Semi-supervised Image Classification and Generation with PCA and K-Means

Micah Reich, David Krajewski*

Carnegie Mellon University School of Computer Science
December 15, 2021

Abstract

Image classification and image generation are long-standing problems in the fields of artificial intelligence and computer science research. While many modern approaches to these tasks make use of deep neural networks like Generative Adversarial Networks and Convolutional Neural Networks, we demonstrate that simple image recognition and generation may be accomplished with classical methods in linear algebra, including Principal Component Analysis and K-means clustering.

1 Introduction

Many modern approaches to image classification and image generation require deep neural networks that must be trained iteratively, often consuming compute power and requiring large amounts of data. For our final project, we wish to demonstrate the power of unsupervised methods in classical machine learning, using PCA for low-rank approximation and embedding-space projections as well as a k-means clustering model for digit classification. Our method is able to correctly classify digits with high (85% to 90%) accuracy, noting the visual similarities between digits; we are also able to take advantage of the low-dimension embedding-space, using the k-means cluster centers and Gaussian noise to create novel MNIST image samples.

2 Methods

2.1 Data Aggregation

Our project made use of the extensive MNIST dataset, which contains 70,000 28x28 images of handwritten digits with their corresponding label. Each of these images have their size normalized and are centered. MNIST is very commonly seen in the world of Machine Learning as a simple way to test classification models. The data is reliable, simple to use, and small enough such that heavy compute power isn't necessarily required. Figure 1 displays an example of the different handwritten digit classes from this dataset.

To make use of these images, we first flatten each of them into a vector of length 784 (i.e. 28×28). We then put the entire dataset into a matrix where each column consists of the flattened version of each image, which makes the computations described in this paper attainable.

*mreich@cmu.edu dkrajews@andrew.cmu.edu

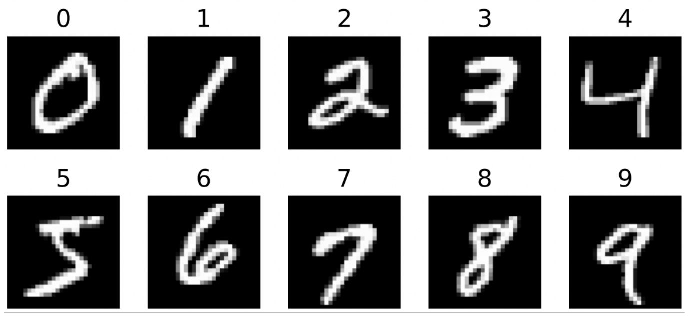


Figure 1: Handwritten Digits ”0” through ”9” From MNIST Dataset

2.2 Principal Component Analysis

2.2.1 SVD and the k-Dimensional Subspace of Best Fit

In order to successfully use clustering algorithms like K-means to classify digits, or to successfully create novel MNIST samples, we wish to embed the data samples into a low-dimensional subspace. Since each sample vector in our dataset is of relatively high dimension (784), an embedding-space model was sought after in order to evade the “Curse of Dimensionality” and overfitting our model when clustering as well as analyze the relative ability of matrix factorization techniques to preserve the important spatial and characteristic information of the images.

The Singular Value Decomposition (SVD) and similar matrix factorization techniques have long been used to accurately approximate high-dimensional sub-spaces using a reduced set of spanning vectors. The rank-r SVD decomposes a rectangular $m \times n$ matrix A into the product of two orthogonal matrices U_r and V_r and one diagonal matrix Σ_r. More generally, the SVD decomposes a rectangular $m \times n$ matrix A into the product of two matrices with orthonormal columns, U and V and one matrix with diagonal entries in the top-left corner with zero padding elsewhere.

$$A = U\Sigma V^T \quad \text{or} \quad A = U_r \Sigma_r V_r^T \quad \text{or} \quad A = \sum_{i=1}^{r} \sigma_i \vec{u}_i \vec{v}_i^T \tag{SVD}$$

The first r columns of U form an orthonormal basis for the column space of A, and the first r columns of V form an orthonormal basis for the row space of A. The r nonzero diagonal entries of Σ are known as the singular values of A. Noting that U and V are matrices with orthonormal columns, U and V act as linear transforms that perform a rotation or a reflection. Noting that Σ has nonzero entries only on the diagonal, Σ acts as a linear transform performing a stretch or dilation. Thus, multiplying a vector $\vec{v} \in \mathbb{R}^n$ by A first rotates the vector, stretches the vector, then re-rotates the vector, decomposing the linear transform that A represents into a sequential series of a rotation (or reflection), dilation, and a rotation (or reflection).

$$A = U\Sigma V^T \implies A\vec{v} = U\Sigma V^T \vec{v}$$

Using the SVD of a matrix A, we wish to find the k-dimensional subspace of best-fit (where $k \leq r$, the rank of matrix A) for the row space or the column space of A. In many applications, the rows of a data matrix A contain observations or data samples, and we thus wish to find a low-dimensional subspace of best fit for the row space of A. That is, we wish to find a subspace that minimizes the perpendicular distance between a vector in the row space and that vector's projection onto the subspace of best-fit. This is synonymous to maximizing the length of the projections of a vector in the row space on to the subspace of best-fit.

Using the SVD, we may choose the appropriate vectors that fit such constraints and maximize the appropriate metrics. For a data matrix with samples in each row, the k-dimensional subspace of best fit is spanned by the set of orthonormal vectors $V_k = \{\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_k\}$ (Strang 7.2, 7.3). Similarly, for a data matrix with samples in each column, the k-dimensional subspace of best fit is spanned by the set of orthonormal vectors $U_k = \{\vec{u}_1, \vec{u}_2, \cdots, \vec{u}_k\}$. Thus, we may also achieve the best low-rank approximation (as judged by the Frobenius Norm) for a matrix A by considering the first k singular vectors and singular values of the SVD of A.

$$A \approx U_k \Sigma_k V_k^T = \sum_{i=1}^{k} \sigma_i \vec{u}_i \vec{v}_i^T$$

Principal Component Analysis (PCA) uses the first k singular vectors of a data matrix A to determine the direction of maximum variance in a dataset and thus best approximate the dataset characteristics. In a traditional k-dimensional subspace of best fit problem, the data matrix is often left unaltered; in PCA, the data matrix is normalized in order to center the data in each observation, with a row-wise or column-wise zero mean so that the principal components (the singular vectors of the normalized data matrix) best capture data's variance.

2.2.2 PCA and Embedding Spaces

In order to reduce the dimensionality of our MNIST samples, projecting them into an embedding space, we first find the k spanning principal components of the dataset (experimenting with different values of k throughout), forming a basis for our low-dimensional subspace. We may then project our MNIST samples in our data matrix down to the low-dimensional subspace of best fit; since we have obtained an orthonormal basis for our subspace of best-fit, the spanning principal components may be projected onto using a simple matrix product.

P_s is a $k \times s$ matrix whose s columns are vectors in $\mathbb{R}^k$, the projections of each MNIST sample onto the subspace of best fit. U_k is the matrix of spanning principal components whose k columns are vectors in $\mathbb{R}^{784}$. D is the data matrix whose s columns are vectors in $\mathbb{R}^{784}$, with each column as an MNIST sample. [1]

Let $\{\vec{u}_1, \vec{u}_2, \cdots, \vec{u}_k\}$ be the spanning set of principal components of the k-dimensional subspace of best-fit. We place these principal components into the columns of a matrix U_k.

$$\text{proj}_{U_k}\vec{x} = U_k(U_k^T U_k)^{-1} U_k^T \vec{x} \quad \text{and} \quad U_k^T U_k = I \implies \text{proj}_{U_k}\vec{x} = U_k U_k^T \vec{x}$$

We see that $U_k U_k^T$ is our projection matrix on to the subspace spanned by the principal components. We may now project all data samples on to this subspace using the projection matrix, transforming our samples into vectors in the subspace spanned by the columns of U_k. We then perform a change-of-basis to get the coordinates of the projected vectors written in the principal component basis, thus transforming vectors once in $\mathbb{R}^{784}$ into vectors in $\mathbb{R}^k$.

$$P_s^T = D^T U_k U_k^T$$

We have thus embedded our data into a lower-dimensional vector subspace of $\mathbb{R}^k$. We may now perform clustering and other operations on the embedded vectors. As shown in figure 2, the vectors of digits that are often written similar to one another are closely packed when

[1] We store samples in the columns of the data matrix, performing PCA for columns.

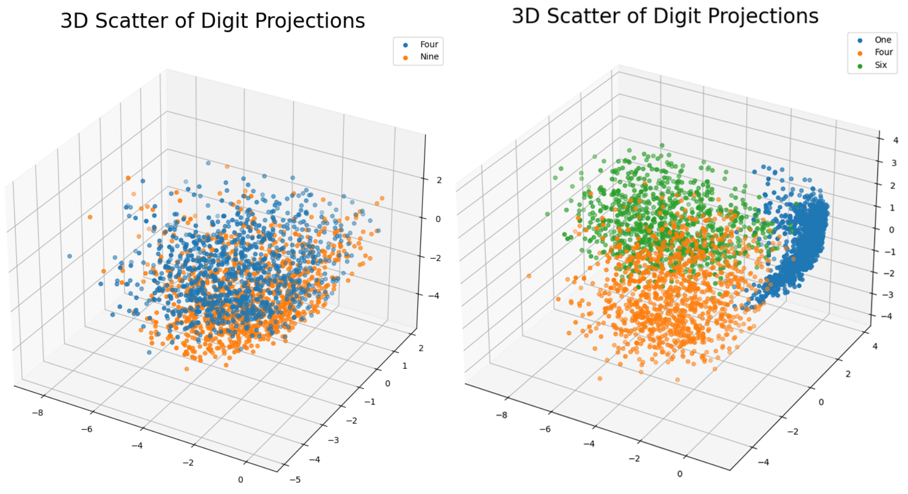


Figure 2: Scatter plot of projected image vectors in $\mathbb{R}^3$

projected down to the subspace. For example, ones are closely packed, and fours and nines are closely interspersed since images of ones lack variation in their styles while fours and nines are visually similar.

2.3 K-Means Clustering

Once we have determined the embedding space for our samples and projected our dataset to a low-dimension subspace of best fit, we wish to perform clustering on the samples to determine the digit class correspondences of the data. The clustering problem is one which seeks to partition a dataset into distinct classes that have intracluster similarity and intercluster disparity. K-means seeks to achieve this by optimizing (minimize) the objective function J, where k is the number of clusters, $\vec{\mu}_i$ is the center of cluster k, and each $\vec{x}_j$ is a sample in the data set corresponding to cluster i, S_i.

$$J = \sum_{i=0}^{k} \sum_{x \in S_i} \|\vec{x} - \vec{\mu}_i\|^2$$

That is, k-means seeks to find each $\vec{\mu}_i := \arg\min \sum_{x \in S_i} \|\vec{x} - \vec{\mu}_i\|^2$; we seek to find k cluster centers that minimize the sum of the squared distances between each point assigned to cluster i and the center of the cluster $\vec{\mu}_i$. This objective function is minimized iteratively. We often start by picking each $\vec{\mu}_i$ randomly by placing them among the sample data points, assigning each data point to the closest cluster center. After each training iteration, k-means updates the positions of each center to further minimize the objective function J. Note that the cost of the model is evaluated using the euclidean distance between each point in a given cluster and that cluster's center.

Training of the k-means model stops once either the maximum number of training iterations has been reached or when a local minimum has been achieved, as each new training iteration does not change the cluster center positions. After training is complete, k-means yields the k cluster centers and each data sample's assigned cluster determined by the closest center to each data point.

Since k-means does not use the labels of the data points to determine which cluster corresponds to which digit class, we perform these assignments after training. We do this by using the data's labels and taking the most frequently occurring digit within each cluster to be the digit class that represents its parent cluster. This way, when a novel unlabeled digit is presented to the model, we may place it in the appropriate cluster and assign it a label based on its cluster membership. Note that we perform clustering within the low-dimensional subspace of best fit, as considering only the most significant sources of variation in the dataset yields a more robust model that is less likely to overfit to the data that can better generalize.

3 Results

Using PCA and K-means clustering, we were able to perform image classification and image generation; we wish to display our results, identify some shortcomings of our models, and elaborate on how future models might better account for the discrepancies outlined below.

3.1 Similarity Scores

In order to evaluate the success of our PCA projection model, we wish to determine how "similar" are projected vectors within the same versus different digit classes, as we expect a successful model to retain the similarity of image vectors within the same class when projected down to the embedding space while keeping images of different classes distinguishable.

While many evaluation metrics exist to quantify the similarity of vectors, we choose to perform our evaluation based on the cosine similarity or dot product similarity of the projected image vectors. The cosine similarity s between two vectors $\vec{x}$ and $\vec{y}$ is defined as

$$s = \frac{\vec{x}^T \vec{y}}{||x|| \cdot ||y||}$$

which evaluates to the cosine of the angle between the two vectors, with similar vectors having an angle near $0 \implies s \approx 1$, orthogonal vectors having an angle near $\pi/2 \implies s \approx 0$, and vectors in opposite directions having an angle near $\pi \implies s \approx -1$. In figure 3 we show the cosine similarities between each pairwise bin. [2] Note that, as expected, the cosine similarities between each bin and itself are the highest. Unfortunately, the similarities of different numbers that are visually disparate are not as low as we had hoped, likely due to the large amount of blackspace that is embedded within each image vector, placing all vectors within some close neighborhood of one another in the low-dimensional subspace, meaning the cluster centers are closer than we would hope. Relatively, however, we do notice that similar-looking digits like nines and fours or eights and fives have very high similarity with respect to self-similarity scores.

3.2 Image Classification

Once we have projected our samples into a lower-dimensional subspace, we used a k-means clustering algorithm to group the projected images into bins corresponding to their digit values. In order to evaluate the accuracy of our model, we began by training the k-means model, then evaluating the bins assigned to each sample in the dataset. In order to determine which digit corresponded to each bin, we simply assigned each bin the most probable digit by taking the most frequently occurring digit class in a fixed bin to be the bin's final digit representation.

[2] The similarities are averaged, as we compute the cosine similarity between each pairwise bin for all bins corresponding to the digit pair

	0	1	2	3	4	5	6	7	8	9
0	**0.818**	0.342	0.601	0.626	0.522	0.659	0.641	0.515	0.648	0.551
1		**0.774**	0.568	0.546	0.428	0.516	0.469	0.479	0.61	0.476
2			**0.823**	0.675	0.613	0.618	0.7	0.561	0.738	0.595
3				**0.876**	0.551	0.73	0.6	0.582	0.764	0.605
4					**0.845**	0.602	0.651	0.641	0.685	0.757
5						**0.775**	0.61	0.595	0.767	0.651
6							**0.813**	0.546	0.671	0.639
7								**0.764**	0.663	0.71
8									**0.894**	0.72
9										**0.832**

Figure 3: Averaged similarities between centers of bins corresponding to each digit

The performance of k-means clustering models depends on the variance of the data set and the number of cluster bins available to the model. In general, as shown by empirical tests on the accuracy of our model in figure 4, a greater number of bins corresponds to higher clustering accuracy as the model is able to group together more specific categories of items in the data set. For example, in the case of MNIST, given only ten bins (the number of digit classes) to use, the k-means model suffers in accuracy, as it is forced to generalize across all digit classes, grouping all fours, nines, etc. together, even though fours often appear written in two distinct ways.

In our final model, k-means correctly classified 85% of samples when using 75 cluster bins. As shown below, this accuracy improves with increasing bin count, reaching up to 90% accuracy with > 200 bins, but such a large bin count introduces the problem of model over-fitting.

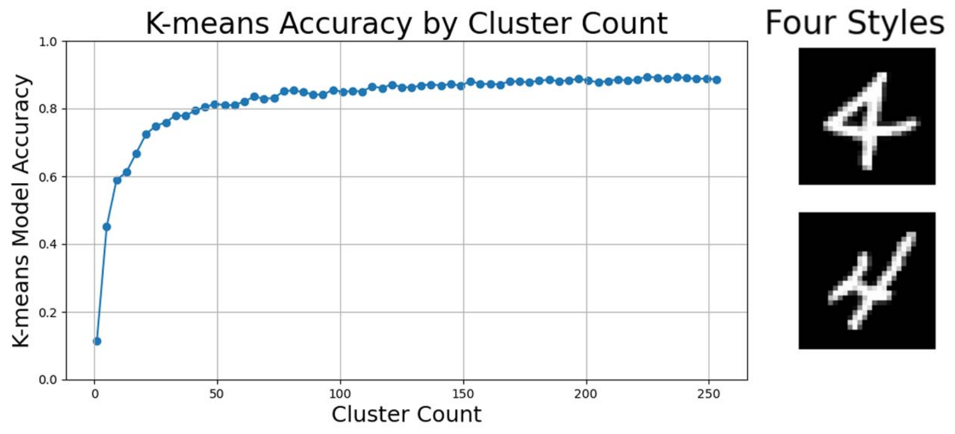


Figure 4: (left) K-means accuracy versus bin count, (right) Different handwritten fours

3.3 Novel Sample Generation

The final objective of our model was novel sample generation. That is, we sought to generate novel images of MNIST digits based on the coherence of our embedding space, drawing samples from each digit cluster within the low-dimension subspace of best fit to then convert back to a 28×28 image. Such sampling from an embedding space or latent space is a common technique in the training of Generative Adversarial Networks (GAN) that learn a latent representation of their image domain through iterative training methods. In a GAN architecture, each novel sample is simply a random Gaussian noise vector that, as training progresses, comes to represent a salient point in the learned latent space.

Similar to the techniques used in GANs, we sought to somewhat randomly sample from our "learned" embedding space in order to create a novel image sample. To do this, we simply based our sample on the cluster centers of each digit class. Indeed, the cluster centers of each digit class by themselves represent novel points in the embedding space corresponding to a given digit since (likely) no image from the dataset falls exactly in the cluster center. However, this method only yields a few novel samples, since cluster centers are fixed and without variation after training a k-means model. Thus, we took inspiration from a GAN and added random Gaussian noise to the cluster center vector at each run of the image generation program, representing a nearly infinite range of novel MNIST images. We empirically decided the degree to which Gaussian noise was added to the center vector, since the proximity of the cluster centers meant adding too much noise would place the embedded vector into a cluster corresponding to a different digit. The likelihood that this digit-shift error occurred was highest on digits with the closest cluster centers (digits that look alike), such as fours and nines or fives and eights.

In order to transform the embedded vectors into images, we construct a low-rank approximation of the noisy cluster center data using the same PCA model as before. Figure 5 shows some example noisy reconstructions of digits taken from the Gaussian-noised cluster center embedded vectors.

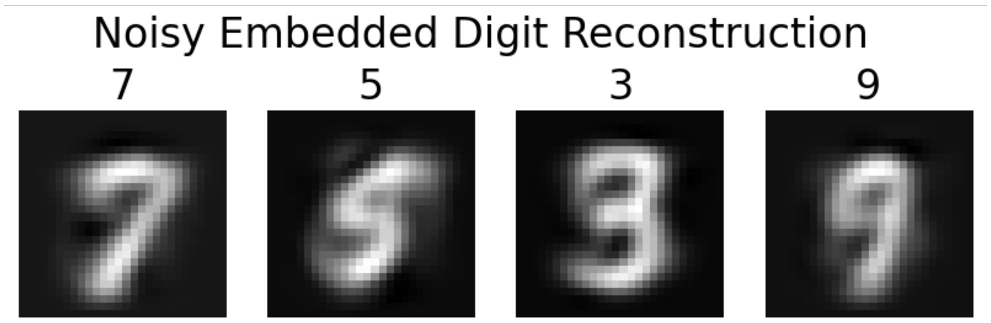


Figure 5: Digits seven, five, three, and nine reconstructed as part of a low-rank approximation of k-means cluster centers with added Gaussian-noise

4 Conclusions

In this paper, we have demonstrated that Principal Component Analysis and K-means clustering is sufficient to perform semi-supervised image classification and may be easily adapted to act as an elementary generative network as demonstrated through use of the MNIST handwritten digit image dataset. Using the k-dimensional subspace of best fit and PCA, we first project our dataset down to a lower-dimensional subspace known as the embedding space, using the first k principal components as the spanning set of the subspace to ensure that the subspace captures the maximal variation of the data. We then perform k-means clustering on the embedded vectors, placing each sample into a bin corresponding to a digit class in order to perform classification, using the labels of the dataset to determine the correspondence of each bin; finally, we may add a Gaussian noise vector to each cluster center and construct a low-rank approximation for the noisy cluster centers, thus producing novel image samples.

Working on this final project was informative and a helpful exercise in real-life applications of linear algebra content. Most notably, we learned how to work with the principal component analysis algorithm and projection matrices, as we were required to project our sample images to a lower-dimensional subspace determined by the principal components of the data matrix, operations which require knowledge of least-squares and the SVD. Working closely with data taught us how to appropriately structure our data matrices and use transposes and left or right multiplication by projection matrices to perform operations on the rows and columns of a given matrix. We also learned how to simplify matrix products when working with matrices with orthonormal columns (like the principal component matrix), taking advantage of the fact that $U^T U = I$; for example, figuring out the expression for the projected data samples using the principal component projection matrix required us to transpose our data matrix and simplify our projection using properties of orthogonal matrices.

We also learned how clustering algorithms like K-means work, an informative experience since k-means is an iterative algorithm, contrasting with previously learned, closed-form algorithms to problems like least squares or best-fit subspaces; during our research, we also discovered that iterative algorithms/models like Support Vector Machines or other machine learning architectures are able to more readily solve classification problems while deep neural networks like GANs are able to perform generative tasks extremely well.

References

[1] Strang, Gilbert. Introduction to Linear Algebra, Fifth Edition. Wellesley-Cambridge Press, 2016.

[2] Sorzano, Carlos Oscar Sánchez, Javier Vargas, and A. Pascual Montano. “A survey of dimensionality reduction techniques.” arXiv preprint arXiv:1403.2877 (2014).

[3] Zhao, Tuo. “Lecture 7: Unsupervised Learning” Georgia Institute of Technology. Atlanta, Georgia. Lecture.

[4] Gormley, Matt. “Principal Component Analysis and Dimensionality Reduction” Carnegie Mellon University. Pittsburgh, Pennsylvania. Lecture.

[5] Li, Lorraine. “Principal Component Analysis for Dimensionality Reduction.” Medium, Towards Data Science, 27 May 2019, https://towardsdatascience.com/principal-component-analysis-for-dimensionality-reduction-115a3d157bad.

[6] Offner, David. 21-241 Matrices and Linear Transformations Lectures 27, 28, 29, 30, 31. 2021. Carnegie Mellon University. Pittsburgh, Pennsylvania. Lecture.

5 Appendix: Code

Below we have provided the code written in Julia 1.6.2 required to reproduce our results. *In our experiments, we use an embedding dimension of 16 and a k-means bin size of 75 constrained to 20 iterations maximum.*

Semi-supervised Image Classification and Generation with PCA and K-Means

By Micah Reich and David Krajewski

Perform necesarry imports

In [1]:
```
using MLDatasets, LinearAlgebra, Clustering, ImageCore, Images, MultivariateStats, StatsBase, Noise, PyPlot, Plots
```

Load in dataset

In [3]:
```
# load full training set
train_x, train_y = MNIST.traindata()

# load full test set
test_x,  test_y  = MNIST.testdata()

println("Training set size: " * string(size(train_x)))
println("Testing set size: " * string(size(test_x)))
```

```
Training set size: (28, 28, 60000)
Testing set size: (28, 28, 10000)
```

Reformat training data into a matrix of flattened vectors where each column is a single image. Create a function to turn a flattened vector back into a 2D image.

In [5]:
```
# turn training /testing sets into data matrix
reshaped_train_x = reshape(train_x, (28*28, size(train_x)[3]))
reshaped_test_x = reshape(test_x, (28*28, size(test_x)[3]))

println("Training set size: " * string(size(reshaped_train_x)))
println("Testing set size: " * string(size(reshaped_test_x)))

# turn flat vectors to 2D images
function reshape_flattened_vector(flat_vector)
    n_rows = 28
    n_cols = 28

    # reshape flat vector to 2d image, for some reason img gets
    # flipped along diagonal when reshaping so we transpose it
    return transpose(reshape(flat_vector, (n_rows, n_cols)))
end
```

```
Training set size: (784, 60000)
Testing set size: (784, 10000)
```

Out[5]:
```
reshape_flattened_vector (generic function with 1 method)
```

This method will give us all the images for a given digit. For example, this is useful when we want to pull all the images containing 4's, etc.

In [6]:
```
# get samples of a certain digit class from a data matrix
function get_mnist_digitclass(digitclass, data, labels)
    digitclass_vec = Array{Float64}(undef, 0, size(data)[1])

    for i = 1:size(data)[2] # number of samples
        if(labels[i] == digitclass)
            digitclass_vec = [ digitclass_vec; transpose(data[:,i]) ]
        end
    end

    return transpose(digitclass_vec)
end
```

Out[6]:
```
get_mnist_digitclass (generic function with 1 method)
```

Use SVD to perform low-rank approximation

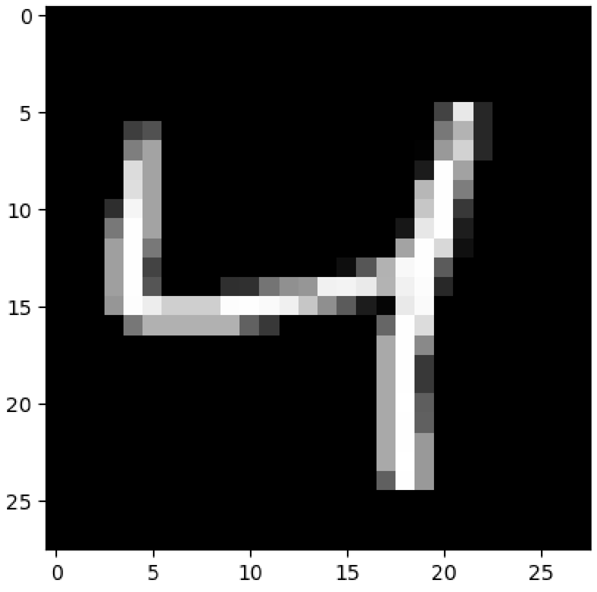
In [12]:
```
test_svd = svd(MNIST.traintensor(3))
```

```
println("Singular values: ", test_svd.S)
```

```
Singular values: Float32[5.744073, 3.4638755, 2.7429197, 1.8240664, 1.4560202, 0.66548216, 0.58808184, 0.5685397,
0.3188651, 0.22387388, 0.19487429, 0.04799972, 0.028930677, 2.6585892f-7, 2.6585892f-7, 2.6585892f-7, 2.6585892f-
7, 2.6585892f-7, 2.6585892f-7, 2.6585892f-7, 2.6585892f-7, 2.6585892f-7, 2.6585892f-7, 2.6585892f-7, 2.6585892f-
7, 2.6585892f-7, 2.6585892f-7, 1.8930024f-13]
```

Plot the SVD representation of our digit

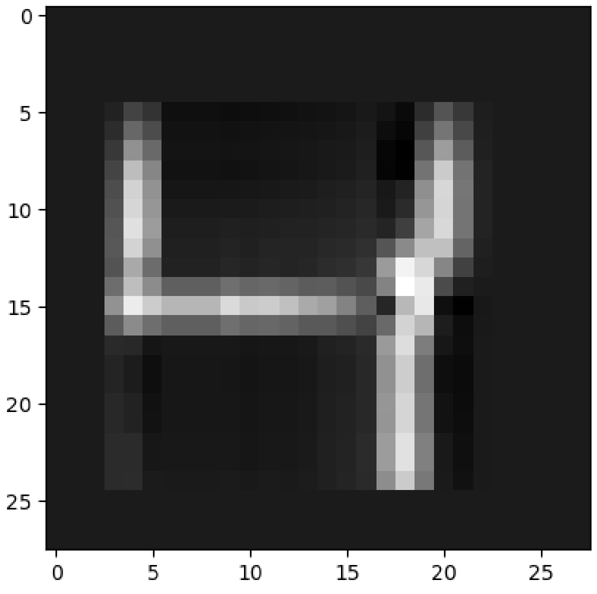
In [15]:
```
reconstructed_test_svd = transpose((test_svd.U * Diagonal(test_svd.S) * test_svd.Vt))

PyPlot.imshow(
    reconstructed_test_svd, cmap="gray"
)
```



Out[15]:
```
PyObject <matplotlib.image.AxesImage object at 0x7f9ff8114790>
```

Using "low rank" approximation for images. We grab the first 3 singular elements from our SVD to show that the digit is still recognizable even with significantly less data

In [14]:
```
# rank 3 approximation for test_svd
test_svd_3 = transpose(test_svd.U[:, 1:3] * Diagonal(test_svd.S[1:3]) * test_svd.Vt[1:3, :])

PyPlot.imshow(
    test_svd_3, cmap="gray"
)
```



Out[14]:
```
PyObject <matplotlib.image.AxesImage object at 0x7fa04832e970>
```

Perform PCA for dimensionality reduction

Helper functions that perform the actual PCA training, displaying sample images and displaying the 3D scatter plots of the projections on to the low dimensional subspace

In [17]:
```
function pca_m(data, n_reduced_dims)
    data = Float64.(data)

    pca_model = fit(PCA, data; maxoutdim=n_reduced_dims)

    princ_vecs = transform(pca_model, data)
    proj_mat = projection(pca_model)
    reconstructed = reconstruct(pca_model, princ_vecs)

    return (pca_model, princ_vecs, proj_mat)
end

function show_sample_img(data, sample_no)
    data_sample = data[:,sample_no]
    return data_sample_reshaped = reshape_flattened_vector(data_sample)
end

function display_proj(proj_coords, r_start, r_end, color, label)
    (x1, y1, z1) = (proj_coords[r_start:r_end, 1],
                    proj_coords[r_start:r_end, 2],
                    proj_coords[r_start:r_end, 3])

    return scatter3D(x1, y1, z1, color = color, label = label, s = 60)
end

function get_digit_projections(data, labels, digit_class)
    for i = 1:size(labels)[1]
        if labels[i] == digit_class
            println(labels[i])
        end
    end
end
```

Out[17]:
```
get_digit_projections (generic function with 1 method)
```

Helper functions that project samples onto the low-dimensional subspace and represent the projected samples as k-dimensional vectors in the principal component basis

In [25]:
```
function least_squares(A, b)
    return inv(transpose(A) * A) * transpose(A) * b
end

function kdim_subspace_proj(basis_matrix, samples)
    # project samples on to k-dimension subspace of best fit spanned by
    # the vectors in the columns of the basis matrix
    proj_matrix = basis_matrix * inv(transpose(basis_matrix)*basis_matrix) * transpose(basis_matrix)

    # project columns of samples onto columnspace of basis matrix
    proj_samples = proj_matrix * samples

    return proj_samples
end

function kdim_proj_coords(basis_matrix, projected_samples)
    # get coordinates of projected samples within new kdim subspace, expect a k-dim vector out
    return least_squares(basis_matrix, projected_samples)
end
```

Out[25]:
```
kdim_proj_coords (generic function with 1 method)
```

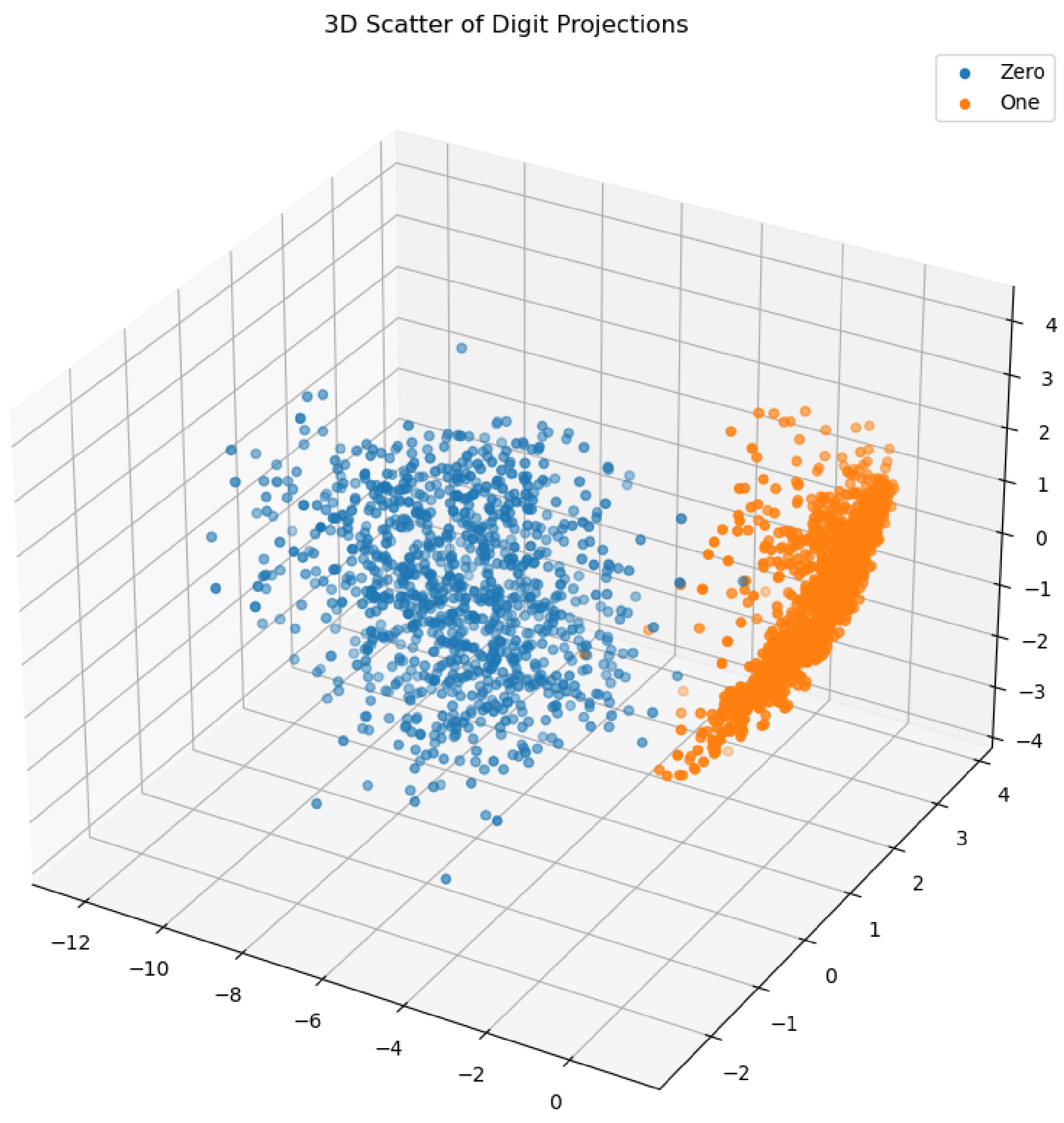
In [19]:
```
# determine embedding dimension, train PCA on mnist samples
embedding_dim = 16
pca_model, princ_vecs, pca_proj_mat = pca_m(reshaped_test_x, embedding_dim)

println("Proj Matrix: " * string(size(pca_proj_mat)))

# get projection coordinates of samples onto k-dim subspace of best fit
kdim_proj_vecs = transpose(reshaped_test_x) * pca_proj_mat
kdim_proj_vecs = transpose(kdim_proj_vecs)
```

```
println("Projection coordinates: " * string(size(kdim_proj_vecs)))
```

```
Proj Matrix: (784, 16)
Projection coordinates: (16, 10000)
```

In [20]:

```
# get coordinates of each digit class
p0 = get_mnist_digitclass(0, kdim_proj_vecs, test_y)
p1 = get_mnist_digitclass(1, kdim_proj_vecs, test_y)
p2 = get_mnist_digitclass(2, kdim_proj_vecs, test_y)
p3 = get_mnist_digitclass(3, kdim_proj_vecs, test_y)
p4 = get_mnist_digitclass(4, kdim_proj_vecs, test_y)
p5 = get_mnist_digitclass(5, kdim_proj_vecs, test_y)
p6 = get_mnist_digitclass(6, kdim_proj_vecs, test_y)
p7 = get_mnist_digitclass(7, kdim_proj_vecs, test_y)
p8 = get_mnist_digitclass(8, kdim_proj_vecs, test_y)
p9 = get_mnist_digitclass(9, kdim_proj_vecs, test_y)

# plot coordinates of each digit class
fig = figure(figsize=(10,10))

p0_plot = scatter3D(p0[1, :], p0[2, :], p0[3, :], label = "Zero", s = 20)
p1_plot = scatter3D(p1[1, :], p1[2, :], p1[3, :], label = "One", s = 20)
# p2_plot = scatter3D(p2[1, :], p2[2, :], p2[3, :], label = "Two", s = 20)
# p3_plot = scatter3D(p3[1, :], p3[2, :], p3[3, :], label = "Three", s = 20)
# p4_plot = scatter3D(p4[1, :], p4[2, :], p4[3, :], label = "Four", s = 20)
# p5_plot = scatter3D(p5[1, :], p5[2, :], p5[3, :], label = "Five", s = 20)
# p6_plot = scatter3D(p6[1, :], p6[2, :], p6[3, :], label = "Six", s = 20)
# p7_plot = scatter3D(p7[1, :], p7[2, :], p7[3, :], label = "Seven", s = 20)
# p8_plot = scatter3D(p8[1, :], p8[2, :], p8[3, :], label = "Eight", s = 20)
# p9_plot = scatter3D(p9[1, :], p9[2, :], p9[3, :], label = "Nine", s = 20)

title("3D Scatter of Digit Projections")
legend(loc="upper right")
PyPlot.savefig("3d-scatter-proj.png")
```



K-means clustering for classification

Helper functions to judge the accuracy of the k-means model as well as functions to get the corresponding digit class for each cluster bucket

In [24]:
```
function acc_score(pred, truth)
    score = 0
    for i = 1:length(pred)
        if(pred[i] == truth[i])
            score += 1
        end
    end

    return score / length(pred)
end

function get_corr_labels(arr, index)
    indexed_arr = Array{Float64}(undef, 0)

    for i = 1:length(arr)
        if index[i] == 1
            indexed_arr  = [indexed_arr; arr[i]]
        end
    end

    return indexed_arr
end
```

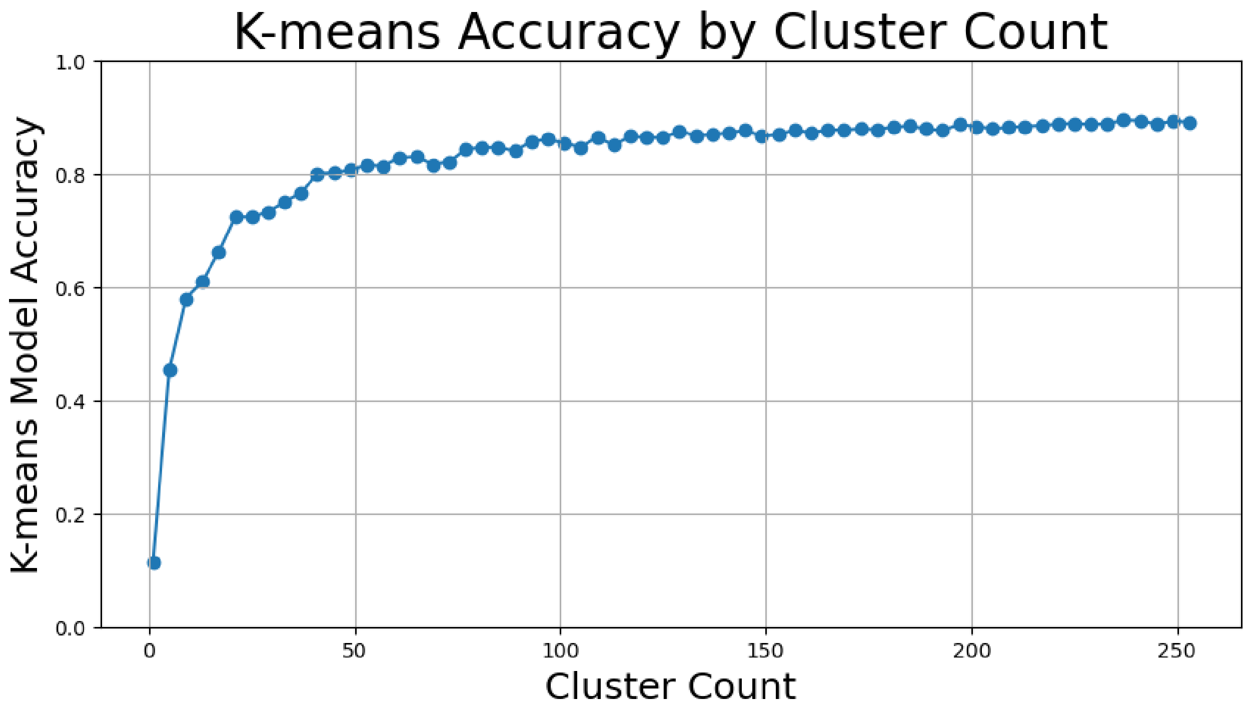
```
# gets most probable ground truth label corresponding to each bucket
# so that we know what cluster corresponds to what number
function get_cluster_assoc(cluster_labels, truth_labels)
    reference_labels = Dict()
    unique_labels = countmap(cluster_labels)

    for i = 1:length(unique_labels)
        index = [ifelse(x == i, 1, 0) for x in cluster_labels]
        indexed_truth_labels = get_corr_labels(truth_labels, index)

        counted_indexed_truth_labels = countmap(indexed_truth_labels)
        reference_labels[i] = Int(findmax(counted_indexed_truth_labels)[2])
    end

    return reference_labels
end
```

Out[24]:
```
get_cluster_assoc (generic function with 1 method)
```

The code below performs the k-means clustering and prints out the dimensions of the projection matrix, the associated value with each clustering bin from our k-means, and the total accuracy our k-means clustering algorithm had. We also look at the accuracy of our k-means model with respect to the number of clusters we allow it to use and plot the data

In [51]:
```
# perform k-means with different bin sizes and plot accuracy as bin size increases

acc_bins = Array{Float64}(undef, 0)
bin_size = Array{Float64}(undef, 0)

for i = 1:4:256
    R_i = kmeans(kdim_proj_vecs, i; maxiter=20, init=:kmpp)
    a_i = assignments(R_i)

    # mappings between clustering bins and which number they represent
    assoc_vals = get_cluster_assoc(a_i, test_y)
    predicted_labels = [Int(assoc_vals[bin]) for bin in a_i]

    acc_bins = [acc_bins; acc_score(predicted_labels, test_y)]
    bin_size = [bin_size;i]
end
```

In [52]:
```
fig = figure(figsize=(10,5))

xlabel("Cluster Count", size=18)
ylabel("K-means Model Accuracy",size=18)

PyPlot.scatter(bin_size, acc_bins)
PyPlot.plot(bin_size, acc_bins)

title("K-means Accuracy by Cluster Count", size = 24)
PyPlot.grid("on")
ylim(0, 1)

PyPlot.savefig("kmeans-performance.png")
```



In [91]:

```
# perform actual k-means clustering
R = kmeans(kdim_proj_vecs, 75; maxiter=20, init=:kmpp)
a = assignments(R)
c = counts(R)

# projection dims of projected samples onto subspace
println("Projections: " * string(size(kdim_proj_vecs)))

# mappings between clustering bins and which number they represent
assoc_vals = get_cluster_assoc(a, test_y)
predicted_labels = [Int(assoc_vals[bin]) for bin in a]
println("Accuracy: " * string(acc_score(predicted_labels, test_y)))

# print which bins correspond to which digit classes
for x in sort(collect(zip(values(assoc_vals),keys(assoc_vals)))) println(x) end
```

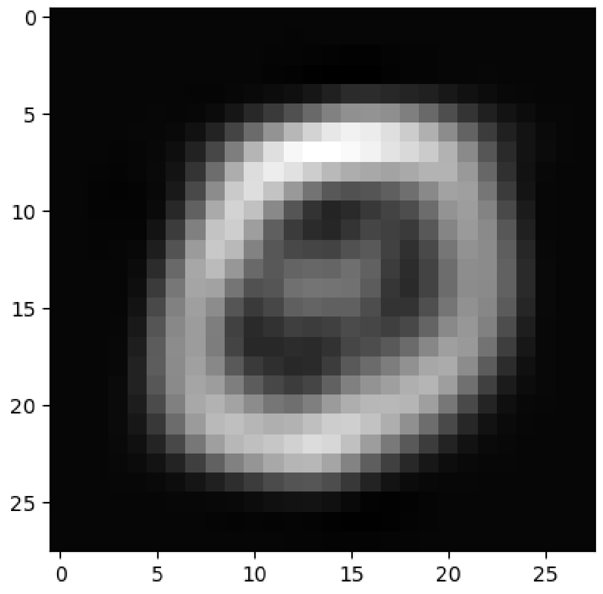
```
Projections: (16, 10000)
Accuracy: 0.8511
```

K-means clustering for generation

Using our clustering bins from above, we can take the center of each bin and use its value to "reconstruct" what a generic digit from that bin looks like

In [92]:

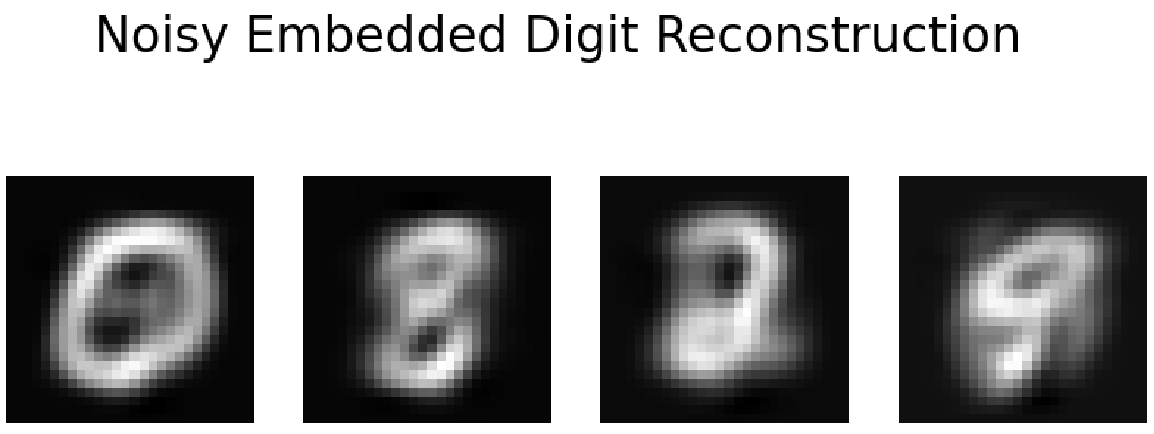
```
# transform k-means centers back to global coordinate frame and
# plot reconstructions of the centers

kmeans_center_images = reconstruct(pca_model, R.centers)

PyPlot.imshow(
    show_sample_img(kmeans_center_images, 27), cmap="gray"
)

PyPlot.savefig("kmeans-performance.png")
```



We may add some random noise to the center vectors and display what our model outputs

In [95]:

```
noisy_centers = mult_gauss(R.centers, 0.4)
kmeans_noisy_centers = reconstruct(pca_model, noisy_centers)

fig = figure(figsize=(10, 4))
subplot(141)

PyPlot.imshow(
    show_sample_img(kmeans_noisy_centers, 27), cmap="gray"
)
axis("off")

subplot(142)
PyPlot.imshow(
    show_sample_img(kmeans_noisy_centers, 22), cmap="gray"
)
axis("off")

subplot(143)
PyPlot.imshow(
    show_sample_img(kmeans_noisy_centers, 26), cmap="gray"
)
axis("off")

subplot(144)
PyPlot.imshow(
    show_sample_img(kmeans_noisy_centers, 34), cmap="gray"
)
axis("off")


suptitle("Noisy Embedded Digit Reconstruction",size=24)
PyPlot.savefig("noisy-digits.png")
```



To get a metric of how similar the bins are, we use the cosine similarity. This analyzes the dot product similarity between the centers of two clusters. We use the centers of the bins representing the digits 9 and 4. The two digits look somewhat similar, so we expect the centers to be somewhat similar

In [97]:
```julia
function euclidian_similarity(x, y)
    return sqrt(sum((x - y) .^ 2))
end

function cosine_similarity(x, y)
    return dot(x, y) / (norm(x) * norm(y))
end

# now look at dot product similarity of cluster centers
c1 = R.centers[:, 8]
c2 = R.centers[:, 22]

println("Cosine Similarity: " * string(cosine_similarity(c1, c2)))
println("Euc. Similarity: " * string(euclidian_similarity(c1, c2)))
```

```
Cosine Similarity: 0.7482903846173461
Euc. Similarity: 5.285774131620924
```

In [98]:
```julia
function digit_similarity(digit_one, digit_two)
    digit_one_bins = [i.first for i in assoc_vals if i.second == digit_one]
    digit_two_bins = [i.first for i in assoc_vals if i.second == digit_two]
    sum = 0
    for bin_one in digit_one_bins
        for bin_two in digit_two_bins
            sum += cosine_similarity(R.centers[:, bin_one], R.centers[:, bin_two])
        end
    end

    return sum/(size(digit_one_bins, 1) * size(digit_two_bins, 1))
end
```

Out[98]:
```
digit_similarity (generic function with 1 method)
```

In [100…
```julia
# construct an adjacency-matrix style table for the pairwise similarity of each digit class with all other
# classes, taking the average between different clusters of the same digit class

similarity_matrix = Array{Float64}(undef, 10, 10)
for i = 1:10
    for j = 1:10
        similarity_matrix[i, j] = digit_similarity(i-1, j-1)
    end
end

rounded_matrix = [round(i, digits=3) for i in similarity_matrix]
println(rounded_matrix)
```

```
[0.828 0.348 0.612 0.622 0.537 0.656 0.649 0.519 0.639 0.561; 0.348 0.755 0.59 0.555 0.431 0.507 0.485 0.463 0.62
4 0.485; 0.612 0.59 0.843 0.695 0.633 0.632 0.723 0.563 0.764 0.629; 0.622 0.555 0.695 0.881 0.549 0.745 0.618 0.
573 0.786 0.611; 0.537 0.431 0.633 0.549 0.841 0.618 0.664 0.648 0.689 0.781; 0.656 0.507 0.632 0.745 0.618 0.766
0.634 0.597 0.77 0.673; 0.649 0.485 0.723 0.618 0.664 0.634 0.853 0.536 0.69 0.666; 0.519 0.463 0.563 0.573 0.648
0.597 0.536 0.793 0.66 0.723; 0.639 0.624 0.764 0.786 0.689 0.77 0.69 0.66 0.885 0.74; 0.561 0.485 0.629 0.611 0.
781 0.673 0.666 0.723 0.74 0.819]
```

THANKS FOR READING!

For any questions: please email mreich@andrew.cmu.edu or dkrajews@andrew.cmu.edu