AMATH 482 - Assignment 4

Clarisa Leu-Rodriguez

Winter 2021 - March 10th, 2021

Abstract

In this paper, we explore the MNIST database of handwritten digits, which contains a training set of 60,000 handwritten digits and a test set of 10,000 handwritten digits. Our goal is to first use principal component analysis (PCA) to perform a low-rank approximation on each image in the training dataset, which will tell us the most important features in the images, and transform the dataset into PCA space. After transforming our training and test data into PCA space in order to improve computational performance - we perform the supervised classification method Linear Discriminant Analysis (LDA) on each pair of digits and each triplet of digits in order to determine the easiest & most difficult pair and triplet of digits to separate when using LDA. Additionally - we also perform two supervised machine learning algorithms on the data in PCA space - Support Vector Machine (SVM) and Decision Tree Learning (DTL) in order to compare and contrast between the three methods of classification between two and three digits. We also look at how well LDA, SVM, and DTL perform when trying to separate all ten digits. We explore the ideas of principal component analysis, linear discriminant analysis, support vector machines, and decision tree classifiers in this paper - as well as provide directions for future work on this problem.

Section I. Introduction and Overview

We are given the Modified National Institute of Standards and Technology (MNIST) database of handwritten digits, which contains a training set of 60,000 handwritten images and a test set of 10,000 handwritten images. The MNIST dataset is a subset of a larger dataset available from the National Institute of Standards and Technology (NIST). In the data set, the images have been size-normalized and centered in a fixed-size image. In Figure 1 below, we show the first eight images in both the training and test set.

Our goal when analyzing the MNIST dataset is to first perform a low-rank approximation on the 60,000 images (of size 28-by-28 pixels) in the training set and the 10,000 images (of size 28-by-28 pixels) in the test set. After reducing the dimensions of the dataset, we then look at three different methods for classifying the digits in the test set for two, three, and all ten digits. The first one is LDA, which is a supervised classification method which finds a linear combination of features which characterizes or separates two or more classes of objects or events - and then makes a classification on a new set of objects or events based on the values of a linear combination of characteristics (i.e. - feature values) according to a threshold. The next one is SVM which is a supervised machine learning model which uses classification algorithms to determine the optimal boundary between possible labels/outputs defined on the training set. The last one is DTL - which is a supervised machine algorithm often used for classification. A decision tree is a tree-like graph where the nodes represent the place we pick an attribute and ask a question and the edges are the answers. A decision tree learns in a binary approach - where it finds that best way to split each variable in the training set and constructs a tree - where the tree can then be used for classification by following the path down the decision tree. We will also discuss the performance of these classification methods not only on the test set, but also the training set for further validation (i.e. - to see if our models are overfitting).

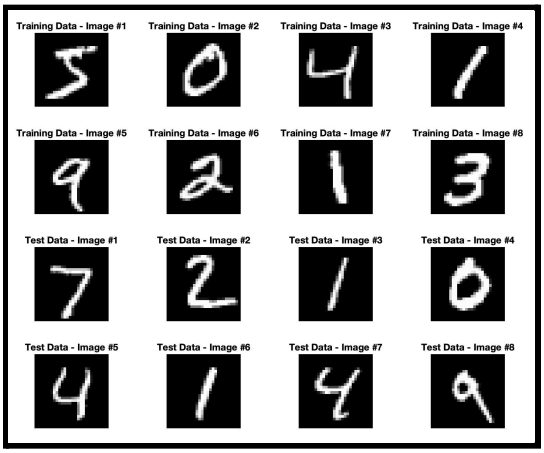

Figure 1: Original pictures of the first eight handwritten images in the MNIST training and test set, labeled respectively.

Section II. Theoretical Background

PCA is the process of computing the principal components (linear combinations of original features) of a system and using them to perform a change of basis on the data - reducing the dimensionality of the original dataset and performing a low rank approximation of the original system. PCA is often used as a preprocessing step before doing machine learning, as it improves the computational speed of these algorithms by reducing the dimensionality. Supervised machine learning uses a training set to teach models to yield a desired/correct output - where the training set includes correct inputs and correct outputs. After the model has been trained, it can then be tested on a test set - which is able to provide an unbiased evaluation of a final model fit on the training dataset. Additionally, one could test if their model is overfitting by using their model on the original training data. If the model is not overfitting - then it should yield similar results to the test dataset. In this section, I will briefly discuss the mathematics and theory behind PCA and the three different supervised classification methods/algorithms used in this paper - LDA, SVM, and DTL; as well as to how each of these apply to our solution if applicable.

Section II.I. Standardization & Scaling Data to Reduce Bias in PCA

Before performing the PCA method on a dataset, we typically standardize the dataset to ensure that all of our data contributes equally to our analysis - since PCA is biased towards features with high variance as it assumes there is only one common, shared variance in the dataset. That is, if there are large differences between the ranges of our variables in our data, those variables with larger ranges will dominate over those with small ranges, which leads to biased results. Standardizing transforms the data to comparable scales and helps to prevent this bias in values with large variation. Recall that standardizing involves subtracting the mean of the dataset from each datapoint and dividing by the standard deviation of the dataset. The formula is:

$$z = \frac{x_i - \mu}{\sigma}$$

Equation 1: The standardization of x where μ is the mean of x and $\boldsymbol{\sigma}$ is the standard deviation of x.

In our solution, in order to reduce bias - we subtract the mean over each particular pixel across all 60,000 images in the training set from both the training & test data. This is because we want to have the average values of each pixel to be zero across all images - reducing bias in pixels with large variation across images. While we did not standardize our data beforehand and instead did this method of scaling, we certainly could try to standardize our data beforehand in the future & compare the results. It is important to note that when performing PCA - you should choose to scale or not scale your data according to the type of data collected/its context (i.e. - sometimes scaling your data is not necessary or and the way you scale your dataset/standardize it might require trial and error) - where generally some method of scaling or normalization of the dataset before PCA is very common.

Section II.II. Singular Value Decomposition (SVD) & Energy

SVD is the factorization of a matrix which is a generalization of the eigendecomposition of a square normal matrix to any m x n matrix - where every m x n matrix A has a SVD. The SVD of a matrix A is shown below.

$$A = U\Sigma V^*$$

Equation 2: The SVD of a Matrix A where A is m x n.

In Equation 2, if A is complex - U is an m x m complex unitary matrix (i.e. its conjugate transpose is also it's inverse - $U^*U = UU^* = I$), Σ is an m x n rectangular diagonal matrix with non-negative, real number along its diagonal, and V^* is an n x n complex unitary matrix. If A is real, then U and $V^{\mathrm{T}} = V^*$ are real orthogonal matrices (i.e. their columns and rows are orthonormal vectors - $U^{\mathrm{T}}U = UU^{\mathrm{T}} = I$ and $V^{\mathrm{T}}V = V V^{\mathrm{T}} = I$).

U and V are rotational matrices and $\Sigma$is a stretching matrix - where the diagonal entries along Σ are the uniquely determined singular values of A, the columns of U are left-singular vectors, and the columns of V are right-singular vectors of A. The total number of non-zero singular values gives us the rank of A - where the SVD is often used as the preferred method for matrix rank calculations in mathematical software, as often it is more reliable due to approximation errors in real numbers. Additionally, other properties of the SVD include:

- The singular values of A are always ordered from largest to smallest (i.e. $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$) along the diagonals of Σ.

- Letting r = rank(A), we have that $\{u_1, u_2, ..., u_r\}$ is a basis for the range of A and $\{v_r + 1, ..., v_n\}$ is a basis for the null space of A.

As it pertains to PCA, the SVD of a matrix tells us the dimension and the direction of the principal components of the matrix. It reveals information as to how much data is in each dimension/rank approximation (via the singular values of A). In our solution, after determining the principal components in our system - we then determine which ones contribute the most energy to our system through looking at the energy captured in each dimension/rank through the following equation:

$$energy_n = \frac{\sigma_1^2 + ... + \sigma_n^2}{\sigma_1^2 + ... + \sigma_r^2}$$

Equation 3: The energy contained in the rank-n approximation of A (where A is of rank r) captured by the singular values (i.e. the $\boldsymbol{\sigma}_i$'s) of A.

When analyzing which principal components contribute the most to our system in the PCA method, we see which PCA modes (in our case - columns of U) have the highest amount of energy captured in its dimension. This tells us which other modes are also redundant to summarizing the system (i.e. - those with low amounts of energy captured with respect to the other modes). This is what we did in our solution, as well as looking at how well our low rank approximation of our system did by projecting our original data back onto the principal component bases to see how well our low-rank approximations did & looking at the bases (columns of U) themselves too.

Section II.III. Multiclass & Two-Class Linear Discriminant Analysis (LDA)

The goal of LDA is to find a suitable projection which both maximizes the distance between inter-class points and minimizes the intra-class points. LDA is very similar to PCA - as they both look for linear combinations of variables which best explain the data. However, LDA explicitly attempts to model the difference between the classes of data while PCA does not take into account any difference in class. To find the right subspace to project into for LDA - we first calculate the means for each of our groups for each feature. After calculating the means, we then define the between-class scatter matrix as:

$$S_B = \sum_{j=1}^{N} (\mu_j - \mu)(\mu_j - \mu)^T$$

Equation 4: The between-class scatter matrix where μ is the overall mean and μ_j is the means of each of the $N \geq 3$ groups/classes.

The between-class scatter matrix is a measure of the variance between the groups (i.e. - between the means). Then, we also define the within-class scatter matrix as:

$$S_w = \sum_{j=1}^{N} \sum_{x} (x - \mu_j)(x - \mu_j)^T$$

Equation 5: The within-class scatter matrix.

The within-class scatter matrix is a measure of the variance within each group. The goal then of LDA is to then find a vector w such that:

$$w = argmax \frac{w^T S_B w}{w^T S_w w}$$

Equation 6: The equation to find the vector w for LDA.

Although this will not be proved in this paper, the vector w which maximizes the above quotient in Equation 6 is the eigenvector corresponding to the largest eigenvalue of the generalized eigenvalue problem:

$$S_B w = \lambda S_w w$$

Equation 7: The generalized eigenvalue problem for LDA related to Equation 6.

Once we have w, i.e. the subspace which is a suitable projection that maximizes the distance between the inter-class data while minimizing the intra-class data - we can then pick a cutoff between the classes to be used as a threshold for decision making on a new dataset based on the linear combination of feature values.

Also note that when implementing LDA for two classes - the between-class scatter matrix varies slightly:

$$S_B = (\mu_2 - \mu_1)(\mu_2 - \mu_1)^T$$

Equation 8: The between-class scatter matrix for two class LDA.

And the within-class scatter matrix varies slightly to Equation 5 - where we restrict $N = 2$.

Section II.IV. Support Vector Machines (SVM)

Simply put, the goal of SVM is to determine the optimal hyperplane/boundary between the possible outputs of a dataset. This is done through complex data transformations and looking at the data in *n*-dimensional space (where *n* is the number of features we have) and the value of each feature being the value of a particular coordinate. SVM then determines the best way to separate the data based on the labels defined. While this paper will not go into depth on the mathematics behind SVM, it is important to note that SVM is a powerful supervised machine learning algorithm where it's main drawback lies in the magic behind it. The complex data transformations and resulting boundary plane are oftentimes very difficult to interpret with data with a lot of features (and is often why it is used as a black box in machine learning) - where in contrast LDA and DTL are generally very easy to visually understand. Additionally, SVMs are generally better to use with data which is linearly separable although it can be used with nonlinear data as well (just it might take longer to train).

Section II.V. Decision Tree Learning (DTL)

In DTL, a binary decision tree is constructed based on a sequential decision process. This decision tree is a flowchart like tree structure, where each internal node denotes a test on an attribute and each branch represents an outcome of the test and each leaf/terminal node holds a class label. At a high level, they are constructed where for each feature in our dataset, we scan over all possible values of that feature and determine the best way to split up the features in the dataset according to a cost function. The best split will have the lowest probability that a specific feature will be classified incorrectly when following the path down the decision tree. In this way, decision trees can be used to perform classification without too much computation by following the path down the decision tree to the leaf/terminal node (i.e. - label). However, they can sometimes be computationally expensive to train since the process of growing a decision tree involves determining the optimal split at each decision node - and each candidate split must be sorted before the best split can be found. Similar to how we are using SVM in our solution, we will also use DTL as a black box for classification and not go in depth into the mathematics behind DTL in this paper - although available references in the References section of this paper are provided for the reader if they would like to learn more.

Section III. Algorithm Implementation and Development

To start - we first import the MNIST data into MATLAB which consists of a training dataset of 60,000 images and a test dataset of 10,000 images. Each image is 784 pixels and 28-by-28 in dimensions. To import the data, we use the provided starter code function `mnist_parse` which correctly parses the MNIST data and puts it into a matrix of size 28-by-28-number of images. We then resize the image data for both the test and training dataset to have every image be a column - giving us a matrix of size 784-by-number of images. Before performing SVD and doing a low-rank approximation of the data using PCA - we also first scale the data by subtracting the row-wise mean of the pixel space of the training data. This makes it so that each pixel has a mean of zero - and makes it so pixels with high variance between different images do not bias our results when performing PCA. We also subtract the row-wise mean of the pixel space for the test data as well. Next, we perform PCA on the scaled training data and get our *U* matrix which is of size 784-by-784, *V* matrix which is of size 60,000-by-784, and *S* matrix which is of size 784-by-784.

Interpreting our *U, S,* and *V* matrices - each column of *U* describes the variance in the columns of our data and gives us a basis where we can represent each column of our original data (i.e. - handwritten images) and are the left singular vectors. That is, the columns of *U* are the principal directions/spatial features needed to construct an image. *S* contains singular values along its diagonal, ordered by importance and describes the energy captured in each column (i.e. - mode) of *U*. Analyzing the singular values of *S* we are able to determine which modes of *U* contribute the most energy and determine the rank of the digit space based on a certain threshold of energy we are looking to capture in our approximation. Interpreting *V*, the rows of *V* give the set of coefficients needed to build a particular image using the columns of *U*. That is, *V* tells us the linear combination of the columns of *U* (i.e. eigen mixture) to make each particular image in the dataset.

The next step in the PCA method is to determine the rank of the digit space - which we will use as the number of columns of U needed to accurately represent our data (i.e. after projecting our data onto the first rank r columns of U - we will have a low-rank approximation of the original data which is of size rank r-by-number of images). The distribution of the singular values and energies captured in each of the 748 modes is shown in Figure 2 which can be found in Appendix C. Additional Figures and Tables. We determine the rank to be 87 which constitutes 90.01% of the energy of the original data using Equation 3.

Next, we look at how well our approximation did by plotting a few of the columns of U and also looking at how well our low-rank approximation did in reconstructing the first eight images of the training data. This is Figure 3, shown in Appendix C. Additional Figures and Tables. Comparing Figure 3 to Figure 1 (found on page 1) - we see that our approximation did pretty well in reconstructing the first eight images of the training set - and the columns of U look reasonable as well. Additionally, we plot our training data projected onto the first three modes of U which is shown in Figure 4, found in Appendix C. Additional Figures and Tables. What we see in the 3D plot are clusters which tell us how the digits are separated. Clusters which are closer together we would expect those digits to be harder to separate as the principal components are very similar for those images/overlap. Clusters which are farther apart we would expect those digits to be more difficult to separate as the principal components are much more distinct between them.

After performing PCA on the training dataset, we then project our training and test data into PCA space (i.e. we take the first rank r columns of U and multiply its transpose with our data) and reduce the dimensions of our datasets from 784-by-number of images to 87-by-number of images. We are then ready to use our low-rank approximation of the image data on three different supervised learning classification methods, LDA, SVM, and DTL. Our main method for comparing the three is to look at the true positive rate - i.e. we train our models with the training data and then use our test data and see the percentage our models were able to correctly classify the test data. For the two-digit case, we loop through all possible digit pairs - which constitutes $C(10, 2) = 45$ different unique pairs as there are ten digits. We construct our test and training data (and corresponding labels) for each digit pair by looking for the indices which correspond to the digit pair we are interested in from the given labels in the original dataset in PCA space. For training, we pass the training data into our LDA, SVM, and DTL with our labels. Additionally, for SVM we rescale/normalize our training data by dividing each value by the maximum value in the training dataset due to how MATLAB is expecting the training data to be formatted (i.e. - they should be within a tight interval between [-1 1]). For DTL - we let MATLAB pick the maximum number of splits at each node (where future work for this portion is discussed in Section V. Summary and Conclusions). Next - we predict the classes/labels of the test data using our constructed models. Then, we calculate the true positive rate our models were able to predict the class of the test data by comparing the test labels to the predicted labels and store this value. This process described above was also repeated for the three-digit case - where instead we just loop through all possible digit triplets - which constitutes $C(10, 3) = 120$ different unique triplets. For the ten digit case - we also repeat the above process described as well - where instead we do not need to loop through any digits as $C(10, 1) = 1$. We also repeat the process described above to try and predict our training data as well with our constructed models - in order to see if our models are overfitting or not.

Finally, when comparing how well each classifier did - we construct six tables which are shown in Appendix C. Additional Figures and Tables which report the true positive rate for every possible pair/triplet and the ten digit case for each method of classification explored - and also reports how well each method of classification did in the worst case, best case, and on average.

Section IV. Computational Results

Please refer to Appendix C. Additional Figures and Tables for our computational results (i.e. - table of results from LDA, SVM, and DTL as well as figures regarding analysis of our low-rank approximation as discussed in Section III. Algorithm Implementation and Development.

Section V. Summary and Conclusions

In this section, we discuss the results of Tables 1 through 6 found in Appendix C. Additional Figures and Tables. More information regarding our analysis for PCA can be found in Section III. Algorithm Implementation and Development. Comparing and contrasting the three different methods of classification explored in this paper - for the two-digit case in Table 1, we find that LDA and SVM found digits 1 and 4 the easiest to separate where LDA had a true positive rate of 99.76% and SVM had a true positive rate of 99.95% for classification. For our DT classifier, digits 0 and 1 were the easiest to separate with a true positive rate of 99.67%. DTLs performance on separating digits 1 and 4 was very comparable - having a true positive rate of 99.57%. For the

hardest digit to separate - LDA and SVM found digits 5 and 8 the most difficult to separate where LDA had a true positive rate of 95.07% and SVM had a true positive rate of 95.82%. DTL found digits 4 and 9 the most difficult to separate with a true positive rate of 91.01%. For digits 5 and 8 - DTL did much better though with a true positive rate of 99.2%. On average, LDA reported a true positive rate of 98.38%, SVM had a true positive rate of 98.77% and DTL had a true positive rate of 96.75%. So - on average we find SVM did slightly better then LDA and DTL. In the worst case - DTL did poorly in comparison although it did slightly better on more difficult to separate digits (i.e. - digits 5 and 8 are very similar in shape).

Looking at the results for the training dataset for the two digit case in order to see if our model was over fitting found in Table 2 - we see that the results vary. In the test data set - LDA found digits 6 and 7 the easiest to separate with a true positive rate of 99.81% and SVM and DTL found digits 0 and 1 the easiest to separate where SVM reported a true positive rate of 99.93% and DTL a rate of 98.74%. In the worst case - LDA and SVM found digits 3 and 5 the most difficult to separate where LDA reported a true positive rate of 95.39% and SVM reported a rate of 95.92%. DTL found digits 4 and 9 the most difficult to separate with a positive rate of 98.74%. On average - we see that LDA reported a true positive rate of 98.3%, SVM a rate of 98.83%, and DTL a rate of 99.56%. While LDA and SVM have very similar values on average for the true positive rate when compared to the test data - but very slightly in the best/worst case from the test data - it would be safe to say our model was not overfit for these classifiers. For the DTL classifier - the very high true positive rate on the training data but not very comparable to the training data set leads us to believe the model was slightly overfitting the data here. Future work for this problem would be to fine tune the different parameters on the DTL classifier - such as the maximum number of splits at each node.

For the three-digit case in Table 3 - in contrast to our two-digit results we find that for the test data classification our results all agree on the easiest and hardest triplet of digits to separate. The easiest triplet of digits to separate for all three methods of classification was 0, 1, and 4 where LDA reported a true positive rate of 99.48%, SVM a rate of 99.74%, and DTL a rate of 98.35%. The hardest triplet of digits to separate was 3, 5, and 8 where LDA reported a true positive rate of 91.86%, SVM a rate of 93.74%, and DTL a rate of 86.44%. On average, LDA reported a true positive rate of 96.71%, SVM a rate of 97.87%, and DTL a rate of 94.24%. So, on average we find that SVM also outperformed the other two methods of classification with LDA doing comparably. In the best and worst case we also see SVM does better than the other two classifiers.

Looking at our results as well for our model when trying to classify the training dataset to see if our model was overfitting in Table 4 - we find that our results vary for the easiest triplet of digits to separate but are consistent with the most difficult triplet of digits to separate. The most difficult triplet of digits to separate for the training dataset was also 3, 5, and 8 with LDA reporting a true positive rate of 91.47%, SVM reporting a rate of 93.71%, and DTL reporting a rate of 97.58%. For the best care - LDA found digits 1, 6, and 9 the easiest to separate for the training data with a rate of 99.08%, SVM found digits 1, 6, and 7 the easiest with a rate of 99.64%, and DTL found digits 0, 1, 4 the easiest to separate with a rate of 99.8%. On average - LDA reported a true positive rate of 96.51% for LDA, 97.92% for SVM, and 99.08% for DTL. While the average rates are very comparable to the results of our model on the test data set for SVM & LDA, we see that for DTL we are reporting a higher rate on average for the training then the test (like in the two-digit case). Similarly, future work for this case would be fine tuning the parameters of our DTL model to try and reduce any overfitting.

For the ten-digit case, analyzing the results of our models on the test set found in Table 5 - we find that LDA reports a true positive rate of 87.67%, SVM a true positive rate of 94.11%, and DTL a true positive rate of 84.35%. So, SVM does significantly better than the other two classifiers when trying to classify between all ten digits. Looking at our results for our model for the ten-digit case on our training data to see if our models are overfitting found in Table 6 - we find that LDA reported a true positive rate of 87.01%, SVM a rate of 94.11%, and DTL a rate of 96.07%. Similar to the two and three-digit case - we find that our LDA and SVM models are most likely not overfitting as our results are very comparable to our results on the test set while DTL is most likely overfitting the training data.

In conclusion, we were able to successfully build three different classification models using MATLAB for the MNIST dataset and compare and contrast the different models. As stated previously, future work on this problem would be trying to fine tune the parameters for our DTL classifier in order to avoid overfitting in our model. Additionally - we could try and explore building the discussed classifiers by hand in order to further understand the mathematics behind them. Also - we can try other different supervised machine learning models and compare them to the three discussed in this paper as well. We could also do further cross validation on our models in the future as well.

Section VI. References

- http://yann.lecun.com/exdb/mnist/ - I used this source to get the MNIST dataset online.
- https://stackoverflow.com/questions/39580926/how-do-i-load-in-the-mnist-digits-and-label-data-in-matlab/ - This source was used in order to get the starter code for importing/arranging the MNIST data in MATLAB.
- https://en.wikipedia.org/wiki/Singular_value_decomposition - I used this source to better understand Singular Value Decomposition.
- https://en.wikipedia.org/wiki/Principal_component_analysis - I used this source to better understand Principal Component Analysis.
- https://en.wikipedia.org/wiki/Support-vector_machine - I used this source to better understand support-vector machines.
- https://en.wikipedia.org/wiki/Linear_discriminant_analysis - I used this source to better understand linear discrriminant analysis.
- https://en.wikipedia.org/wiki/Decision_tree_learning - I used this source to better understand decision tree learning.
- In addition, the course notes provided on the course website and all lecture material was used.

Appendix A. MATLAB Functions Used and Brief Implementation Explanation

This appendix contains a list of the MATLAB functions used and additional information regarding how it was used to solve the given problem. Additional information can also be found in Section II. Theoretical Background, Section III. Algorithm Implementation and Development, and in the comments of Appendix B. MATLAB Codes. Note, descriptions of functions were sourced from the MATLAB documentation which can be found here online: https://www.mathworks.com/help/matlab/.

- `FID = fopen(FILENAME)` opens the file FILENAME for read access where `FILENAME` is the name of the file to be opened and returns a file identifier `FID`. This function was used to load in the downloaded MNIST data into MATLAB in the provided helper function `mnist_parse`..
- `A = fread(FID, SIZE, PRECISION)` reads the file with file identifier `FID` according to the data format specified `PRECISION`, where the `SIZE` argument is optional. This function was used to read the MNIST data into MATLAB with a specified format in the provided helper function `mnist_parse`.
- `ST = fclose(FID)` closes the file associated with file identifier `FID`, which is an integer value obtained from `fopen`. Returns 0 if successful and -1 otherwise. We used this after loading and reading in the MNIST data into MATLAB in order to close the file in the provided helper function `mnist_parse`..
- `Y = reshape(X, M, N, P, ...)` or `Y = reshape(X, [M, N, P, ...])` returns an `N-D` array with the same elements as `X` but reshaped to have the size `M`-by-`N`-by-`P`-by-`...`. The product of the specified dimensions, `M * N * P * ...,` must be the same as `NUMEL(X)`. We used this function in our code to reshape matrices to different dimensions when visualizing our data, reading in our data, or performing SVD.
- `I2 = im2double(I1)` converts the image `I1` to double precision, rescaling the data if necessary. We used this to convert our image data after importing to double precision in order to perform computations on it.
- `I2 = im2uint8(I1)` converts the image `I1` to uint8 precision, rescaling the data if necessary. We used this to convert our image data from double precision back to uint8 when displaying the image.
- `M = size(X, DIM1, DIM2, …, DIMN)` returns the lengths of the dimensions `DIM1, …, DIMN` as a row vector. This was used to determine the sizes of different dimensions in matrices.
- `imshow(X)` displays the image `X` in MATLAB. We used this to display images in our code.
- `mean(X, DIM)` takes the mean along the dimension `DIM` of `X`. This was used to calculate the row wise mean of our training data when scaling before doing SVD.
- `B = repmat(A, M, N)` creates a large matrix B consisting of an `M`-by-`N` tiling of copies of `A`. If `A` is a matrix, the size of `B` is `[size(A, 1) * M, size(A, 2) * N]`. This was used in order to subtract the row-wise mean from our training and test data before doing SVD.
- `D = diag(X)` returns the main diagonal of matrix `X`. We used this when analyzing the singular values after performing SVD.
- `[U, S, V] = svd(X)` produces a diagonal matrix `S`, of the same dimension as `X` and with nonnegative diagonal elements in decreasing order, and unitary matrices `U` and `V` so that `X = U * S * V'`. This was used to calculate the SVD of our scaled training data.
- `R = rescale(A)` rescales all entries of an array `A` to [0, 1]. We used this when looking at the columns of U after performing SVD and using `imshow`.
- `I = find(X)` returns the linear indices corresponding to the nonzero entries of the array `X`. `X` may be a logical expression. We used this to find the indices of different digits in our training and test labels - where we then took these indices and divided our training and test data based on digits.
- `CLASS = classify(SAMPLE, TRAINING, GROUP, TYPE)` classifies each row of the data in `SAMPLE` into one of the groups in `TRAINING` using discriminant analysis. `SAMPLE` and `TRAINING` must be matrices with the same number of columns. `GROUP` is a grouping variable for `TRAINING`. Its unique values define groups, and each element defines which group the corresponding row of `TRAINING` belongs to. `GROUP` can be a categorical variable, numeric vector, a string array, or a cell array of strings. `TRAINING` and `GROUP` must have the same number of rows. `CLASS`

indicates which group each row of SAMPLE has been assigned to, and is of the same type as GROUP. TYPE specifies the type of discriminant function to use and is by default linear. Linear discrimination fits a multivariate normal density to each group, with a pooled estimate of covariance and uses likelihood ratios to assign observation to groups. We used this when performing LDA on our training and test dataset.

- OBJ = **fitcecoc**(X, Y) fits a multiclass model for Support Vector Machine or other classifiers. X is an N-by-P matrix of predictors with one row per observation and one column per predictor. Y is the response and is an array of N class labels. Note that X should also only contain data within [-1, 1] or another tight interval. We use this when performing SVM on our training data.
- LABEL = **predict**(MODEL, X) returns predicted class labels LABEL. X must be a table if MODEL was originally trained on a table, or a numeric matrix if MODEL was originally trained on a matrix. If X is a table, it must contain all the predictors used for training this model. If X is a matrix, X must have P columns, where P is the number of predictors used for training. Classification labels LABEL have the same type as Y used for training. We used this in order to predict the labels of our test data set after building the model for SVM and DTL.
- TREE = **fitctree**(X, Y) returns a classification decision tree for data in the N-by-P matrix of predictors X with one row per observation and one column per predictor. Y is the response and is an array of N class labels. We used this when performing DTL on our training data.

Appendix B. MATLAB Codes

This appendix contains the MATLAB code used in this paper. Please refer to Appendix A. MATLAB Functions Used and Brief Implementation Explanation for more information regarding how each of the MATLAB functions used in our code works, as well as the process used to solve the given problem.

`assignment_4.m`

```
%{
    Assignment #4 - Classifying Digits (MNIST)
    AMATH482 - Computational Methods For Data Science - Mar. 10th, 2021
    Taught by Professor Jason J. Bramburger (Winter 2021)
    Written By: Clarisa Leu-Rodriguez - email: cleu@uw.edu
%}


%% Import & Parse MNIST Data
clear all; close all; clc;
[images_train, labels_train] = mnist_parse('train-images-idx3-ubyte1', ...
    'train-labels-idx1-ubyte1');
[images_test, labels_test] = mnist_parse('t10k-images-idx3-ubyte1', ...
    't10k-labels-idx1-ubyte1');


%% Part #1 - Perform SVD Analysis of Digit Images & Low-Rank Approximation
% Convert image data to double to do calculations.
images_train = im2double(images_train);
images_test = im2double(images_test);

% Resize Data - Each column vector is a different image.
S_train = size(images_train);
S_test = size(images_test);
dat_train = reshape(images_train, [S_train(1) * S_train(2), S_train(3)]);
dat_test = reshape(images_test, [S_test(1) * S_test(2), S_test(3)]);

% Plot first 8 images of training data and test data.
figure();
image_dim = 28;  % Images are 28 * 28 = 784 pixels.
for k = 1:8
    subplot(4, 4, k);
    im_check = reshape(im2uint8(dat_train(:, k)), image_dim, image_dim);
    imshow(im_check);
    title(strcat("Training Data - Image #", num2str(k)));
    set(gca, 'fontsize', 13);
end

% Plot the first 8 images of test data.
for k = 1:8
    subplot(4, 4, k + 8);
    im_check = reshape(im2uint8(dat_test(:, k)), image_dim, image_dim);
    imshow(im_check);
    title(strcat("Test Data - Image #", num2str(k)));
    set(gca, 'fontsize', 13);
end

dat_train = dat_train';
dat_test = dat_test';

% Get training data row wise mean to scale test & train data.
[m, n] = size(dat_train);
[m1, n2] = size(dat_test);
mn = mean(dat_train, 1);
sub1 = repmat(mn, m, 1);
sub2 = repmat(mn, m1, 1);
```

```
% Subtract the mean pixel location from each row.
dat_train = dat_train - sub1;
dat_test = dat_test - sub2;
dat_train = dat_train';
dat_test = dat_test';

% Perform SVD with econ mode.
[U, S, V] = svd (dat_train, 'econ');
lambdas = diag(S).^2;  % Look at singular values.
energies = lambdas ./ sum(lambdas);  % Save energies to plot later.

% Find rank r of the digital space such that 90% energy is met.
% To meet 90% energy - rank of 87 needed which accounts for 90.01% energy.
energy = 0;
total_energy = sum(diag(S).^2);
threshold = 0.9;
r = 0;
while energy <= threshold
    r = r + 1;
    energy = energy + S(r,r)^2 / total_energy;
end

% Plot Energies and Singular Values
figure();
subplot(2, 1, 1);
plot(energies, 'k.', 'markersize', 20);
title('Energies'); xlabel('Mode');
ylabel('Percentage of Energy Captured in Mode');
grid on; set(gca, 'Fontsize', 13);

subplot(2, 1, 2);
plot(lambdas, 'k.', 'markersize', 20);
title('Singular Values'); xlabel('Mode');
ylabel('Singular Value of Mode');
grid on; set(gca, 'Fontsize', 13);


% Look at image reconstruction of low-rank approximation of first 8 images
% in training data and plot first 8 principal components.
figure();
x_train_approx = U(:, 1:r) * S(1:r, 1:r) * (V(:, 1:r)');
for k = 1:8
    subplot(4, 4, k);
    im_check = reshape(im2uint8(x_train_approx(:, k)), image_dim, image_dim);
    imshow(im_check);
    title(strcat("Low Rank Approx. - Image #", num2str(k)));
    set(gca, 'fontsize', 13);
end

% Plot first 8 principal components.
for k = 1:8
    subplot(4, 4, k + 8);
    ut1 = reshape(U(:, k), image_dim, image_dim);
    ut2 = rescale(ut1);
    imshow(ut2);
    title(strcat("Principal Component #", num2str(k)));
    set(gca, 'fontsize', 13);
end

% Plot 3D - Project onto first three V-modes (columns) colored by their
% digit label.
xs = U(:, 1)' * dat_train;
ys = U(:, 2)' * dat_train;
zs = U(:, 3)' * dat_train;

zeros_indices = find(labels_train == 0);
```

```
ones_indices = find(labels_train == 1);
twos_indices = find(labels_train == 2);
threes_indices = find(labels_train == 3);
fours_indices = find(labels_train == 4);
fives_indices = find(labels_train == 5);
sixes_indices = find(labels_train == 6);
sevens_indices = find(labels_train == 7);
eights_indices = find(labels_train == 8);
nines_indices = find(labels_train == 9);

figure();
plot3(xs(zeros_indices), ys(zeros_indices), zs(zeros_indices), 'o', ...
    xs(ones_indices), ys(ones_indices), zs(ones_indices), 'o', ...
    xs(twos_indices), ys(twos_indices), zs(twos_indices), 'o', ...
    xs(threes_indices), ys(threes_indices), zs(threes_indices), 'o', ...
    xs(fours_indices), ys(fours_indices), zs(fours_indices), 'o', ...
    xs(fives_indices), ys(fives_indices), zs(fives_indices), 'o', ...
    xs(sixes_indices), ys(sixes_indices), zs(sixes_indices), 'o', ...
    xs(sevens_indices), ys(sevens_indices), zs(sevens_indices), 'o', ...
    xs(eights_indices), ys(eights_indices), zs(eights_indices), 'o', ...
    xs(nines_indices), ys(nines_indices), zs(nines_indices), 'o');

title('Projection of Training Data onto First Three Modes');
xlabel('Mode 1'); ylabel('Mode 2'); zlabel('Mode 3');
legend('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
grid on; set(gca, 'Fontsize', 13);


%% Part 2: Build Classifiers to Identify Individual Digits in Training Set

% Get data into PCA space - 90% energy captured
project_dat = U(:, 1:r)' * dat_train;
project_dat_test = U(:, 1:r)' * dat_test;

% Build LDA, SVM, and decision tree for every pair of two digits
percent_correct_lda_two_digits_test = [];
percent_correct_svm_two_digits_test = [];
percent_correct_decis_tree_two_digits_test = [];
percent_correct_lda_two_digits_train = [];
percent_correct_svm_two_digits_train = [];
percent_correct_decis_tree_two_digits_train = [];
for i = 0:9
    % For digit i - get the training/test data and their respective labels.
    test_indices_i = find(labels_test == i);
    train_indices_i = find(labels_train == i);
    train_dat_i = project_dat(:, train_indices_i);
    test_dat_i = project_dat_test(:, test_indices_i);
    train_labels_i = i.* ones(length(train_dat_i), 1);
    test_labels_i = i.* ones(length(test_dat_i), 1);

    for j = i+1:9
        % For digit j - get the training/test data and their respective
        % labels.
        test_indices_j = find(labels_test == j);
        train_indices_j = find(labels_train == j);
        train_dat_j = project_dat(:, train_indices_j);
        test_dat_j = project_dat_test(:, test_indices_j);
        train_labels_j = j.* ones(length(train_dat_j), 1);
        test_labels_j = j.* ones(length(test_dat_j), 1);

        % Combine training/test data and labels for digit i and j.
        train_dat_combined = [train_dat_i, train_dat_j];
        test_dat_combined = [test_dat_i, test_dat_j];
        train_labels_combined = [train_labels_i; train_labels_j];
        test_labels_combined = [test_labels_i; test_labels_j];
```

```
        % Perform LDA
        class_test = classify(test_dat_combined', train_dat_combined', ...
            train_labels_combined, 'linear');
        class_train = classify(train_dat_combined', train_dat_combined', ...
            train_labels_combined, 'linear');
        equal_test = class_test == test_labels_combined;
        equal_train = class_train == train_labels_combined;
        number_right_test = sum(equal_test(:) == 1);
        number_right_train = sum(equal_train(:) == 1);
        percent_right_test = number_right_test / (length(test_labels_combined));
        percent_right_train = number_right_train / (length(train_labels_combined));
        percent_correct_lda_two_digits_test = [percent_correct_lda_two_digits_test; ...
            [i, j, percent_right_test]];
        percent_correct_lda_two_digits_train = [percent_correct_lda_two_digits_train; ...
            [i, j, percent_right_train]];

        % Do SVM
        train_dat_combined_trans = train_dat_combined';
        mdl = fitcecoc((1 / max(train_dat_combined_trans(:))).* ...
            train_dat_combined', train_labels_combined);
        test_labels = predict(mdl, (1 / ...
            max(train_dat_combined_trans(:))).* test_dat_combined');
        train_labels = predict(mdl, (1 / ...
            max(train_dat_combined_trans(:))).* train_dat_combined');
        equal_svm_test = test_labels_combined == test_labels;
        equal_svm_train = train_labels_combined == train_labels;
        number_right_svm_test = sum(equal_svm_test(:) == 1);
        number_right_svm_train = sum(equal_svm_train(:) == 1);
        percent_right_svm_test = number_right_svm_test / (length(test_labels_combined));
        percent_right_svm_train = number_right_svm_train / (length(train_labels_combined));
        percent_correct_svm_two_digits_test = [percent_correct_svm_two_digits_test; ...
            [i, j, percent_right_svm_test]];
        percent_correct_svm_two_digits_train = [percent_correct_svm_two_digits_train; ...
            [i, j, percent_right_svm_train]];

        % Decision Tree Learning
        tree = fitctree(train_dat_combined', train_labels_combined);
        test_labels_tree = predict(tree, test_dat_combined');
        train_labels_tree = predict(tree, train_dat_combined');
        equal_tree_test = test_labels_tree == test_labels_combined;
        equal_tree_train = train_labels_tree == train_labels_combined;
        number_right_tree_test = sum(equal_tree_test(:) == 1);
        number_right_tree_train = sum(equal_tree_train(:) == 1);
        percent_right_tree_test = number_right_tree_test / (length(test_labels_combined));
        percent_right_tree_train = number_right_tree_train / (length(train_labels_combined));
        percent_correct_decis_tree_two_digits_test = [percent_correct_decis_tree_two_digits_test; ...
            [i, j, percent_right_tree_test]];
        percent_correct_decis_tree_two_digits_train = [percent_correct_decis_tree_two_digits_train; ...
            [i, j, percent_right_tree_train]];
    end
end

% Build LDA, SVM, and decision tree for every triple of three digits
percent_correct_lda_three_digits_test = [];
percent_correct_svm_three_digits_test = [];
percent_correct_decis_tree_three_digits_test = [];
percent_correct_lda_three_digits_train = [];
percent_correct_svm_three_digits_train = [];
percent_correct_decis_tree_three_digits_train = [];
for i = 0:9
    % For digit i - get the training/test data and their respective labels.
    test_indices_i = find(labels_test == i);
    train_indices_i = find(labels_train == i);
    train_dat_i = project_dat(:, train_indices_i);
    test_dat_i = project_dat_test(:, test_indices_i);
    train_labels_i = i.* ones(length(train_dat_i), 1);
```

```
test_labels_i = i.* ones(length(test_dat_i), 1);

for j = i+1:9
    % For digit j - get the training/test data and their respective
    % labels.
    test_indices_j = find(labels_test == j);
    train_indices_j = find(labels_train == j);
    train_dat_j = project_dat(:, train_indices_j);
    test_dat_j = project_dat_test(:, test_indices_j);
    train_labels_j = j.* ones(length(train_dat_j), 1);
    test_labels_j = j.* ones(length(test_dat_j), 1);

    for k = j+1:9
        % For digit k - get the training/test data and their respective
        % labels.
        test_indices_k = find(labels_test == k);
        train_indices_k = find(labels_train == k);
        train_dat_k = project_dat(:, train_indices_k);
        test_dat_k = project_dat_test(:, test_indices_k);
        train_labels_k = k.* ones(length(train_dat_k), 1);
        test_labels_k = k.* ones(length(test_dat_k), 1);

        % Combine training/test data and labels for digit i, j, and k.
        train_dat_combined = [train_dat_i, train_dat_j, train_dat_k];
        test_dat_combined = [test_dat_i, test_dat_j, test_dat_k];
        train_labels_combined = [train_labels_i; train_labels_j; ...
            train_labels_k];
        test_labels_combined = [test_labels_i; test_labels_j; ...
            test_labels_k];

        % Perform LDA
        class_test = classify(test_dat_combined', train_dat_combined', ...
            train_labels_combined, 'linear');
        class_train = classify(train_dat_combined', train_dat_combined', ...
            train_labels_combined, 'linear');
        equal_test = class_test == test_labels_combined;
        equal_train = class_train == train_labels_combined;
        number_right_test = sum(equal_test(:) == 1);
        number_right_train = sum(equal_train(:) == 1);
        percent_right_test = number_right_test / (length(test_labels_combined));
        percent_right_train = number_right_train / (length(train_labels_combined));
        percent_correct_lda_three_digits_test = [percent_correct_lda_three_digits_test; ...
            [i, j, k, percent_right_test]];
        percent_correct_lda_three_digits_train = [percent_correct_lda_three_digits_train; ...
            [i, j, k, percent_right_train]];

        % Do SVM
        train_dat_combined_trans = train_dat_combined';
        mdl = fitcecoc((1 / max(train_dat_combined_trans(:))).* ...
            train_dat_combined', train_labels_combined);
        test_labels = predict(mdl, (1 / ...
            max(train_dat_combined_trans(:))).* test_dat_combined');
        train_labels = predict(mdl, (1 / ...
            max(train_dat_combined_trans(:))).* train_dat_combined');
        equal_svm_test = test_labels_combined == test_labels;
        equal_svm_train = train_labels_combined == train_labels;
        number_right_svm_test = sum(equal_svm_test(:) == 1);
        number_right_svm_train = sum(equal_svm_train(:) == 1);
        percent_right_svm_test = number_right_svm_test / (length(test_labels_combined));
        percent_right_svm_train = number_right_svm_train / (length(train_labels_combined));
        percent_correct_svm_three_digits_test = [percent_correct_svm_three_digits_test; ...
            [i, j, k, percent_right_svm_test]];
        percent_correct_svm_three_digits_train = [percent_correct_svm_three_digits_train; ...
            [i, j, k, percent_right_svm_train]];

        % Decision Tree Learning
```

```
            tree = fitctree(train_dat_combined', train_labels_combined);
            test_labels_tree = predict(tree, test_dat_combined');
            train_labels_tree = predict(tree, train_dat_combined');
            equal_tree_test = test_labels_tree == test_labels_combined;
            equal_tree_train = train_labels_tree == train_labels_combined;
            number_right_tree_test = sum(equal_tree_test(:) == 1);
            number_right_tree_train = sum(equal_tree_train(:) == 1);
            percent_right_tree_test = number_right_tree_test / (length(test_labels_combined));
            percent_right_tree_train = number_right_tree_train / (length(train_labels_combined));
            percent_correct_decis_tree_three_digits_test =
[percent_correct_decis_tree_three_digits_test; ...
                [i, j, k, percent_right_tree_test]];
            percent_correct_decis_tree_three_digits_train =
[percent_correct_decis_tree_three_digits_train; ...
                [i, j, k, percent_right_tree_train]];
        end
    end
end

% LDA - All Digits
class_test = classify(project_dat_test', project_dat', labels_train, 'linear');
class_train = classify(project_dat', project_dat', labels_train, 'linear');
equal_test = labels_test == class_test;
equal_train = labels_train == class_train;
number_right_test = sum(equal_test(:) == 1);
number_right_train = sum(equal_train(:) == 1);
percent_right_test = number_right_test / length(labels_test);
percent_right_train = number_right_train / length(labels_train);

% SVM - All Digits
project_dat_trans = project_dat';
mdl = fitcecoc((1 / max(project_dat_trans(:))).* project_dat', labels_train);
test_labels = predict(mdl, (1 / max(project_dat_trans(:))).* project_dat_test');
train_labels = predict(mdl, (1 / max(project_dat_trans(:))).* project_dat');
equal_svm_test = labels_test == test_labels;
equal_svm_train = labels_train == train_labels;
number_right_svm_test = sum(equal_svm_test(:) == 1);
number_right_svm_train = sum(equal_svm_train(:) == 1);
percent_right_svm_test = number_right_svm_test / length(labels_test);
percent_right_svm_train = number_right_svm_train / length(labels_train);

% Decision Tree Learning - All Digits
tree = fitctree(project_dat', labels_train);
test_labels_tree = predict(tree, project_dat_test');
train_labels_tree = predict(tree, project_dat');
equal_tree_test = test_labels_tree == labels_test;
equal_tree_train = train_labels_tree == labels_train;
number_right_tree_test = sum(equal_tree_test(:) == 1);
number_right_tree_train = sum(equal_tree_train(:) == 1);
percent_right_tree_test = number_right_tree_test / length(labels_test);
percent_right_tree_train = number_right_tree_train / length(labels_train);
```

mnist_parse.m

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

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

    % 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).';
    end

    % Read in the labels
    labels = fread(fid2, totalImages, 'uint8');

    % Close the files
    fclose(fid1);
    fclose(fid2);

end
```

Appendix C. Additional Figures and Tables

This appendix contains additional tables created in this paper as it relates to comparing how well each method of classification did in separating pairs, triplets, and all ten digits in the MNIST dataset.

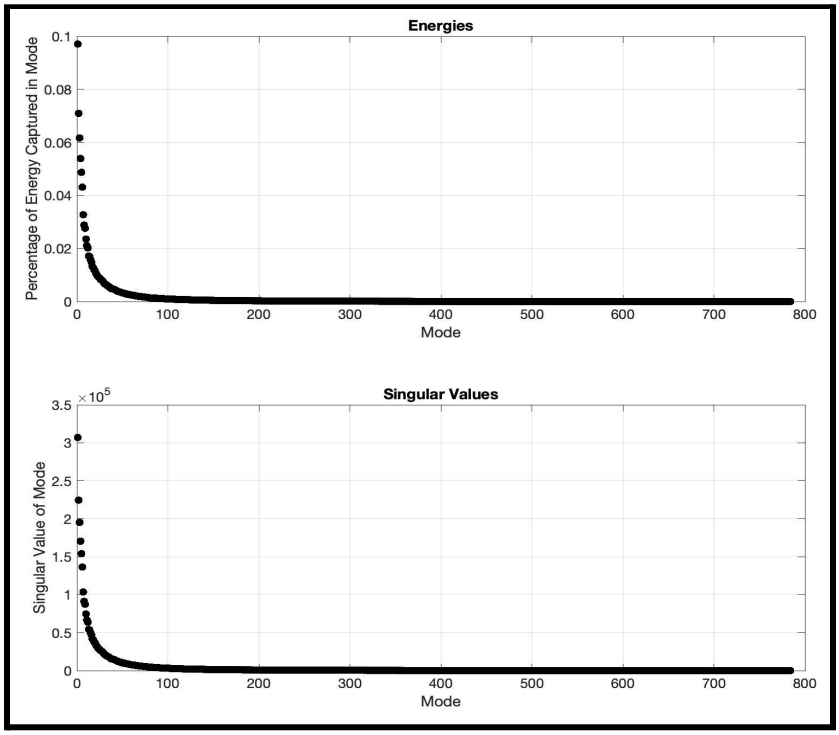

Figure 2: Figure showing the different energies captured in each mode using Equation 3 - as well as the distribution of Singular Values for the training data after performing SVD.

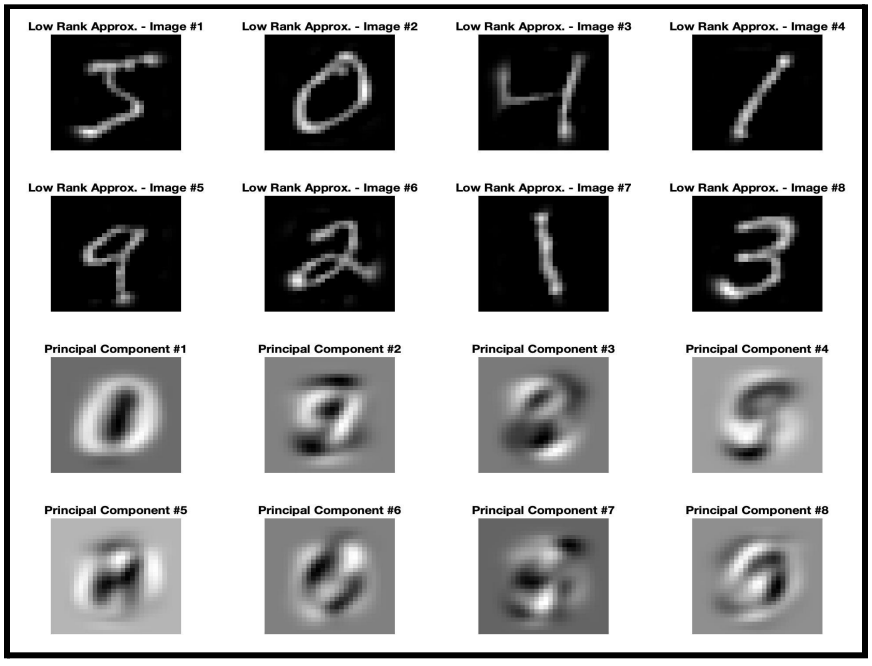

Figure 3: Figure showing the first eight images of the training data after performing a low rank approximation as well as the first eight columns of U after performing SVD (i.e. first eight principal components).

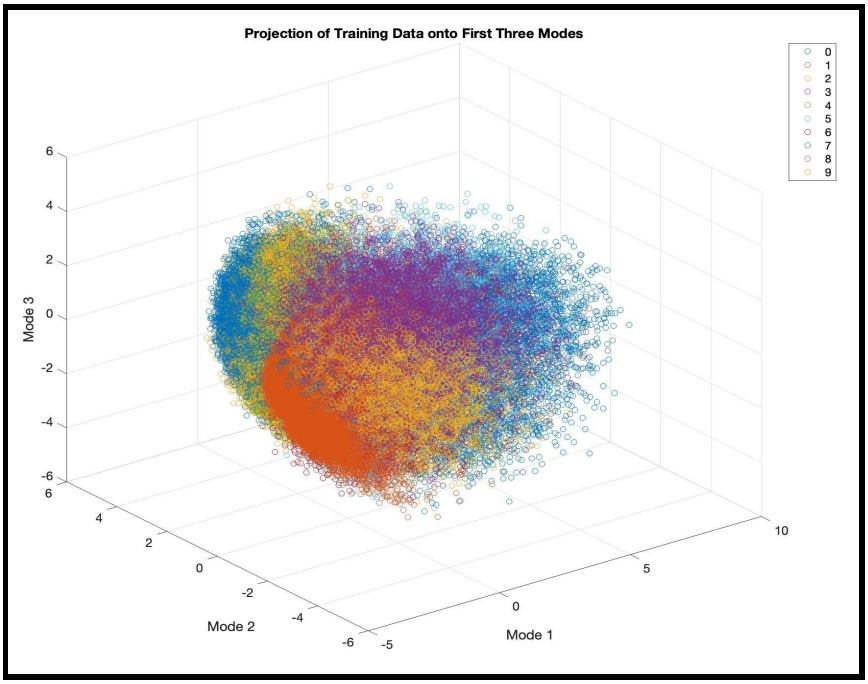

Figure 4: Figure of 3D plot showing the projection of the training data onto the first three columns of U (modes) colored by their digit label.

Two Digits - Test Dataset				
Digit #1	Digit #2	Percentage Correct LDA	Percentage Correct SVM	Percentage Correct DTL
0	1	0.9972	0.9991	0.9967
0	2	0.9871	0.9906	0.9692
0	3	0.995	0.9975	0.9719
0	4	0.9969	0.9985	0.9776
0	5	0.9882	0.9909	0.9621
0	6	0.9897	0.9897	0.9603
0	7	0.993	0.996	0.988
0	8	0.9893	0.9933	0.9734
0	9	0.991	0.9935	0.9754
1	2	0.9843	0.9949	0.9875
1	3	0.9925	0.9963	0.9879
1	4	0.9976	0.9995	0.9957
1	5	0.9936	0.9965	0.9926
1	6	0.9967	0.9967	0.989
1	7	0.9898	0.9926	0.9838
1	8	0.9796	0.9905	0.9844
1	9	0.9958	0.9967	0.9897
2	3	0.9765	0.978	0.951
2	4	0.9796	0.9826	0.9652
2	5	0.9756	0.9797	0.9667
2	6	0.9774	0.9839	0.9704
2	7	0.9786	0.9801	0.9626
2	8	0.9681	0.9781	0.9462
2	9	0.9829	0.9868	0.9711
3	4	0.9925	0.997	0.9769
3	5	0.9632	0.9621	0.9159
3	6	0.9939	0.9954	0.9802
3	7	0.9814	0.9858	0.9715
3	8	0.9602	0.9682	0.9219
3	9	0.9792	0.9846	0.9604
4	5	0.9899	0.9931	0.968
4	6	0.9892	0.9907	0.9691
4	7	0.9841	0.99	0.9602
4	8	0.9908	0.9928	0.9637

4	9	0.9508	0.9684	0.9101
5	6	0.9741	0.9789	0.9611
5	7	0.9906	0.9911	0.9771
5	8	0.9507	0.9582	0.9384
5	9	0.9826	0.9863	0.9553
6	7	0.9975	0.998	0.9849
6	8	0.9871	0.9902	0.9788
6	9	0.9964	0.9964	0.9858
7	8	0.984	0.9895	0.965
7	9	0.9617	0.9627	0.9288
8	9	0.9753	0.9834	0.9481
Best Case		0.9976 - Digits (1, 4)	0.9995 - Digits (1, 4)	0.9967 - Digits (0, 1)
Worst Case		0.9507 - Digits (5, 8)	0.9582 - Digits (5, 8)	0.9101 - Digits (4, 9)
Average Case		0.9838044444	0.9876622222	0.9675466667

Table 1: Table containing the percentage LDA, SVM, and DTL correctly classified the given test data for two digits and how each method of classification did in the best case, worst case, and on average for each pair of digits.

Two Digits - Training Dataset				
Digit #1	Digit #2	Percentage Correct LDA	Percentage Correct SVM	Percentage Correct DTL
0	1	0.9961	0.9993	0.9994
0	2	0.9844	0.9911	0.9966
0	3	0.9919	0.9947	0.9964
0	4	0.9946	0.9969	0.9982
0	5	0.984	0.9882	0.9959
0	6	0.9899	0.9927	0.9952
0	7	0.9962	0.9975	0.9986
0	8	0.9875	0.9937	0.9969
0	9	0.9922	0.9953	0.997
1	2	0.985	0.9931	0.9978
1	3	0.9859	0.9934	0.9978
1	4	0.9952	0.9975	0.9985
1	5	0.9911	0.9968	0.9972
1	6	0.9964	0.9991	0.9987
1	7	0.9921	0.9964	0.998
1	8	0.9667	0.9848	0.9966
1	9	0.9953	0.9965	0.9985
2	3	0.9689	0.973	0.994
2	4	0.9827	0.9874	0.9962
2	5	0.9719	0.9795	0.9948
2	6	0.9762	0.9843	0.9965
2	7	0.9818	0.9887	0.9949
2	8	0.9669	0.975	0.993
2	9	0.9835	0.989	0.9956
3	4	0.9907	0.9963	0.9977
3	5	0.9539	0.9592	0.9898
3	6	0.991	0.9963	0.9964
3	7	0.9848	0.9897	0.9964
3	8	0.962	0.97	0.9907
3	9	0.9776	0.9868	0.9944
4	5	0.9875	0.9912	0.9951
4	6	0.9901	0.993	0.997
4	7	0.9851	0.9907	0.9944
4	8	0.9899	0.9934	0.9946

4	9	0.9563	0.9682	0.9874
5	6	0.9745	0.9812	0.9928
5	7	0.9926	0.9949	0.9958
5	8	0.9581	0.9622	0.992
5	9	0.9835	0.9894	0.9934
6	7	0.9981	0.999	0.9987
6	8	0.9859	0.9906	0.9962
6	9	0.9972	0.9984	0.9981
7	8	0.9877	0.9923	0.9951
7	9	0.9551	0.961	0.9899
8	9	0.9758	0.9843	0.9942
Best Case		0.9981 - Digits (6, 7)	0.9993 - Digits (0, 1)	0.9994 - Digits (0, 1)
Worst Case		0.9539 - Digits (3, 5)	0.9592 - Digits (3, 5)	0.9874 - Digits (4, 9)
Average Case		0.9829733333	0.9882666667	0.9956088889

Table 2: Table containing the percentage LDA, SVM, and DTL correctly classified the given training data for two digits and how each method of classification did in the best case, worst case, and on average for each pair of digits.

Three Digits - Test Dataset					
Digit #1	Digit #2	Digit #3	Percentage Correct LDA	Percentage Correct SVM	Percentage Correct DTL
0	1	2	0.9727	0.9911	0.9739
0	1	3	0.9885	0.9958	0.9725
0	1	4	0.9948	0.9974	0.9835
0	1	5	0.985	0.992	0.9697
0	1	6	0.9873	0.9909	0.9688
0	1	7	0.9844	0.992	0.9796
0	1	8	0.9773	0.9893	0.9728
0	1	9	0.9891	0.9933	0.9773
0	2	3	0.9732	0.9808	0.9404
0	2	4	0.9783	0.9826	0.9492
0	2	5	0.9666	0.979	0.9415
0	2	6	0.9727	0.9791	0.9421
0	2	7	0.973	0.9796	0.9503
0	2	8	0.9635	0.9772	0.932
0	2	9	0.9778	0.9841	0.9626
0	3	4	0.9899	0.9956	0.9623
0	3	5	0.9625	0.9695	0.9244
0	3	6	0.9861	0.9885	0.9556
0	3	7	0.9775	0.9877	0.9556
0	3	8	0.9636	0.9764	0.917
0	3	9	0.9767	0.987	0.9527
0	4	5	0.9797	0.9905	0.9506
0	4	6	0.9812	0.988	0.9524
0	4	7	0.9836	0.992	0.9599
0	4	8	0.9796	0.9911	0.9506
0	4	9	0.9532	0.9737	0.9037
0	5	6	0.9682	0.9774	0.9438
0	5	7	0.9762	0.9893	0.9593
0	5	8	0.9473	0.968	0.9202
0	5	9	0.9719	0.9844	0.9462
0	6	7	0.9811	0.9899	0.9649
0	6	8	0.9825	0.9852	0.9481
0	6	9	0.9827	0.9885	0.9596
0	7	8	0.9782	0.9863	0.95
0	7	9	0.9662	0.9708	0.9304

0	8	9	0.971	0.9821	0.946
1	2	3	0.9685	0.9827	0.9572
1	2	4	0.9724	0.987	0.9768
1	2	5	0.964	0.9846	0.9618
1	2	6	0.9718	0.9862	0.9718
1	2	7	0.969	0.9815	0.9671
1	2	8	0.9586	0.9803	0.9564
1	2	9	0.9754	0.9871	0.9704
1	3	4	0.9901	0.9965	0.9754
1	3	5	0.9661	0.9737	0.9332
1	3	6	0.991	0.9936	0.9684
1	3	7	0.9754	0.9868	0.9694
1	3	8	0.9602	0.974	0.9336
1	3	9	0.9788	0.9857	0.9654
1	4	5	0.9857	0.9953	0.9701
1	4	6	0.9857	0.9915	0.9753
1	4	7	0.9822	0.9908	0.9647
1	4	8	0.9809	0.989	0.9631
1	4	9	0.9661	0.9776	0.9271
1	5	6	0.9755	0.9846	0.9672
1	5	7	0.9787	0.9885	0.9728
1	5	8	0.951	0.9687	0.939
1	5	9	0.9806	0.9891	0.9684
1	6	7	0.9833	0.9923	0.9782
1	6	8	0.9749	0.9873	0.968
1	6	9	0.9894	0.9942	0.9765
1	7	8	0.9704	0.9844	0.9678
1	7	9	0.9644	0.9741	0.9417
1	8	9	0.9689	0.983	0.9567
2	3	4	0.963	0.9742	0.9395
2	3	5	0.9427	0.9543	0.9083
2	3	6	0.9637	0.9753	0.928
2	3	7	0.9534	0.9684	0.9313
2	3	8	0.9416	0.9586	0.8972
2	3	9	0.9577	0.9731	0.9325
2	4	5	0.9663	0.9752	0.9336
2	4	6	0.969	0.9781	0.9468

2	4	7	0.9622	0.9734	0.9356
2	4	8	0.9515	0.9746	0.9173
2	4	9	0.9451	0.9636	0.9176
2	5	6	0.958	0.9691	0.9316
2	5	7	0.957	0.9715	0.9367
2	5	8	0.9379	0.9507	0.9044
2	5	9	0.9635	0.9758	0.9366
2	6	7	0.9655	0.9761	0.9503
2	6	8	0.9541	0.9733	0.9376
2	6	9	0.9727	0.981	0.9597
2	7	8	0.9519	0.971	0.9275
2	7	9	0.9417	0.9635	0.9189
2	8	9	0.9506	0.9705	0.924
3	4	5	0.9636	0.9705	0.9189
3	4	6	0.9827	0.9898	0.9576
3	4	7	0.9728	0.9848	0.9477
3	4	8	0.9616	0.9751	0.9154
3	4	9	0.9494	0.9687	0.8957
3	5	6	0.9552	0.9626	0.9231
3	5	7	0.9522	0.9638	0.9239
3	5	8	0.9186	0.9374	0.8644
3	5	9	0.9543	0.9629	0.9134
3	6	7	0.9736	0.9856	0.9643
3	6	8	0.9619	0.9721	0.93
3	6	9	0.9751	0.9866	0.956
3	7	8	0.9548	0.9691	0.922
3	7	9	0.9514	0.9632	0.9166
3	8	9	0.9499	0.9666	0.9081
4	5	6	0.964	0.9781	0.9371
4	5	7	0.9745	0.9855	0.9407
4	5	8	0.9561	0.9652	0.9175
4	5	9	0.9469	0.9688	0.8862
4	6	7	0.9808	0.9882	0.9525
4	6	8	0.9725	0.9849	0.9434
4	6	9	0.9539	0.9719	0.9081
4	7	8	0.9722	0.9856	0.9403
4	7	9	0.9232	0.9556	0.8808

4	8	9	0.947	0.9646	0.8782
5	6	7	0.968	0.9802	0.9524
5	6	8	0.9448	0.9572	0.9302
5	6	9	0.9654	0.978	0.9451
5	7	8	0.9499	0.9627	0.9174
5	7	9	0.9583	0.9662	0.9215
5	8	9	0.9437	0.9569	0.9113
6	7	8	0.9736	0.9848	0.9514
6	7	9	0.9629	0.9723	0.9362
6	8	9	0.9704	0.9827	0.9408
7	8	9	0.9525	0.9635	0.9107
Best Case			0.9948 - Digits (0, 1, 4)	0.9974 - Digits (0, 1, 4)	0.9835 - Digits (0, 1, 4)
Worst Case			0.9186 - Digits (3, 5, 8)	0.9374 - Digits (3, 5, 8)	0.8644 - Digits (3, 5, 8)
Average Case			0.9670741667	0.97866	0.9423908333

Table 3: Table containing the percentage LDA, SVM, and DTL correctly classified the given test data for three digits and how each method of classification did in the best case, worst case, and on average for each triplet of digits.

Three Digits - Training Dataset					
Digit #1	Digit #2	Digit #3	Percentage Correct LDA	Percentage Correct SVM	Percentage Correct DTL
0	1	2	0.9736	0.9896	0.9968
0	1	3	0.9828	0.9912	0.9951
0	1	4	0.9902	0.9959	0.998
0	1	5	0.9805	0.9898	0.9955
0	1	6	0.9871	0.9943	0.996
0	1	7	0.9873	0.9952	0.9973
0	1	8	0.9664	0.9854	0.9944
0	1	9	0.9883	0.9943	0.9968
0	2	3	0.9632	0.9757	0.9903
0	2	4	0.9759	0.9855	0.9928
0	2	5	0.9608	0.9771	0.9899
0	2	6	0.9691	0.9809	0.9906
0	2	7	0.9739	0.9855	0.9941
0	2	8	0.9607	0.9761	0.9891
0	2	9	0.9758	0.9856	0.9934
0	3	4	0.9837	0.9932	0.994
0	3	5	0.9516	0.9658	0.9875
0	3	6	0.983	0.9905	0.9913
0	3	7	0.979	0.9894	0.9928
0	3	8	0.9594	0.9751	0.9878
0	3	9	0.9734	0.9853	0.9914
0	4	5	0.975	0.9857	0.9918
0	4	6	0.9836	0.9901	0.9922
0	4	7	0.9834	0.9905	0.9931
0	4	8	0.9755	0.9909	0.9928
0	4	9	0.96	0.9746	0.9874
0	5	6	0.9676	0.978	0.9878
0	5	7	0.9788	0.988	0.9947
0	5	8	0.9469	0.9679	0.9869
0	5	9	0.9685	0.984	0.9905
0	6	7	0.9875	0.9934	0.9952
0	6	8	0.9781	0.9864	0.992
0	6	9	0.9856	0.9921	0.9927
0	7	8	0.9768	0.9898	0.9928
0	7	9	0.9633	0.9702	0.9905

0	8	9	0.9686	0.984	0.9916
1	2	3	0.9591	0.9769	0.9923
1	2	4	0.9714	0.9869	0.995
1	2	5	0.9607	0.9833	0.9934
1	2	6	0.9702	0.9863	0.995
1	2	7	0.9724	0.9868	0.9941
1	2	8	0.9452	0.9727	0.9911
1	2	9	0.973	0.9871	0.9942
1	3	4	0.9806	0.9923	0.9956
1	3	5	0.9577	0.9699	0.9922
1	3	6	0.9823	0.9929	0.9949
1	3	7	0.9713	0.9889	0.9945
1	3	8	0.9486	0.9693	0.9896
1	3	9	0.9723	0.9861	0.9936
1	4	5	0.9807	0.9919	0.9957
1	4	6	0.985	0.9937	0.9962
1	4	7	0.9819	0.9911	0.9934
1	4	8	0.9667	0.9853	0.9932
1	4	9	0.9665	0.9773	0.9908
1	5	6	0.9739	0.9861	0.9944
1	5	7	0.9774	0.9931	0.9945
1	5	8	0.9448	0.9689	0.9914
1	5	9	0.976	0.99	0.994
1	6	7	0.9883	0.9964	0.9961
1	6	8	0.9661	0.984	0.9941
1	6	9	0.9908	0.9959	0.9961
1	7	8	0.9616	0.9845	0.9951
1	7	9	0.963	0.9718	0.9913
1	8	9	0.9574	0.9793	0.9928
2	3	4	0.9597	0.9747	0.9898
2	3	5	0.9296	0.9487	0.9858
2	3	6	0.9591	0.9723	0.9889
2	3	7	0.9567	0.9731	0.9895
2	3	8	0.9356	0.9562	0.9817
2	3	9	0.9499	0.9696	0.9907
2	4	5	0.9648	0.9769	0.9895
2	4	6	0.9686	0.9813	0.9919

2	4	7	0.9648	0.9808	0.9905
2	4	8	0.9542	0.9763	0.9888
2	4	9	0.9491	0.9666	0.9858
2	5	6	0.9562	0.9712	0.9906
2	5	7	0.9598	0.9776	0.9896
2	5	8	0.9335	0.9523	0.985
2	5	9	0.9589	0.9749	0.9885
2	6	7	0.97	0.9825	0.9943
2	6	8	0.9544	0.9721	0.9918
2	6	9	0.9739	0.9839	0.9931
2	7	8	0.9563	0.9745	0.9909
2	7	9	0.9424	0.9641	0.9881
2	8	9	0.9472	0.9708	0.9873
3	4	5	0.9538	0.9685	0.9868
3	4	6	0.9802	0.9916	0.9926
3	4	7	0.9722	0.9861	0.9919
3	4	8	0.9626	0.9764	0.9861
3	4	9	0.9491	0.9699	0.9838
3	5	6	0.9493	0.9622	0.988
3	5	7	0.952	0.966	0.9865
3	5	8	0.9147	0.9371	0.9758
3	5	9	0.9455	0.9611	0.9838
3	6	7	0.9788	0.9906	0.994
3	6	8	0.9587	0.9731	0.9874
3	6	9	0.9753	0.9887	0.9934
3	7	8	0.958	0.9734	0.9878
3	7	9	0.9465	0.9634	0.9875
3	8	9	0.9511	0.9678	0.9838
4	5	6	0.963	0.9812	0.9899
4	5	7	0.9768	0.9865	0.9917
4	5	8	0.9558	0.9692	0.9857
4	5	9	0.9488	0.969	0.9822
4	6	7	0.9799	0.9898	0.9927
4	6	8	0.9754	0.9871	0.9919
4	6	9	0.9571	0.975	0.9872
4	7	8	0.9748	0.9855	0.9911
4	7	9	0.9294	0.9539	0.9797

4	8	9	0.9479	0.9662	0.9816
5	6	7	0.9736	0.9851	0.9923
5	6	8	0.9489	0.9614	0.9882
5	6	9	0.9644	0.9821	0.9912
5	7	8	0.9556	0.9694	0.9879
5	7	9	0.9537	0.966	0.9853
5	8	9	0.9477	0.9627	0.9855
6	7	8	0.9789	0.9888	0.9933
6	7	9	0.9611	0.9729	0.9922
6	8	9	0.9695	0.9838	0.991
7	8	9	0.9487	0.9613	0.9861
Best Case			0.9908 - Digits (1, 6, 9)	0.9964 - Digits (1, 6, 7)	0.998 - Digits (0, 1, 4)
Worst Case			0.9147 - Digits (3, 5, 8)	0.9371 - Digits (3, 5, 8)	0.9758 - Digits (3, 5, 8)
Average Case			0.9651108333	0.9792408333	0.9908291667

Table 4: Table containing the percentage LDA, SVM, and DTL correctly classified the given training data for three digits and how each method of classification did in the best case, worst case, and on average for each triplet of digits.

All Ten Digits - Test Dataset		
Percentage Correct LDA	Percentage Correct SVM	Percentage Correct DTL
0.8767	0.9411	0.8435

Table 5: Table containing the percentage LDA, SVM, and DTL correctly classified the given test data for all ten digits.

All Ten Digits - Training Dataset		
Percentage Correct LDA	Percentage Correct SVM	Percentage Correct DTL
0.8701	0.9411	0.9607

Table 6: Table containing the percentage LDA, SVM, and DTL correctly classified the given training data for all ten digits.