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))
```

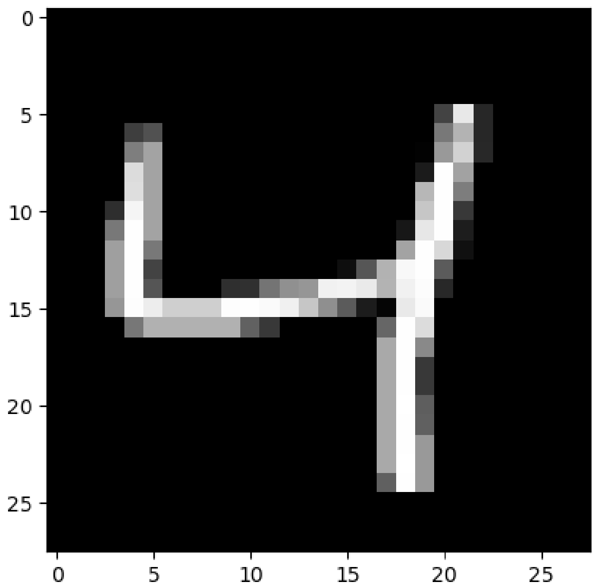
```
println("Singular values: ", test_svd.S)
```

```
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"
)
```

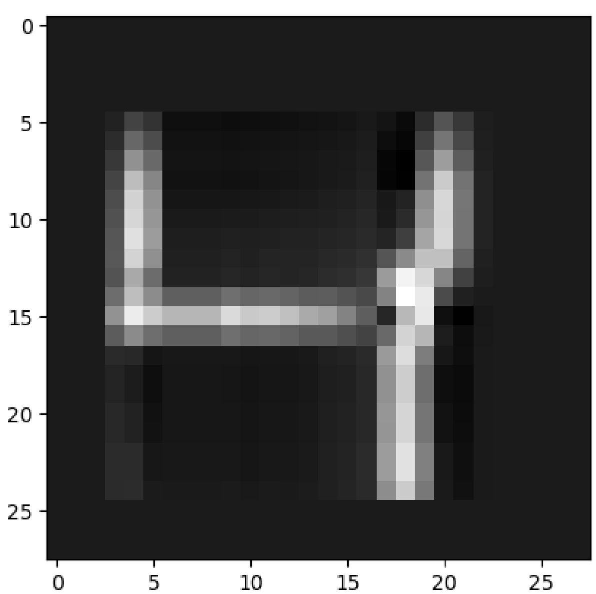


Out[15]: `PyObject <matplotlib.image.AxesImage object at 0x7f9ff8114790>`

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]:

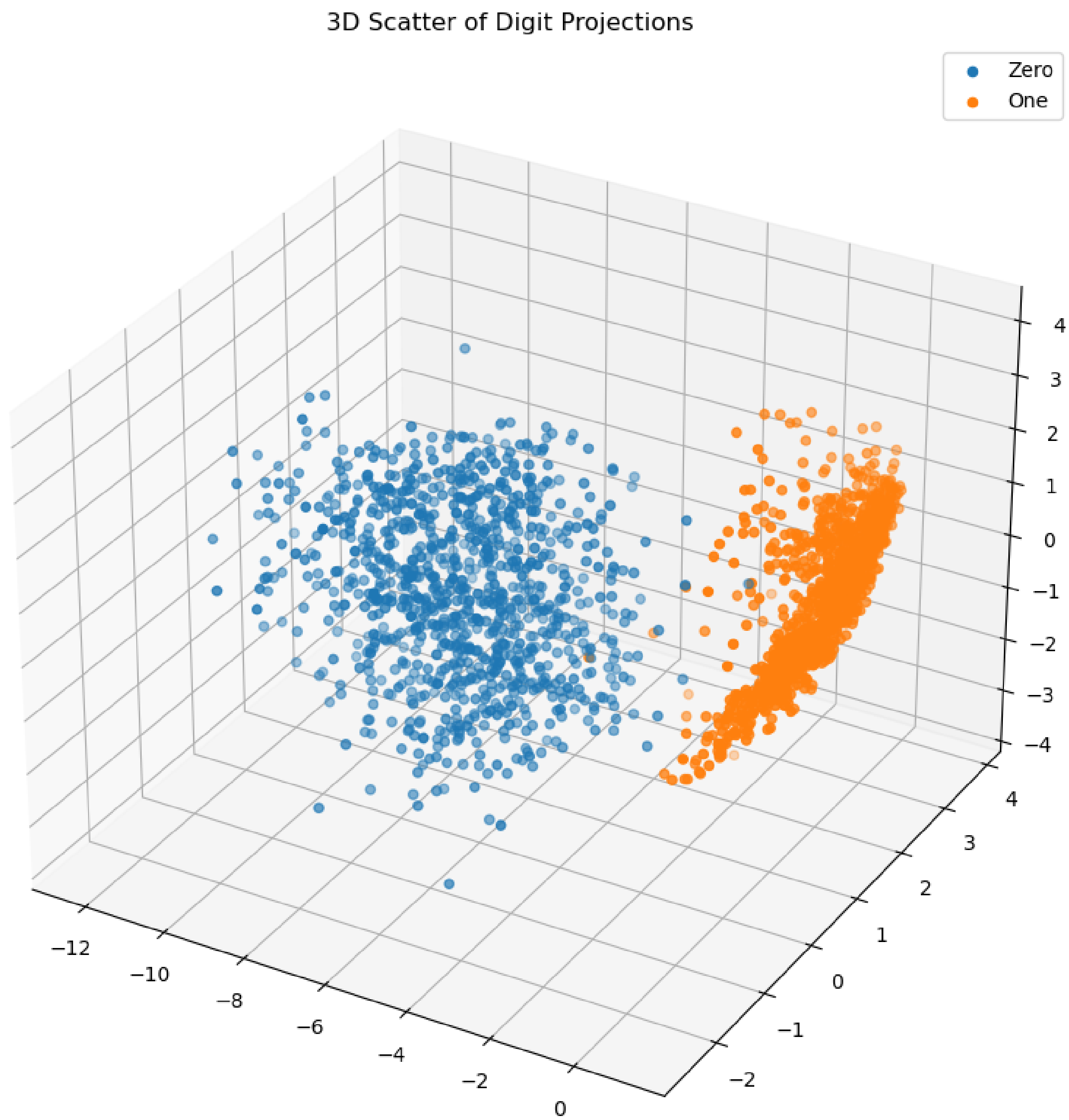
```
# determine embedding dimension, train PCA on mnist samples
embedding_dim = 16
pca_model, princ_vecs, pca_proj_mat = pca_m(reshaped_test_x, embedding_dim)

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
```

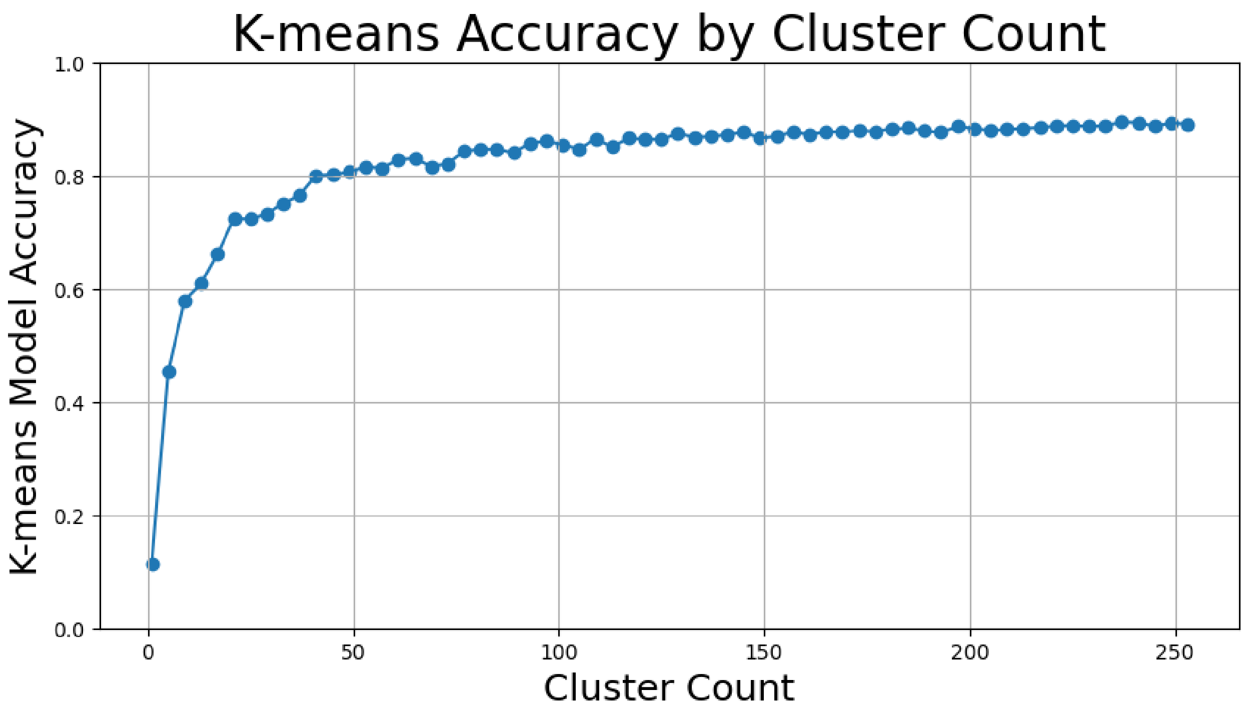
```
# gets most probable ground truth label corresponding to each bucket
# so that we know what cluster corresponds to what number
function get_cluster_assoc(cluster_labels, truth_labels)
    reference_labels = Dict()
    unique_labels = countmap(cluster_labels)

    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
```

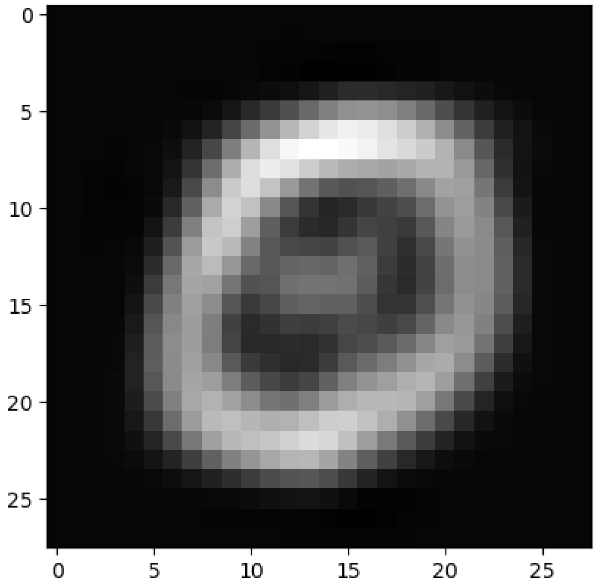
```
Projections: (16, 10000)
Accuracy: 0.8511
```

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]:

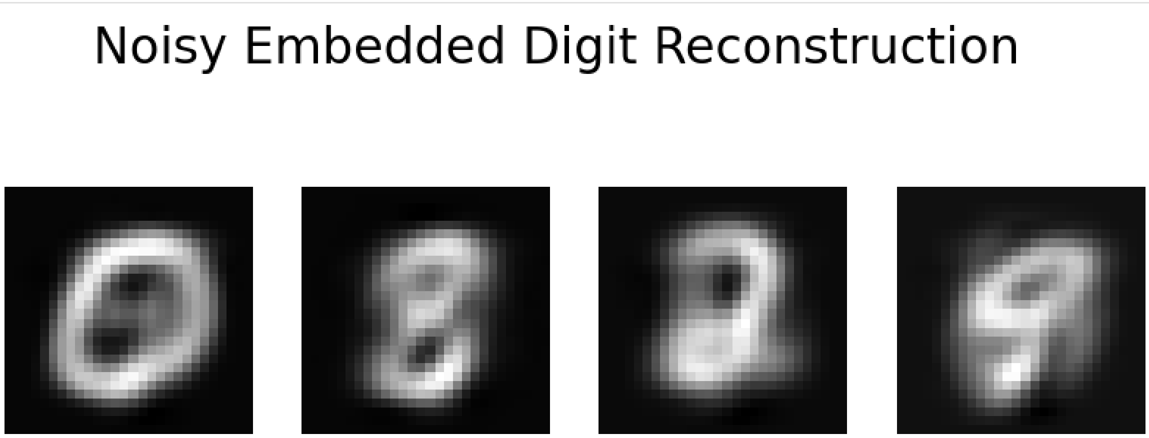
```
# transform k-means centers back to global coordinate frame and
# plot reconstructions of the centers

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]:

```
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]:

```
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…

```
# 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