listing 1. The raw data includes a test set and training set. The training set is a series of 60,000 images each of size 28x28. The test set contains images of the dimensions but with only 10,000 images. Due to computational limitations only 10% of each data set was used to conduct this analysis. The only processing needed to use the data was a reshape so that each image became a single column vector. The final result produced a matrix of size 784x6000. Additionally, the data set was sorted using the *sort* command (Appendix B) to provide a clear visualization of the results (Listing 2).

The SVD of the data matrix was used to determine the number of modes needed for semi-accurate image reconstruction (Listing 3). Modal energy plots were produced, and visual representations of the images were created using varying mode counts. Once determined, the data and test sets were truncated to include only the first N modes deemed necessary for accurate results. The resulting matrices were computed by multiplying Σ and V for both the test and train sets. The new matrix dimensions became Nx6000 and Nx1000 with N representing the number of modes.

To identify hard to distinguish pairs of digits, a graphical process was used to avoid a brute force method of testing all possible combinations of digit pairs. First, the index of every digit was found within the training set and ten different arrays were created representing the indices of each digit (Listing 4). This allows for individual digits to be looked at rather then the entire data set at once. A 3D plot of the data projected onto three different V-modes was created with each digit represented by a different color (Listing 5). To make this cloud of data easier to distinguish, the centroid of each data cloud was calculated and plotted using k-means (Appendix A). Difficult to distinguish digits were close together on the 3D plot, with easier to distinguish pairs being further away. This was confirmed by an LDA classifier (Listing 6).

The next section of code involved the entire data set. SVM and Classification Tree analysis was done on all 6000 training images to sort all 1000 test images (Listings 7 and 8). The results were compared to eachother, and compared to the original data set with all 784 modes still present to see how modal truncation would affect the results.

The final step was to directly compare LDA, SVM, and Classification Trees on the easiest and hardest pairs of digits to separate. With the final step completed, the only remaining item was cross verification which will be discussed more in the results section.

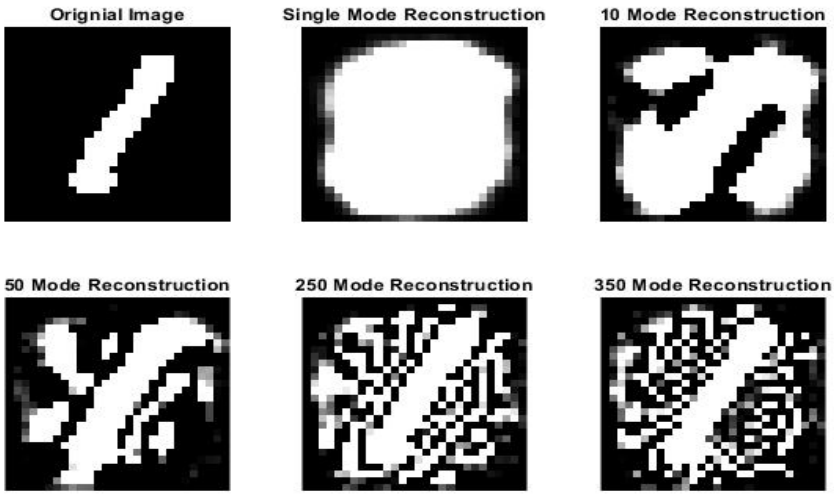
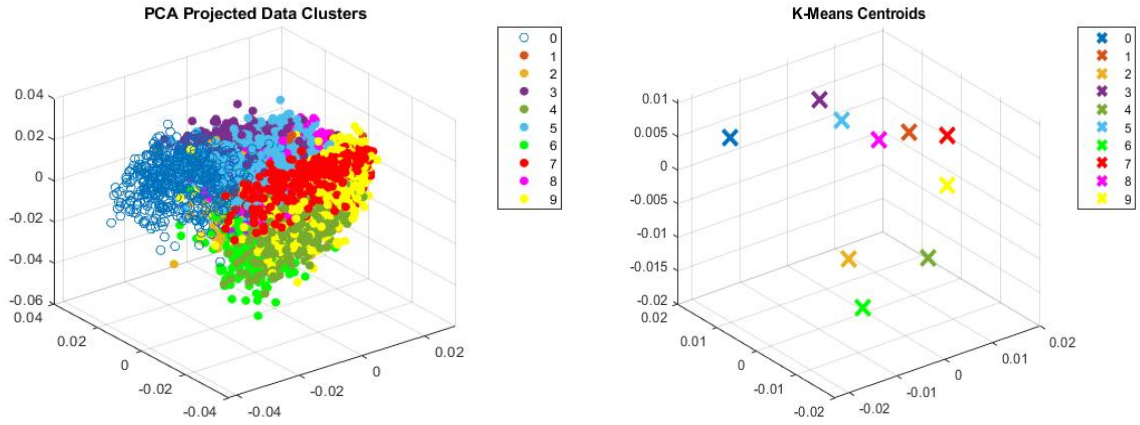


Figure 1: Image reconstruction using a variety of modes

Computational Results

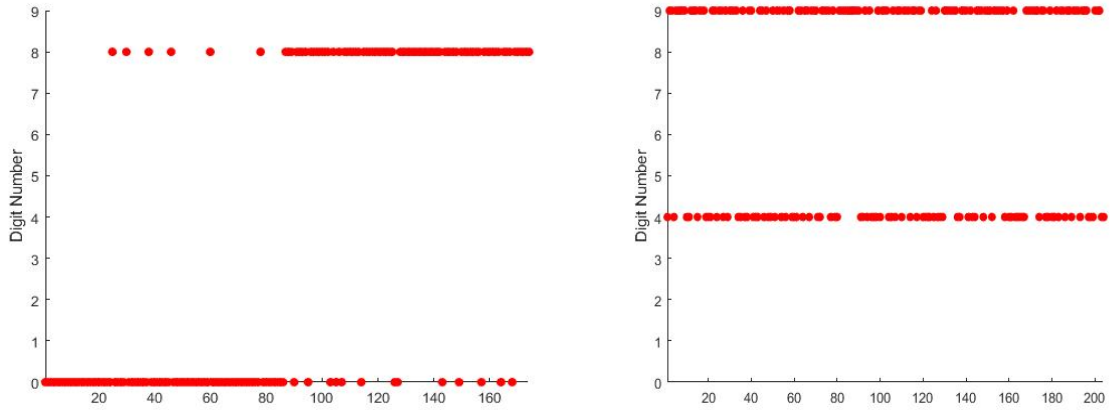
The results from the SVD of the test and training sets produce the same modal energy graph (Figure 8 Appendix A). Over 60% of the energy in the data is within the first mode. There is a gradually sloping curve starting at just over 20% for the remaining 783 modes. To better understand what this means qualitatively, images were reconstructed using an increasing number of modes to see how they compare to the original image (Figure 1). The first mode captures the boundary around the area where the digits appear, but lacks any detail within the white space. As more and more modes are added, the image becomes clearer with a distinct shape appearing after 50 modes are included. After jumping to 250 and then 350 modes, more detail is added but returns are diminishing. 250 modes were the amount chosen to conduct the classification analysis described below.

The data was projected onto V-modes 2,3, and 5. A 3D cluster plot color coded by digit type was created (Figure 2a). The clusters with the most crossover were pairs 7 and 9, 4 and 9, and 3 and 5. The pair with the least crossover was 0 and 8. LDA was conducted on the pairs to confirm which performed the worst and which performed the best. The conclusion being that the digits 0 and 8 were the easiest to tell apart with a error rate of 11.49% (Figure 3a). The hardest pair to distinguish was 4 and 9 with a 48.04% error rate (Figure 3b). A third case was investigated which included three digits 0, 4, and 8 (Figure 9 Appendix A). This combination had a 36.62% error rate. Repetition of the LDA analysis using SVM and Classification Trees was conducted to compare the three algorithms ability to accurately classify the data. For the digit pair of 0 and 8, SVM had an error rate of only 9.77% while classification trees had an error rate of 20.69%. The difficult to separate digit pair, 4 and 9, resulted in an error rate of 48.04% for SVM and 51.96% for Classification Trees. A final series of classification tests were conducted on all ten digits using both Classification Trees and SVM. The data set with only 250 modes showed an error rate of 64.4% while the original data set showed only 24% error (Figure 11 Appendix A). SVM showed an error rate of 58%



(a) Projected data clusters (b) Simplified centroidal plot

Figure 2: 3D plot of data clusters for each digit.



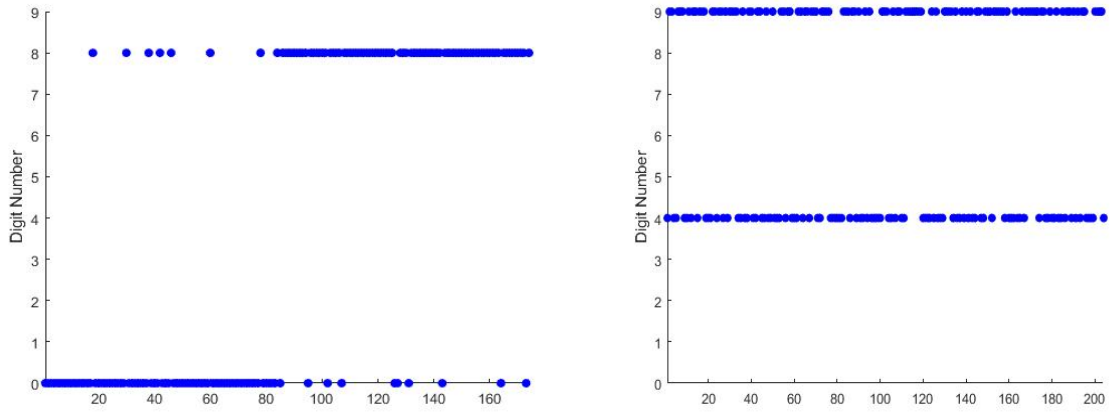
(a) Best performing digit pair 0 and 8 (b) Worst performing digit pair 4 and 9

Figure 3: LDA of digit pairs

for the truncated data and only 15.3% for the original set (Figure 10 Appendix A). Limited cross validation was conducted for all results using figures from the pool of test data not included during the original image reduction described in the Algorithm section. Five cases of cross validation were conducted for each result. SVM for ten digits was cross validated resulting in an average error rate of 70% for modally truncated data and 10.8% for the entire set with all 784 modes. Cross validation for classification trees showed an average error rate of 74% for the truncated data and 24.55% for the entire set.

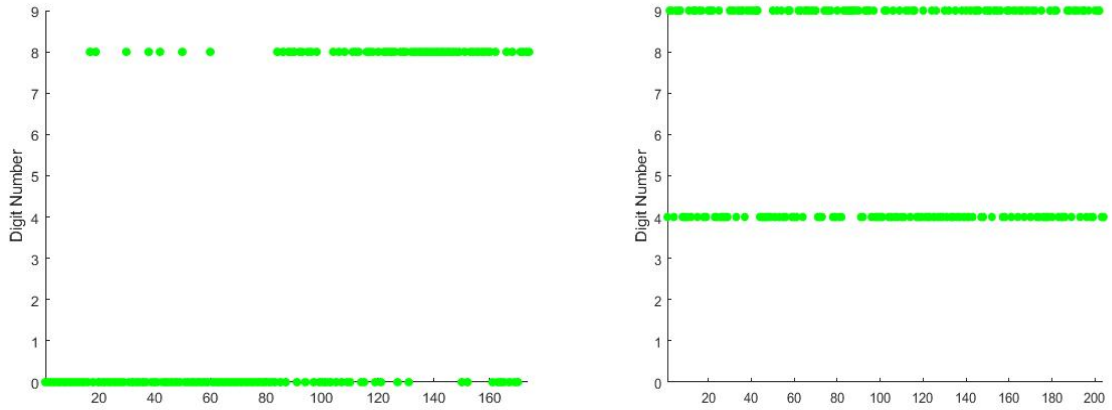
Summary and Conclusions

SVM performs the best of all three algorithms at classifying the digits. Its low error rate corresponds with the recency of its development as SVM was seen as ”state of the



(a) Best performing digit pair 0 and 8 (b) Worst performing digit pair 4 and 9

Figure 4: SVM of digit pairs



(a) Best performing digit pair 0 and 8 (b) Worst performing digit pair 4 and 9

Figure 5: Classification Tree sorting of digit pairs

art" as late as 2014 [2]. Surprisingly, classification trees were the worst of the three. One possible explanation could be the lack of customization of the classification trees used. The split setting was left in automatic mode, and no pruning or other advanced settings were put in place to affect the performance of the classification tree. It would be expected that had these other settings been used, the error rate could be pushed down below that of LDA. Additionally, the modal reduction used in this paper, runs counter productive to accurate digit classification. For both SVM and classification trees, the original data set left unchanged by the SVD modal truncation was far more accurate than the alternate version. While reducing the number of modes may be an enticing option when faced with limited computational power, it cannot be relied upon to produce a well trained algorithm.

References

[1] Gareth James et al. *An introduction to statistical learning*. Vol. 112. Springer, 2013.

[2] Jose Nathan Kutz. *Data-driven modeling & scientific computation: methods for complex systems & big data*. Oxford University Press, 2013.

A Figures

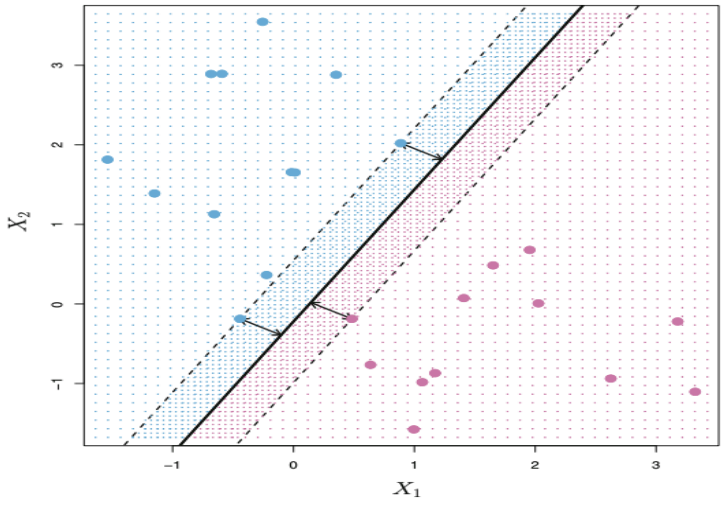


Figure 6: Visual representation of maximum margin hyperplane (James, Whitten, Hastie, and Tibshirani: 2013, p. 342[1])

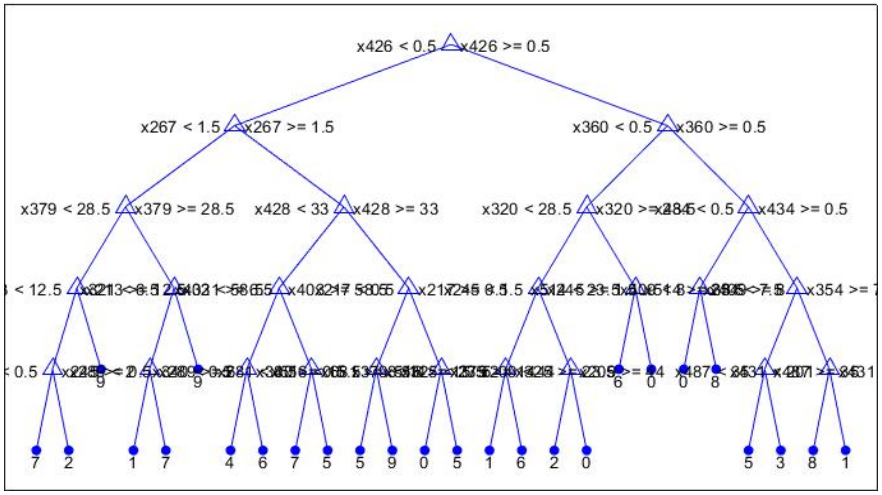


Figure 7: 25 branch Classification Tree used to identify digits 0 - 9

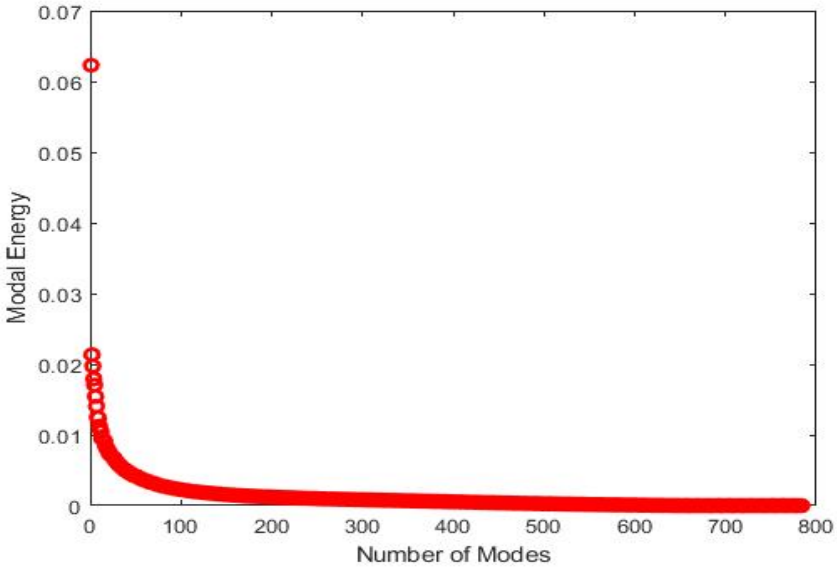


Figure 8: Modal energy values as a fraction of total energy

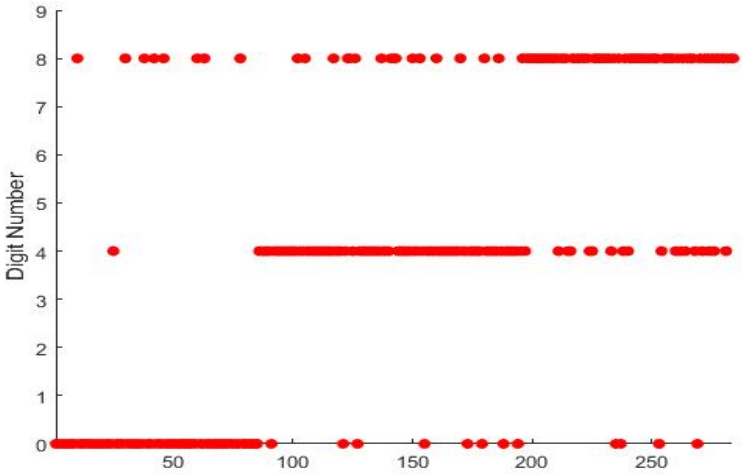
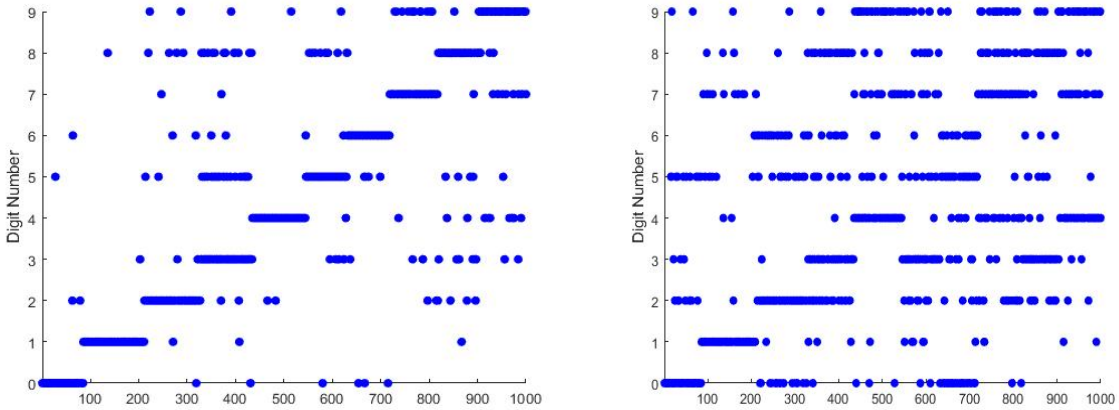
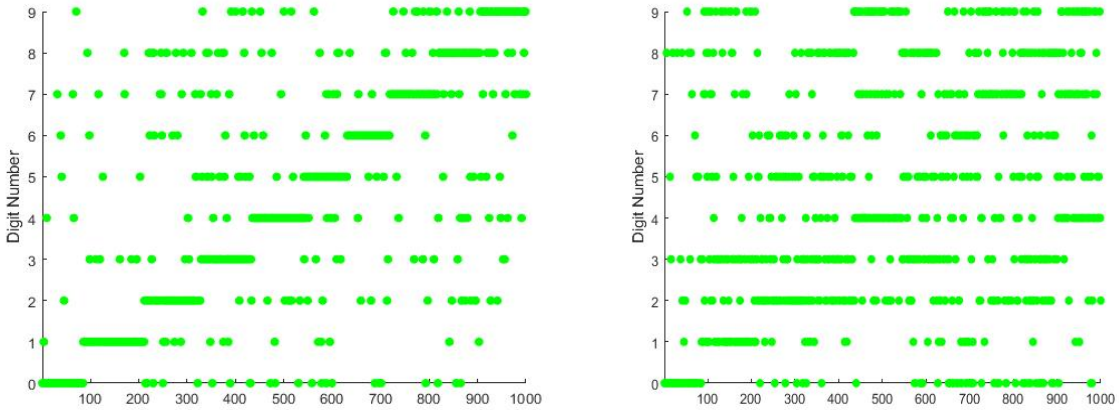


Figure 9: 25 branch Classification Tree used to identify digits 0 - 9



(a) Original sorted data (784 modes) (b) Truncated data (250 modes)

Figure 10: SVM classification performance between original and truncated data sets.



(a) Original sorted data (784 modes) (b) Truncated data (250 modes)

Figure 11: Classification tree performance between original and truncated data sets.

B MATLAB Functions

- `B = reshape(A,sz1,...,szN)` reshapes A into a sz1-by-...-by-szN array where sz1,...,szN indicates the size of each dimension. You can specify a single dimension size of [] to have the dimension size automatically calculated, such that the number of elements in B matches the number of elements in A. For example, if A is a 10-by-10 matrix, then reshape(A,2,2,[]) reshapes the 100 elements of A into a 2-by-2-by-25 array.

- `B = sort(_,direction)` returns sorted elements of A in the order specified by direction using any of the previous syntaxes. 'ascend' indicates ascending order (the default) and 'descend' indicates descending order.

- `[U,S,V] = svd(A,'econ') produces an economy-size decomposition of m-by-n matrix A. The economy-size decomposition removes extra rows or columns of zeros from the diagonal matrix of singular values, S, along with the columns in either U or V that multiply those zeros in the expression A = U*S*V'. Removing these zeros and columns can improve execution time and reduce storage requirements without compromising the accuracy of the decomposition.`

- `scatter3(X,Y,Z)` displays circles at the locations specified by the vectors X, Y, and Z.

- `idx = kmeans(X,k)` performs k-means clustering to partition the observations of the n-by-p data matrix X into k clusters, and returns an n-by-1 vector (idx) containing cluster indices of each observation. Rows of X correspond to points and columns correspond to variables. By default, kmeans uses the squared Euclidean distance metric and the k-means++ algorithm for cluster center initialization.

- `Mdl = fitcecoc(Tbl,ResponseVarName)` returns a full, trained, multiclass, error-correcting output codes (ECOC) model using the predictors in table Tbl and the class labels in Tbl.ResponseVarName. fitcecoc uses $K(K-1)/2$ binary support vector machine (SVM) models using the one-versus-one coding design, where K is the number of unique class labels (levels). Mdl is a ClassificationECOC model.

- class = classify(sample,training,group) classifies each row of the data in sample into one of the groups in training. sample and training must be matrices with the same number of columns. group is a grouping variable for training. Its unique values define groups; each element defines the group to which the corresponding row of training belongs. group can be a categorical variable, a numeric vector, a character array, a string array, or a cell array of character vectors. training and group must have the same number of rows. classify treats <undefined> values, NaNs, empty character vectors, empty strings, and <missing> string values in group as missing data values, and ignores the corresponding rows of training. The output class indicates the group to which each row of sample has been assigned, and is of the same type as group.

- `label = predict(SVMModel,X)` returns a vector of predicted class labels for the predictor data in the table or matrix X, based on the trained support vector machine

(SVM) classification model SVMModel. The trained SVM model can either be full or compact.

- `tree = fitctree(Tbl,ResponseVarName)` returns a fitted binary classification decision tree based on the input variables (also known as predictors, features, or attributes) contained in the table Tbl and output (response or labels) contained in Tbl.ResponseVarName. The returned binary tree splits branching nodes based on the values of a column of Tbl.
- `plot3(X,Y,Z,LineSpec)` creates the plot using the specified line style, marker, and color.

C MATLAB Code

```
function [images, labels] = mnist_parse(path_to_digits, path_to_labels)

% The function is curtesy of stackoverflow user rayryeng from Sept. 20,
% 2016. Link: https://stackoverflow.com/questions/39580926/how-do-i-load-in-the-mnist-

% Open files
fid1 = fopen(path_to_digits, 'r');

% The labels file
fid2 = fopen(path_to_labels, 'r');

% Read in magic numbers for both files
A = fread(fid1, 1, 'uint32');
magicNumber1 = swapbytes(uint32(A)); % Should be 2051
fprintf('Magic Number - Images: %d\n', magicNumber1);

A = fread(fid2, 1, 'uint32');
magicNumber2 = swapbytes(uint32(A)); % Should be 2049
fprintf('Magic Number - Labels: %d\n', magicNumber2);

% Read in total number of images
% Ensure that this number matches with the labels file
A = fread(fid1, 1, 'uint32');
totalImages = swapbytes(uint32(A));
A = fread(fid2, 1, 'uint32');
if totalImages ~= swapbytes(uint32(A))
    error('Total number of images read from images and labels files are not the same');
end
fprintf('Total number of images: %d\n', totalImages);

% Read in number of rows
A = fread(fid1, 1, 'uint32');
numRows = swapbytes(uint32(A));

% Read in number of columns
A = fread(fid1, 1, 'uint32');
numCols = swapbytes(uint32(A));

fprintf('Dimensions of each digit: %d x %d\n', numRows, numCols);

% For each image, store into an individual slice
images = zeros(numRows, numCols, totalImages, 'uint8');
for k = 1 : totalImages
    % Read in numRows*numCols pixels at a time
    A = fread(fid1, numRows*numCols, 'uint8');

    % Reshape so that it becomes a matrix
    % We are actually reading this in column major format
    % so we need to transpose this at the end
    images(:,:,k) = reshape(uint8(A), numCols, numRows).';
```

```
%Load in data
[images_train, labels_train] = mnist_parse('train-images-idx3-ubyte', 'train-labels-idx1
[images_test, labels_test] = mnist_parse('t10k-images-idx3-ubyte', 't10k-labels-idx1-uby

for i = 1:6000
    X_train(:,i) = reshape(images_train(:,:,i),[784 1]);
end
for i = 1:1000
    X_test(:,i) = reshape(images_test(:,:,i),[784 1]);
end
X_train = double(X_train);
X_test = double(X_test);
labels_train = labels_train(1:6000);
labels_test = labels_test(1:1000);

%% Sorting
[labels_train_sort, Indextrain] = sort(labels_train,'ascend');
X_train_sort=X_train(:,Indextrain);
[labels_test_sort, Indextest] = sort(labels_test,'ascend');
X_test_sort=X_test(:,Indextest);
```

Listing 2: Code used to create the data matricies used in the rest of the code

```
%% SVD
[U, S, V] = svd(X_train_sort,'econ');
[Us, Ss, Vs] = svd(X_test_sort,'econ');
Xr_test = Ss(1:250,1:250)*Vs(:,1:250)';
Xr_train = S(1:250,1:250)*V(:,1:250)';
figure(1)
% subplot(2,2,1)
% plot(diag(S)/sum(diag(S)),'ro','Linewidth',[2])
%% Modal Truncation
for j = 1:1
    Xm1 = U(:,1:j)*S(1:j,1:j)*V(:,1:j)';
end
for j = 1:10
    Xm10 = U(:,1:j)*S(1:j,1:j)*V(:,1:j)';
end
for j = 1:50
    Xm50 = U(:,1:j)*S(1:j,1:j)*V(:,1:j)';
end
for j = 1:250
    Xm250 = U(:,1:j)*S(1:j,1:j)*V(:,1:j)';
end
for j = 1:350
    Xm350 = U(:,1:j)*S(1:j,1:j)*V(:,1:j)';
end
figure(2)
subplot(2,3,1);
imshow(reshape(X_train_sort(:,600),[28 28]));
title('Orignial Image');
subplot(2,3,2);
imshow(reshape(Xm1(:,600),[28 28]));
title('Single Mode Reconstruction');
subplot(2,3,3);
imshow(reshape(Xm10(:,600),[28 28]));
title('10 Mode Reconstruction')
subplot(2,3,4);
imshow(reshape(Xm50(:,600),[28 28]));
title('50 Mode Reconstruction');
subplot(2,3,5);
imshow(reshape(Xm250(:,600),[28 28]));
title('250 Mode Reconstruction');
subplot(2,3,6);
imshow(reshape(Xm350(:,600),[28 28]));
title('350 Mode Reconstruction');
```

Listing 3: SVD and image reconstruction code

```
%% Count Data
c1 = 0; c2 = 0; c3 = 0; c4 = 0; c5 = 0; c6 = 0;
c7 = 0; c8 = 0; c9 = 0; c0 = 0;
sort_data = zeros
for i = 1:length(labels_train)
    if labels_train(i) == 1
        c1 = c1+1;
    end
    if labels_train(i) == 2
        c2 = c2+1;
    end
    if labels_train(i) == 3
        c3 = c3+1;
    end
    if labels_train(i) == 4
        c4 = c4+1;
    end
    if labels_train(i) == 5
        c5 = c5+1;
    end
    if labels_train(i) == 6
        c6 = c6+1;
    end
    if labels_train(i) == 7
        c7 = c7+1;
    end
    if labels_train(i) == 8
        c8 = c8+1;
    end
    if labels_train(i) == 9
        c9 = c9+1;
    end
    if labels_train(i) == 0
        c0 = c0+1;
    end
end

count1 = 0; count2 = 0; count3 = 0; count4 = 0; count5 = 0; count6 = 0;
count7 = 0; count8 = 0; count9 = 0; count0 = 0;
sort_data = zeros
for i = 1:length(labels_test)
    if labels_test(i) == 1
        count1 = count1+1;
    end
    if labels_test(i) == 2
        count2 = count2+1;
    end
    if labels_test(i) == 3
        count3 = count3+1;
    end
```

```
%% Problem 4

 figure(3)
 scatter3(V(zero,2),V(zero,3),V(zero,5));
 %plot zeros
 hold on
 scatter3(V(one,2),V(one,3),V(one,5),'filled');
 %plot ones
 hold on
scatter3(V(two,2),V(two,3),V(two,5),'filled');
%plot twos
 hold on
scatter3(V(three,2),V(three,3),V(three,5),'filled');
%plot threes
 hold on
 scatter3(V(four,2),V(four,3),V(four,5),'filled');
%plot fours
 hold on
 scatter3(V(five,2),V(five,3),V(five,5),'filled');
%plot fives
 hold on
 scatter3(V(six,2),V(six,3),V(six,5),'go','filled');
% %plot sixes
 hold on
scatter3(V(seven,2),V(seven,3),V(seven,5),'ro','filled');
%plot sevens
 hold on
scatter3(V(eight,2),V(eight,3),V(eight,5),'mo','filled');
%plot eights
 hold on
 scatter3(V(nine,2),V(nine,3),V(nine,5),'yo','filled');
%plot nines
legend('0','1','2','3','4','5','6','7','8','9');

[idx0,C0] = kmeans([V(zero,2) V(zero,3) V(zero,5)],1);
[idx1,C1] = kmeans([V(one,2) V(one,3) V(one,5)],1);
[idx2,C2] = kmeans([V(two,2) V(two,3) V(two,5)],1);
[idx3,C3] = kmeans([V(three,2) V(three,3) V(three,5)],1);
[idx4,C4] = kmeans([V(four,2) V(four,3) V(four,5)],1);
[idx5,C5] = kmeans([V(five,2) V(five,3) V(five,5)],1);
[idx6,C6] = kmeans([V(six,2) V(six,3) V(six,5)],1);
[idx7,C7] = kmeans([V(seven,2) V(seven,3) V(seven,5)],1);
[idx8,C8] = kmeans([V(eight,2) V(eight,3) V(eight,5)],1);
[idx9,C9] = kmeans([V(nine,2) V(nine,3) V(nine,5)],1);

figure(4)
plot3(C0(:,1),C0(:,2),C0(:,3),'x','Markersize',15,'LineWidth',3);
hold on
plot3(C1(:,1),C1(:,2),C1(:,3),'x','Markersize',15,'LineWidth',3);
hold on
```

```
%% Linear Discriminant Analysis
%initialize pairs and triplets of data
Test_Pair = [Zero Eight];
Train_Pair = [Zero_t Eight_t];
label_Train_Pair = [0*ones(c0,1); 8*ones(c8,1)];
label_Test_Pair = [0*ones(count0,1); 8*ones(count8,1)];

%LDA classification line
pre2 = classify(transpose(Test_Pair),transpose(Train_Pair),label_Train_Pair);
counter = 0;
for i =1:length(pre2)
    if pre2(i) == label_Test_Pair(i);
        counter = counter+1;
    end
end
fprintf('LDA Percent Error is %4.2f %', ((length(pre2)-counter)/length(pre2))*100);
figure(5)
scatter(i,pre2(i),'ro','filled');
axis([1 length(pre2) 0 9]);
```

Listing 6: LDA and two digit data matrix creation code

```
%% SVM classifier with training data, labels and test set
  Mdl = fitcecoc(transpose(Train_Pair),label_Train_Pair);
  pre3 = predict(Mdl,transpose(Test_Pair));

  counter1 = 0;
for i =1:length(pre3)
    if pre3(i) == label_Test_Pair(i);
        counter1 = counter1+1;
    end
end
fprintf('SVM Percent Error is %4.2f %', ((length(pre3)-counter1)/length(pre3))*100);
 scatter(i,pre3(i),'b','filled');
 axis([1 length(pre3) 0 9]);
```

Listing 7: SVM code

```
% Classification Tree
%
 tree=fitctree(transpose(X_train_sort),labels_train_sort);
% view(tree,'mode','graph');
 pre4 = predict(tree,transpose(X_test_sort));

 %data plotting and error calcuation
 counter4 = 0;
for i =1:length(pre4)
    if pre4(i) == labels_test_sort(i);
        counter4 = counter4+1;
    end
end
fprintf('Classification Tree Percent Error is %4.2f %', ((length(pre4)-counter4)/length(
figure(7)
scatter(1:length(pre4),pre4,'go','filled');
ylabel('Digit Number');
axis([1 length(pre4) 0 9]);
```

Listing 8: Classification Tree code