MAT 167 Final Project

Handwritten Digit Recognition via SVD

Adam Hetherwick

Instructor: Luze Xu

12/10/2023

Introduction

Hand digit classification is an automation and efficiency tool that is utilized for postal services, healthcare, education, data handling and processing, and human-computer interaction. Many aspects of real world practices involve handwriting numbers and digits for different uses. Eventual transfer of these digits to computer based languages is essential. Numerous methods exist for recognizing digits, though some can retrieve higher classification rates than others. Once photos of the handwritten digits are taken, the pixels are decomposed into matrices with numbers to indicate the general darkness at each pixel for a given resolution. Each matrix, or collection of pixel data, is grouped together by digit, and the mean pixel data can be used to predict what other pixel data's digit could be.

For other pixel data matrices in which the digits are not known, we can find the Euclidean two-norm distance between the mean pixel matrices and the current pixel matrix we are examining. This will result in 10 two-norm distances, (one for each digit 0, 1, 2, …, 9) and the lowest 2-norm distance we conclude is the digit at that index. This however results in a classification rate that is unsatisfactory. This poses a problem for many people and fields, thus a higher classification rate model is needed. Higher classification rates can be obtained by utilizing the process of singular value decomposition.

Singular value decomposition (SVD), is a decomposition of a matrix A into key components $U\Sigma V^T$ (more information in later sections). These three matrices contain all the necessary properties of matrix A. A value k for the rank of the SVD can be used to approximate A, represented as A_k. The U matrix in our decomposition multiplied by the expansion coefficients results in our approximations of each digit's pixel data. The index of the lowest two-norm difference between the original pixel data and our approximation is the predicted digit. The overall classification success rate of our model is calculated using confusion matrices. The overall success rate is used to gauge the optimal value for k to approximate A. More information on such calculations are covered in the following sections.

This report examines this classification rate difference on data that originates from Canvas' *usps.mat* file, as well as Sci-Kit's handwritten dataset. The next section examines the data sets utilized and the relevant information for prediction modeling.

Data Set

The *usps.mat* dataset consists of 4 matrices representing relevant information for handwritten digit recognition. The *train_patterns* and *test_patterns* matrices are of dimension 256×4649 which contain a raster scan of 16×16 gray level pixel intensities. These 4649 raster scans are

elongated into 256 rows ($16 \cdot 16 = 256$), each column representing a unique handwritten digit. Both the *train_patterns* and *test_patterns* matrices are paired with a 10×4649 label matrix titled *train_label* or *test_label*. These matrices are exclusively -1's for non digits, and 1's at each row *i* to represent digit *i.* The *train_patterns* paired with the *train_labels* are used to train a function that accurately predicts the digit classification's in the *test_patterns* matrix.

To understand the *usps.mat* digit data further, we can display some of the raster scans. Doing so will give us a visual representation of what each raster scan could include.

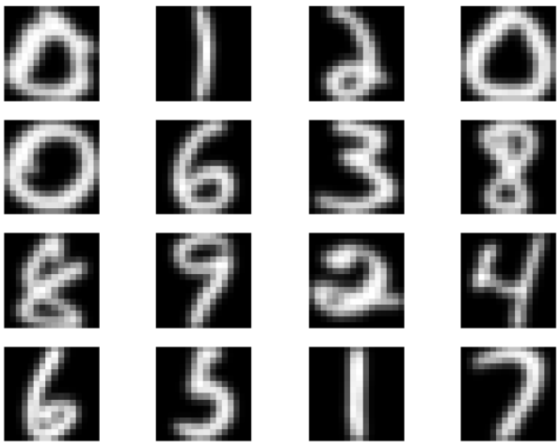


Figure 1: The first 16 raster scans in *usps.mat*

Here we can see that each digit has a different shape and thus different pixel data. We also notice that the images are composed in seemingly random order. In order for prediction and modeling, we must understand the general qualities of each digit. We can do this by grouping each digit's pixel data together and find the mean to generalize the pixel data for each digit.

Figure 2: Average pixel data for each digit in *usps.mat*

We can see that some figures have very consistent patterns, as seen by their mean pixel data representation being primarily bright solid white, as seen in numbers 1 and 7. Other digits have more varied mean pixel data, as seen in figures 4 and 8. Digits with more varied mean pixel data

may result in lower classification rates, because it could be harder to differentiate what is that digit and what is not.

This report also utilizes the Science-Kit Learn package (sklearn) which includes a handwritten digit dataset with 1797 samples. These samples are split into two groups to form the *x_train* and *x_test* objects, which consist of 898 8×8 raster gray level pixel intensities. These 8×8 pixel data matrices are flattened to express *x_train* and *x_test* as 64×898 two-dimensional matrices. Both *x_train* and *x_test* matrices are paired with a label matrix *y_train* and *y_test.* These label matrices are composed of all -1's at the indices that don't indicate the digit, and contain a 1 at index *i* which signifies the raster scan at that column is digit *i.* It follows that these label matrices are of size 10×898, for each digit (0, 1, …, 9) and each sample ($898 + 898 = 1796$, one is dropped for symmetry).

In order to gain more understanding of our Science-Kit digit pixel data, we can plot the first 16 digits. This information will help us visualize our dataset and understand the accompanying pixel intensities.

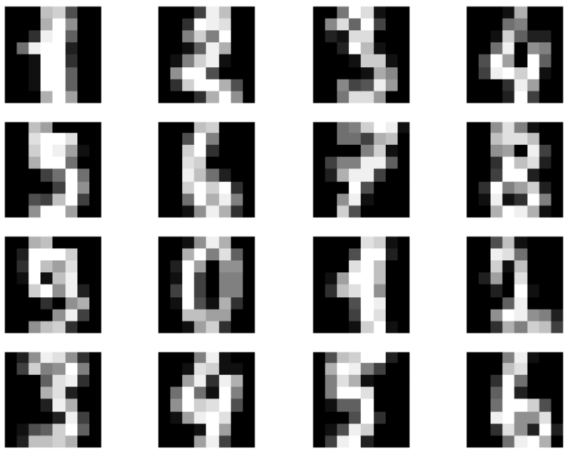


Figure 3: The first 16 scans in *Sci-Kit*'s handwritten digits database

Here we can see that the raster scans of the first 16 digits tell us (some) information about what the digits could look like. Given that this data is 8×8, differentiating the digits may be more difficult because we have less pixels to work with.

Figure 4: The average pixel data for each digit in *Sci-Kit*'s handwritten digits database

Here we can see that each digit is fairly blurry due to it being 8×8. Predictive modeling the digits in this dataset could be more difficult because the data is limited to less pixels, thus less variation. The following section investigates different methods of predictive modeling for these two datasets and calculates the relative success rate of each digit via confusion matrices.

Methods

Our prediction goal is to correctly predict a certain digit based on its raster pixel intensities data. To begin our predictive modeling, and to use as a reference later on, we can find the Euclidean 2-norm difference between each digit in *usps.mat*'s *test_patterns* and the average pixel intensities in *train_patterns*, grouped by digit. This will create a 10×4649 *test_classif* matrix, with each column *j* corresponding to each image's pixel intensities vector, and each row *i* corresponding to the Euclidean 2-norm distance between the pixel intensities in vector *j* and mean digit *i.*

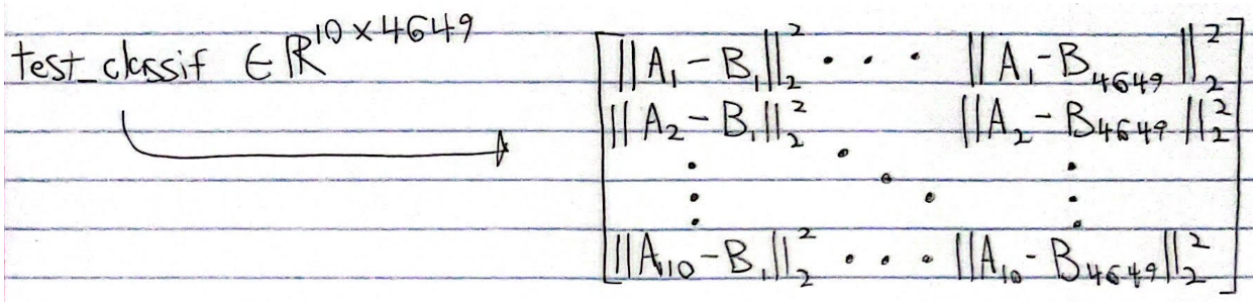


Figure 5: Matrix representation of object *test_classif*

$$||x||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

The lowest 2-norm distance is the smallest value in each vector in *test_classif*. Our predicted digit is then the row index in which the smallest 2-norm distance resides for each column. We can express each predicted digit as a 1×4649 matrix *test_classif_res*.

To calculate the classification rate of our initial prediction model, we can create a 10×10 confusion matrix which signifies the correctly predicted values in row *i* column *i,* and all incorrectly predicted values off the main diagonal.

```
[656,   1,   3,   4,  10,  19,  73,   2,  17,   1]
[  0, 644,   0,   1,   0,   0,   1,   0,   1,   0]
[ 14,   4, 362,  13,  25,   5,   4,   9,  18,   0]
[  1,   3,   4, 368,   1,  17,   0,   3,  14,   7]
[  3,  16,   6,   0, 363,   1,   8,   1,   5,  40]
[ 13,   3,   3,  20,  14, 271,   9,   0,  16,   6]
[ 23,  11,  13,   0,   9,   3, 354,   0,   1,   0]
[  0,   5,   1,   0,   7,   1,   0, 351,   3,  34]
[  9,  19,   5,  12,   6,   6,   0,   1, 253,  20]
[  1,  15,   0,   1,  39,   2,   0,  24,   3, 314]
```

Figure 6: The confusion matrix reflecting prediction model results

Here we see that the majority of our predictions were correct, though still many misclassifications took place. The highest being 73 6 digits were misrecognized and predicted to be 0's. The overall classification rate of this model is 84.66% success rate, and can be calculated as follows, where C denotes the confusion matrix above:

$$\frac{trace(C)}{sum(C)}$$

A classification rate of 84.66% is not acceptable and being the advanced mathematicians that we are, we may calculate our predictions after a decomposition of our training pixel data. To deconstruct our training pixel data into its' defining qualities, we may use singular value decomposition, which is defined as follows:

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

$$U_i = \frac{A \cdot V_i}{||A \cdot V_i||_2}$$

$$\Sigma = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_i \end{bmatrix}$$

$$(A^T \cdot A - \lambda_i \cdot I_n)V_i = 0$$

In our analysis, we will utilize a rank *k* SVD to gather the important properties of our training pixel data. To do so, we only need the left singular vectors in the matrix U, which has dimensions $256 \times k \times 10$, where k denotes the number of columns in each matrix U, one for each digit (0, 1, …, 9).

In order to utilize our U matrices for analysis, we must multiply them by the expansion coefficients to produce our rank *k* approximations for each digit in *test_patterns*. The expansion coefficients can be calculated by multiplying each transposed *U* matrix corresponding to each digit by the *test_patterns*. This will result in a $k \times 4649 \times 10$ matrix which maps any transposed left singular matrix *U* into the dimensions of the *test_patterns* matrix.

Next we calculate the rank *k* approximation matrices for each digit by multiplying each *U* matrix by the expansion coefficients. This will result in a $256 \times 4649 \times 10$ matrix that holds our predictions for each digit in the *test_patterns* matrix.

Like before, we may compute the 2-norm Euclidean distance between each digit's prediction in each sheet of the prediction matrix calculated in the previous step, and the actual *test_patterns* digit data. This will result in a matrix of shape 10×4649 that holds the 2-norm distance between digit *i*'s prediction in row *i.* This entails that our prediction results can be computed by finding the index in which the smallest 2-norm resides, and assigning that as the predicted digit for each column *j.* This would result in a matrix of shape 1×4649, which can be used to compute the confusion matrix and classification success rate as shown before.

This method of prediction is proven to be more accurate by a much higher margin, though the specific amount of components *k* chosen can vary the success rate. A function (listed in the appendix) is implemented to compute the optimal *k* value for the highest prediction rate. The following Frobenius norm difference between the original *test_patterns* matrix and the approximation A_k will hold that the higher prediction value *k,* the lower the Frobenius norm differences will be:

$$||A - A_k||_F$$

$$||A||_F = \sqrt{trace(A^T \cdot A)}$$

The following section will report the results of this function and the highest classification rate *k* along with its Frobenius norm errors compared to lower classification rate *k*'s. This function is applied to the *usps.mat* digit data and the Sci-Kit Learn's handwritten digit database.

Results

The function utilizes all the steps illustrated above and returns the classification rate for a given k. The matrix U is involved in retaining the most important aspects of our original training patterns. To visualize our matrix U for each digit in the *usps.mat training_patterns*, we may plot the matrices.

Figure 7: Each digit's matrix U left singular vectors

We see that the important qualities of each digit are retained in each left singular matrix U. You can faintly see the outline of each digit, and the lines that are the most bold represent the defining qualities of those digits when handwritten. After implementation of the function, the classification rates vs their k values were calculated and displayed for k values from 1 to 20.

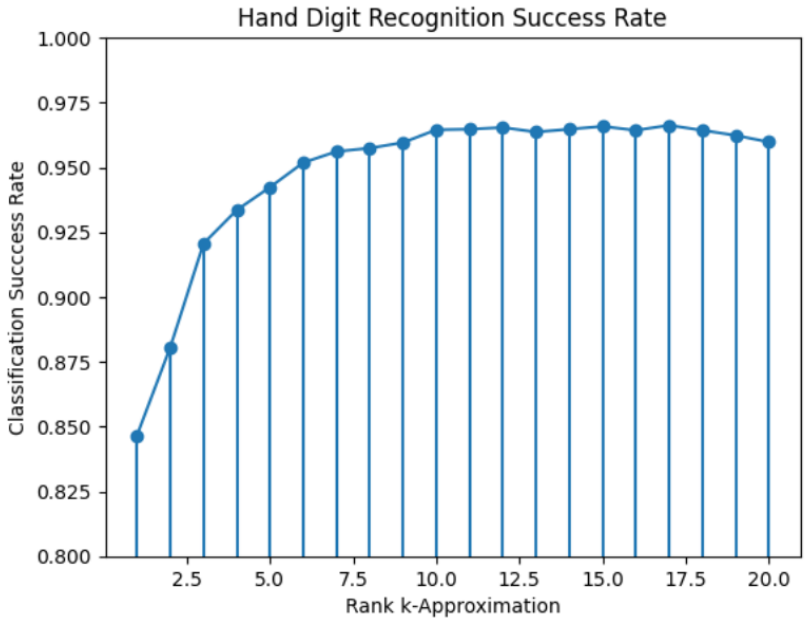


Figure 8: Rank k approximations of *usps.mat* digit data

plotted vs overall success rate

Here we can see that the highest classification rate is returned at $k = 17$, with a classification success rate of 96.62%. This is much higher than the classification success rate of the original model that did not utilize SVD. The confusion matrix for $k = 17$, also reflects such success, as the values off the main diagonal are very small compared to that of the original model prediction.

```
[772,   2,   1,   3,   1,   1,   2,   1,   3,   0]
[  0, 646,   0,   0,   0,   0,   0,   0,   0,   1]
[  3,   6, 431,   6,   0,   3,   1,   2,   2,   0]
[  1,   1,   4, 401,   0,   7,   0,   0,   4,   0]
[  2,   8,   1,   0, 424,   1,   1,   5,   0,   1]
[  2,   0,   0,   5,   2, 335,   7,   1,   1,   2]
[  6,   4,   0,   0,   2,   3, 399,   0,   0,   0]
[  0,   2,   0,   0,   2,   0,   0, 387,   0,  11]
[  2,   9,   1,   5,   1,   1,   0,   0, 309,   3]
[  0,   5,   0,   1,   0,   0,   0,   4,   1, 388]
```

Figure 9: Confusion matrix for highest classification rank $k = 17$ for the *usps.mat* digit data

This confusion matrix confirms our prediction rate being very high. Another way of quantifying our prediction success is to calculate the Frobenius norm distance between the original matrix data in *test_patterns* and our rank *k* approximation for $k = 17$. Below are the results from the Frobenius norm error with $k = 10$ (left), and $k = 17$ (right).

	Digit	Frobenius Norm Error
0	0	416.344496
1	1	427.103242
2	2	413.074698
3	3	424.663474
4	4	426.979200
5	5	430.343440
6	6	430.840114
7	7	429.525212
8	8	413.580089
9	9	440.114515

	Digit	Frobenius Norm Error
0	0	333.178143
1	1	376.798858
2	2	357.975033
3	3	342.574259
4	4	370.491506
5	5	326.749094
6	6	351.284773
7	7	372.313935
8	8	340.563460
9	9	372.307304

Figure 10: Frobenius norm errors between the *test_patterns* and the rank *k* approximation for $k = 10$ (left), 17 (right)

Here we can see that the Frobenius norm decreases substantially for each digit approximation as the value of k becomes closer to the highest prediction k. We may also test our function on other digit data to ensure our SVD approximations result in high accuracy. We can test this in Sci-Kit Learn's handwritten digit database that is available as an installable Python package. The data was manipulated slightly to be compatible with our function (details can be found in the Code Appendix section). The function was applied to this dataset and the classification success rates for different rank k approximations were calculated.

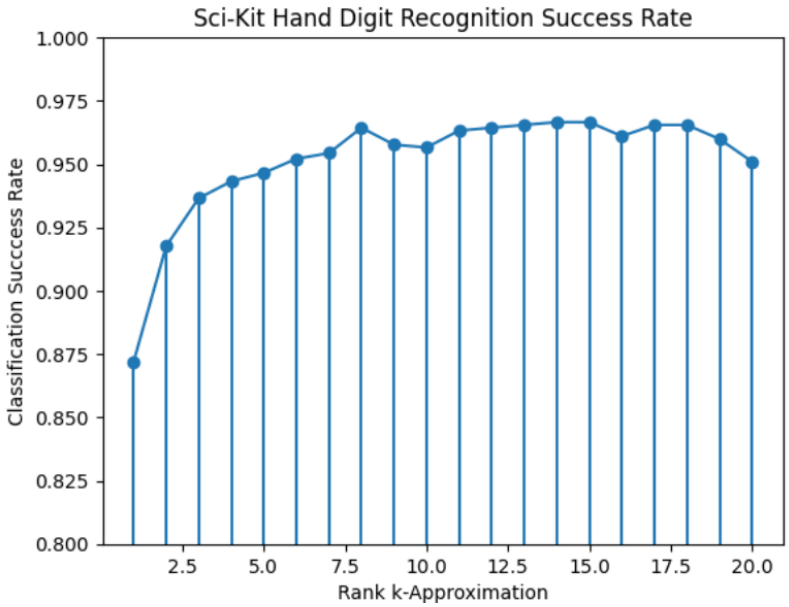


Figure 11: Rank k approximations of Sci-Kit Learn's handwritten digit database plotted vs overall success rate

The highest rank k approximation of this digit database is at $k = 15$ with classification success rate 96.66%. This is nearly identical to our classification success rate with the *usps.mat* data, indicating that the model is highly accurate, and this may be the best it can get. The confusion matrix for the Sci-Kit-Learn data reciprocates our findings.

```
[88,  0,  0,  0,  0,  0,  0,  0,  0,  0]
[ 0, 89,  1,  0,  0,  0,  0,  0,  1,  0]
[ 0,  0, 86,  0,  0,  0,  0,  0,  0,  0]
[ 0,  0,  1, 87,  0,  0,  0,  1,  2,  0]
[ 1,  0,  0,  0, 85,  0,  0,  2,  0,  4]
[ 0,  0,  0,  0,  0, 86,  0,  0,  0,  5]
[ 0,  0,  0,  0,  0,  0, 91,  0,  0,  0]
[ 0,  0,  0,  0,  0,  0,  0, 89,  0,  0]
[ 0,  8,  0,  0,  0,  0,  0,  0, 79,  0]
[ 0,  0,  0,  3,  0,  1,  0,  0,  0, 88]
```

Figure 12: Confusion matrix for Sci-Kit Learn's handwritten digit database

This confirms that our function predicts the digits very accurately and can be used for other handwritten datasets. It also follows that the data intensities being 8×8 scale did not influence our prediction rate.

Conclusions

This report calculated the classification prediction rates for handwritten digit data from the *usps.mat* file from Canvas, and Sci-Kit Learn's handwritten digit database available via Python. The original 2-norm difference between each digit and the mean pixel intensities data for that digit resulted in classification errors and an accuracy of 84.66%. The singular value decomposition was computed to retain the most important properties of our training pixel data. The U matrices, the expansion coefficients, and the rank k approximations proved to be sufficient materials to conduct a high classification rate model. A function was implemented to calculate the optimal value of k via the highest classification rate.

We concluded that the optimal value of k for the training patterns was $k = 17$ for the *usps.mat* file and $k = 15$ for Sci-Kit Learn's dataset. The confusion matrices confirmed our classification rates by possessing primarily 0's off the main diagonal, and the majority of data on the trace of the matrix. The final classification prediction results of 96.62% for the *usps.mat* file and 96.66% for the Sci-Kit Learn's database are sufficient to utilize in the future for hand digit recognition.

This information is important for future consideration because utilization of the singular value decomposition of our matrix resulted in a substantially higher classification rate. This information is relevant to employment agencies, postal services, educational institutions, hospitals, and many other relevant fields.

Disclaimer

Throughout this report I may sometimes use the collective terms 'Our', 'We', 'Ours' etc. This is because I am referring to myself and the reader. I completed this report entirely on my own with no assistance from my classmates. I occasionally asked Luze for assistance and read a little about how to load the Sci-Kit database but that is it.

Code Appendix

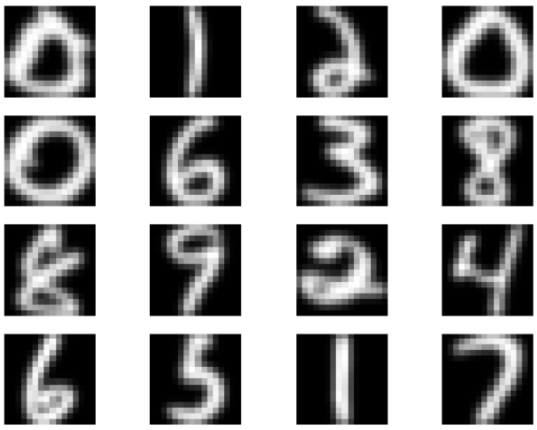
[1]:
```
from scipy import io
import matplotlib.pyplot as plt
import numpy as np
from math import sqrt
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import pandas as pd
from scipy.sparse.linalg import svds
import scipy.sparse as sp
from numpy.linalg import svd
```

[2]:
```
# Load the data and create the matrices

data_dict = io.loadmat('usps.mat')

train_patterns = data_dict['train_patterns']
train_labels = data_dict['train_labels']

test_patterns = data_dict['test_patterns']
test_labels = data_dict['test_labels']
```

2.3a)

2.3a)

[3]:
```
# Display the original figures

fig, axes = plt.subplots(4, 4)
for i in range(4):
    for j in range(4):
        col = i * 4 + j + 1
        axes[i, j].imshow(np.reshape(train_patterns[:, col], (16, 16)), cmap = 'grey')
        axes[i, j].axis('off')
plt.show()
```



2.3b) Compute the mean pixel data for each digit (0, 1, ..., 9)

[4]:
```
# Make train_aves of size 256x10

train_aves = []
fig, axes = plt.subplots(2, 5)

for i in range(2):
    for j in range(5):
        digit = 5*i + j
        digit_mean = np.mean(train_patterns[:,train_labels[digit,:]==1], axis=1)

        # Reshape digit_mean to have 1 column
        digit_mean = np.reshape(digit_mean, (256, 1))
        train_aves.append(digit_mean)

        # Reshape digit_mean for image
        digit_img = np.reshape(digit_mean, (16, 16))
        axes[i, j].imshow(digit_img, cmap = 'grey')
        axes[i, j].axis('off')
plt.show()
train_aves = np.concatenate(train_aves, axis=1)
train_aves.shape
```

[4]:
```
(256, 10)
```

Basically, what I did in the above step was look in the train_labels matrix and find the indexes that pair with number 0, 1, ..., 9 and use those indices to find the mean digit data in the train_patterns matrix. In simple terms, find the mean pixels for each number and show the images.

[6]:
```
# Calculating the condition number k(A) of the mean digit pixel data matrix train_aves
# Large k(A) signifies linear system is not stable, solution Ax = b has large error and A is close to linear dependence
# Low k(A) signifies linear system is stable, Ax= b has small error, and A has distinct column vectors, not close to linear dependence.

tmp, train_aves_sigma, tmp2 = np.linalg.svd(train_aves)

# Calculate the k(A) with sigma1 / sigma10:
print(train_aves_sigma[0] / train_aves_sigma[-1])
train_aves_sigma
```

```
17.646001908077142
```

[6]:
```
array([30.45664032,  8.769182  ,  7.05170477,  5.24734103,  4.56601767,
        4.31044144,  3.54615739,  3.00793607,  2.12147029,  1.72597966])
```

The condition number is high here, signifying that the matrix is ill conditioned, and may have nearly linearly dependent columns

2.3c) Conduct the classification experiment

2.3c.1) Create a matrix test_classif (10x4649) with the Euclidean distance between each image in the test_patterns and each mean digit in train_aves

[7]:
```
test_classif = []

for i in range(4649):  # 4649 cols in test_patterns
    test_classif_col = []
    for j in range(10):  # 10 cols in train_aves.T
        diff = test_patterns[:, i] - train_aves[:, j]
        two_norm = np.linalg.norm(diff, ord=2)
        test_classif_col.append(two_norm*two_norm)
    test_classif_col = np.array(test_classif_col)
    test_classif_col = np.reshape(test_classif_col, (10, 1))
    test_classif.append(test_classif_col)

test_classif = np.concatenate(test_classif, axis=1)
test_classif.shape
```

[7]:
```
(10, 4649)
```

Basically this step in english, I found the 2 norm distance between each column (image pixel data) in test_pattern with each column in train_aves (the mean pixel data for 0, 1, ..., 9) and appended that 2 norm to each row of each column of test_classif.

test_classif is now a 10x4649 matrix with each tc[1, 1] being the 2-norm difference between column 1 in test_patterns and column 1 in train_aves. tc[1, 1] represents test_pattern's first pixel image data with mean pixel data of digit 0. tc[1, 2] represents test_pattern's first pixel image data with mean pixel data of digit 1. etc

2.3c.2) Compute the classification results by finding the smallest 2-norm in test_classif

[8]:
```
test_classif_res = []

for i in range(4649):  # 4649 cols in test_classif
    index = np.argmin(test_classif[:, i])
    test_classif_res.append(index)

test_classif_res = np.reshape(test_classif_res, (1, 4649))
test_classif_res.shape
```

[8]:
```
(1, 4649)
```

In this step we found the smallest 2-norm in each column of test_classif, and used that index as determining what the pixels in test_patterns most closely relate to. test_classif_res is a 1x4649 matrix, of the digits that the 2-norm thinks is the most likely digit, based off the difference.

For example: test_classif_res[0], is the digit that we think test_patterns's first column is, based off 2-norm.

2.3c.3) Compute the 10x10 confusion matrix with the ratio of test_patterns with the correct predicted label.

[51]:
```
# for each valid digit (0, 1, ..., 9) count how many of those digits we correctly guessed in test_classif_res and turn it into a
# vector with the frequency of each digit at it's column location in v.
test_confusion = []

for k in range(10):  # 10 because 10 total digits (0, 1, ..., 9)

    tmp = test_classif_res[0, np.where(test_labels[k, :] == 1)[0]]
    tmp_flat = tmp.flatten()
    unique_elements, counts = np.unique(tmp_flat, return_counts=True)

    # Add the frequencies of each digit into the row_k
    row_k = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    for val, count in zip(unique_elements, counts):
        row_k[val] = count
    test_confusion.append(row_k)

test_confusion = np.array(test_confusion)
print(test_confusion.shape)
test_confusion
```

```
(10, 10)
```

[51]:
```
array([[656,   1,   3,   4,  10,  19,  73,   2,  17,   1],
       [  0, 644,   0,   1,   0,   0,   1,   0,   1,   0],
       [ 14,   4, 362,  13,  25,   5,   4,   9,  18,   0],
       [  1,   3,   4, 368,   1,  17,   0,   3,  14,   7],
       [  3,  16,   6,   0, 363,   1,   8,   1,   5,  40],
       [ 13,   3,   3,  20,  14, 271,   9,   0,  16,   6],
       [ 23,  11,  13,   0,   9,   3, 354,   0,   1,   0],
       [  0,   5,   1,   0,   7,   1,   0, 351,   3,  34],
       [  9,  19,   5,  12,   6,   6,   0,   1, 253,  20],
       [  1,  15,   0,   1,  39,   2,   0,  24,   3, 314]], dtype=int64)
```

[10]:
```
# No SVD approximation success rate:

correct = np.trace(test_confusion)
total = np.sum(test_confusion)
correct / total
```

[10]:
```
0.8466336846633684
```

In this step we found which column indexes have which digits in them, then used those indexes to look in our test_classif_res object to see (for example) if our 1's actually had 1's in them. We then counted the number of each element into a vector row_k then turned that into a matrix test_confusion

2.3d.1) Compute the rank k SVD of the train_patterns for each digit (0, 1, ..., 9), then put the left singular vectors into a tensor of size 256 x k x 10

[11]:
```
# start with k = 10
k = 10
train_u = np.zeros((256, k, 10))

for j in range(10):
    train_u[:,:,j], tmp, tmp2 = svds(train_patterns[:,train_labels[j,:] == 1], k)
```

[11]:
```
train_u.shape
```

[11]:
```
(256, 10, 10)
```

[12]:
```
# Visualize the left singular vectors:

fig, axes = plt.subplots(2, 5)

for i in range(2):
    for j in range(5):
        digit = 5*i + j
        left_sing_vector = np.reshape(train_u[:, j, digit], (16, 16))
        axes[i, j].imshow(left_sing_vector, cmap = 'grey')
        axes[i, j].axis('off')
plt.show()
```

In this step, we created a tensor matrix train_u that has dimensions 256xkx10, where k is the A_k SVD approximation of all the train patterns, with the k left singular vectors.

2.3d.2) Compute the expansion coefficients of each test digit. Compute k x 10 numbers for each image and put the results in the 3D tensor array test_svd of size k x 4649 x 10.

[13]:
```
test_svd = np.zeros((k, 4649, 10))

for j in range(10):
    test_svd[:, :, j] = train_u[:, :, j].T @ test_patterns

test_svd.shape
```

[13]:
```
(10, 4649, 10)
```

2.3d.3) Compute the error between each original test digit image and its rank k approximation.

[14]:
```
# Compute the approximations:
rank_k_approx = np.zeros((256, 4649, 10))

for j in range(10):
    rank_k_approx[:, :, j] = train_u[:, :, j] @ test_svd[:, :, j]

rank_k_approx.shape
```

[14]:
```
(256, 4649, 10)
```

[15]:
```
error_data = []
indices = []

for i in range(10):
    error_data.append(np.linalg.norm(rank_k_approx[:, :, i] - test_patterns, ord='fro'))
    indices.append(i)

pd.DataFrame({'Digit': indices,
              'Frobenius Norm Error': error_data})
```

[15]:

	Digit	Frobenius Norm Error
0	0	416.344496
1	1	427.103242
2	2	413.074698
3	3	424.663474
4	4	426.979200
5	5	430.343440
6	6	430.840114
7	7	429.525212
8	8	413.580089
9	9	440.114515

[16]:
```
test_svdres = []

for i in range(4649):  # 4649 train_patterns is 256x4649
    original = test_patterns[:, i]
    row = []
    for j in range(10):
        rank_k_approx_col = rank_k_approx[:, i, j]
        diff = np.linalg.norm(original - rank_k_approx_col, ord=2)
        row.append(diff)
    test_svdres.append(row)

test_svdres = np.column_stack(test_svdres)
test_svdres.shape
```

[16]: (10, 4649)

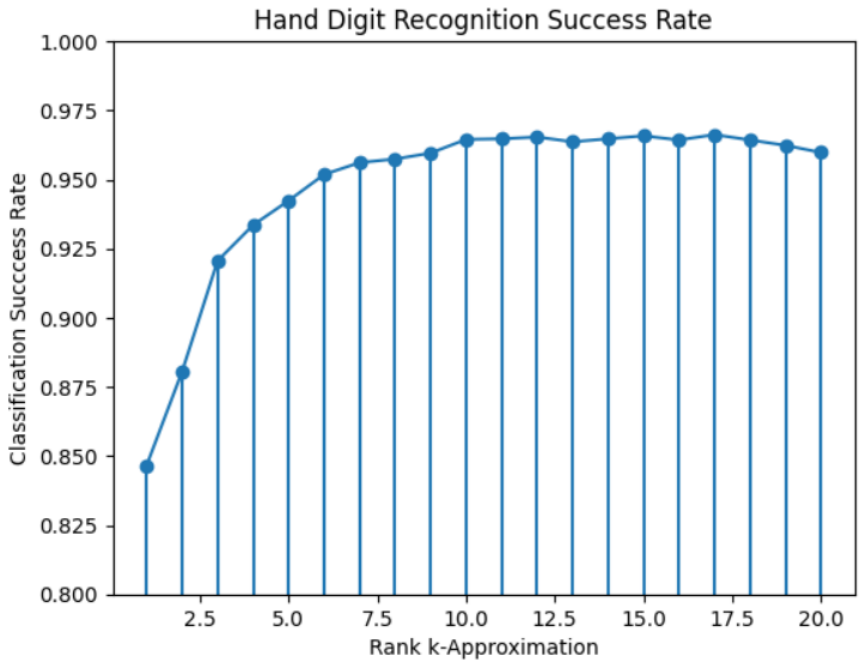
2.3d.4) Create a function usps_svd_classification that compares the overall classification rate for k = 1:20.

[17]:
```
def usps_svd_classification(train_patterns, test_patterns, train_labels, test_labels, k) -> int:
    """
    This function approximates the pixel data in train_patterns using different k components in the SVD,
    then returns the overall predicted classification rate and predicted labels for each digit in test_patterns

    INPUT:

    - train_patterns: A mxn matrix with n pixel data vectors, describing handwritten digits (0, 1, ..., 9). This will train the prediction model.
    - test_patterns: A mxn matrix with n pixel data vectors that we will utilize to quantify the success of our prediction model.
    - train_label: A 10xn matrix with each n vector consisting of nine -1's and one 1 at the index where the train_pattern pixel data digit corresponds t
    - test_label: A 10xn matrix with each n vector consisting of nine -1's and one 1 at the index where the test_pattern pixel data digit corresponds to.
    - k: the starting number of components in the SVD

    OUTPUT:

    - rate: the overall classification rate of our predictive model on the test_patterns
    - test_predict: the predicted digit labels for the test_patterns
    """
    nrows = train_patterns.shape[0]
    ncols = train_patterns.shape[1]

    # Create a 3D matrix for each U in the SVD decomposition of rank k for digits (0, 1, ..., 9)
    train_u = np.zeros((nrows, k, 10))

    # Create a 3D matrix for the expansion coefficients for each digit
    test_svd = np.zeros((k, ncols, 10))

    # Create a 3D matrix for the predicted digits, based on 2-norm
    rank_k_approx = np.zeros((nrows, ncols, 10))

    for j in range(10):

        # Each sheet of train_u are the left singular values of the rank k SVD
```

```
        train_u[:,:,j], tmp, tmp2 = svds(train_patterns[:,train_labels[j,:] == 1], k)

        # Each sheet of test_svd are the expansion coefficients in a = U.T @ x
        test_svd[:, :, j] = train_u[:, :, j].T @ test_patterns

        # Our approximation for each digit using the k-rank SVD, left singular vectors, and expansion coefficients
        rank_k_approx[:, :, j] = train_u[:, :, j] @ test_svd[:, :, j]

    test_svdres = []
    for i in range(ncols):
        original = test_patterns[:, i]
        row = []
        for j in range(10):
            rank_k_approx_col = rank_k_approx[:, i, j]
            # Quantify the difference between our prediction and the original photo pixel data, lower 2-norm the better
            diff = np.linalg.norm(original - rank_k_approx_col, ord=2)
            row.append(diff)
        test_svdres.append(row)
    test_svdres = np.column_stack(test_svdres)

    test_svd_pred = []
    for i in range(ncols):
        # Find the index in which test_svdres has the lowest 2-norm
        index = np.argmin(test_svdres[:, i])
        test_svd_pred.append(index)
    test_svd_pred = np.reshape(test_svd_pred, (1, ncols))

    test_svd_confusion = []
    for k in range(10):
        # Find the lowest 2-norm difference between the original pixel data and our approximation.
        tmp = test_svd_pred[0, np.where(test_labels[k, :] == 1)[0]]
        tmp_flat = tmp.flatten()
        unique_elements, counts = np.unique(tmp_flat, return_counts=True)
            # Add the frequencies of each digit into the row_k
            row_k = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
            for val, count in zip(unique_elements, counts):
                row_k[val] = count
            test_svd_confusion.append(row_k)
        test_svd_confusion = np.array(test_svd_confusion)

        correct = np.trace(test_svd_confusion)
        total = np.sum(test_svd_confusion)
        classif_rate = correct / total
        return classif_rate
```

18]:

```
classif_rates = []
k_s = []

for k in range(1, 21):
    classif_rate = usps_svd_classification(train_patterns, test_patterns, train_labels, test_labels, k)
    k_s.append(k)
    classif_rates.append(classif_rate)

plt.stem(k_s, classif_rates)
plt.plot(k_s, classif_rates)
plt.ylim(0.8, 1.0)
plt.title('Hand Digit Recognition Success Rate')
plt.xlabel('Rank k-Approximation')
plt.ylabel('Classification Succcess Rate')
plt.show()
```



[19]:
```
max_val, max_index = max((val, index) for index, val in enumerate(classif_rates))
print('The maximum classification rate is', max_val, 'at k =', max_index+1)
```

```
The maximum classification rate is 0.9662292966229297 at k = 17
```

2.3d.5) Compute the confusion matrix and overall classification rate using the best k approximation via SVD.

[20]:
```
# Using k=17, the maximum classification rate:

best_k = 17
best_train_u = np.zeros((256, best_k, 10))

for j in range(10):
    best_train_u[:,:,j], tmp, tmp2 = svds(train_patterns[:,train_labels[j,:] == 1], best_k)
```

[21]:
```
best_test_svd = np.zeros((best_k, 4649, 10))

for j in range(10):
    best_test_svd[:, :, j] = best_train_u[:, :, j].T @ test_patterns
```

[22]:
```
# Compute the approximations:
best_rank_k_approx = np.zeros((256, 4649, 10))

for j in range(10):
    best_rank_k_approx[:, :, j] = best_train_u[:, :, j] @ best_test_svd[:, :, j]
```

[23]:
```
error_data = []
indices = []

for i in range(10):
    error_data.append(np.linalg.norm(best_rank_k_approx[:, :, i] - test_patterns, ord='fro'))
    indices.append(i)
```

```
pd.DataFrame({'Digit': indices,
              'Frobenius Norm Error': error_data})
```

[23]:

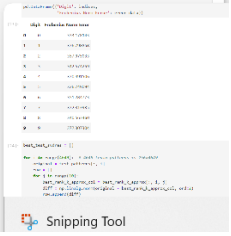
	Digit	Frobenius Norm Error
0	0	333.178143
1	1	376.798858
2	2	357.975033
3	3	342.574259
4	4	370.491506
5	5	326.749094
6	6	351.284773
7	7	372.313935
8	8	340.563460
9	9	372.307304

[24]:
```
best_test_svdres = []

for i in range(4649):  # 4649 train_patterns is 256x4649
    original = test_patterns[:, i]
    row = []
    for j in range(10):
        best_rank_k_approx_col = best_rank_k_approx[:, i, j]
        diff = np.linalg.norm(original - best_rank_k_approx_col, ord=2)
        row.append(diff)
    best_test_svdres.append(row)
```

```
best_test_svdres = np.column_stack(best_test_svdres)
```

[25]:
```
best_test_svd_pred = []

for i in range(4649):  # 4649 cols in test_svdres
    index = np.argmin(best_test_svdres[:, i])
    best_test_svd_pred.append(index)

best_test_svd_pred = np.reshape(best_test_svd_pred, (1, 4649))
```

[26]:
```
# for each valid digit (0, 1, ..., 9) count how many of those digits we correctly guessed in test_classif_res and turn it into a
# vector with the frequency of each digit at it's column location in v.
best_test_svd_confusion = []

for k in range(10):  # 10 because 10 total digits (0, 1, ..., 9)

    tmp = best_test_svd_pred[0, np.where(test_labels[k, :] == 1)[0]]
    tmp_flat = tmp.flatten()
    unique_elements, counts = np.unique(tmp_flat, return_counts=True)

    # Add the frequencies of each digit into the row_k
    row_k = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    for val, count in zip(unique_elements, counts):
        row_k[val] = count
    best_test_svd_confusion.append(row_k)

best_test_svd_confusion = np.array(best_test_svd_confusion)
best_test_svd_confusion
```



[26]:
```
array([[772,   2,   1,   3,   1,   1,   2,   1,   3,   0],
       [  0, 646,   0,   0,   0,   0,   0,   0,   0,   1],
       [  3,   6, 431,   6,   0,   3,   1,   2,   2,   0],
       [  1,   1,   4, 401,   0,   7,   0,   0,   4,   0],
       [  2,   8,   1,   0, 424,   1,   1,   5,   0,   1],
       [  2,   0,   0,   5,   2, 335,   7,   1,   1,   2],
       [  6,   4,   0,   0,   2,   3, 399,   0,   0,   0],
       [  0,   2,   0,   0,   2,   0,   0, 387,   0,  11],
       [  2,   9,   1,   5,   1,   1,   0,   0, 309,   3],
       [  0,   5,   0,   1,   0,   0,   0,   4,   1, 388]], dtype=int64)
```

[27]:
```
correct = np.trace(best_test_svd_confusion)
total = np.sum(best_test_svd_confusion)

correct / total
```

[27]:
```
0.9662292966229297
```

[28]:
```
# Testing with data from sci-kit
# usps_svd_classification(train_patterns, test_patterns, train_labels, test_labels, k)

from sklearn import datasets
from sklearn.model_selection import train_test_split
```

[29]:
```
digits = datasets.load_digits()
```

[52]:
```
digits
```

[31]:
```
n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))
```

[32]:
```
x_train, x_test, y_train, y_test = train_test_split(
    data, digits.target, test_size=0.5, shuffle=False
)
```

[33]:
```
# Reformat objects for function to read

x_train_new = x_train.T
x_test_new = x_test.T

# Remove the last column because x_train and x_test have a different amount of cols
x_test_new = x_test_new[:, :-1]
```

[34]:
```
y_train_new = -1 * np.ones((10, y_train.shape[0]))
for index, i in enumerate(y_train):
    y_train_new[i , index] = 1

y_train_new.shape
```

[34]:
```
(10, 898)
```

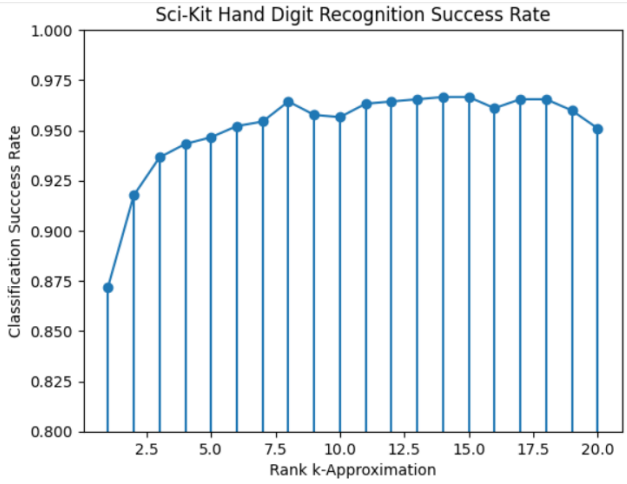
[35]:
```
y_test_new = -1 * np.ones((10, y_test.shape[0]))
for index, i in enumerate(y_test):
    y_test_new[i , index] = 1

# Remove 1 column so that our x and y datasets have the same number of cols
y_test_new = y_test_new[:, :-1]
y_test_new.shape
```

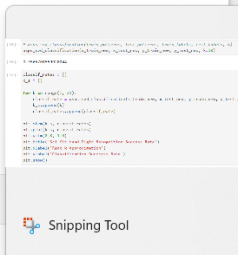
[35]:
```
(10, 898)
```

[36]:
```
# usps_svd_classification(train_patterns, test_patterns, train_labels, test_labels, k)
usps_svd_classification(x_train_new, x_test_new, y_train_new, y_test_new, k=10)
```

[36]:
```
0.9565701559020044
```

[37]:
```
classif_rates = []
k_s = []

for k in range(1, 21):
    classif_rate = usps_svd_classification(x_train_new, x_test_new, y_train_new, y_test_new, k)
    k_s.append(k)
    classif_rates.append(classif_rate)

plt.stem(k_s, classif_rates)
plt.plot(k_s, classif_rates)
plt.ylim(0.8, 1.0)
plt.title('Sci-Kit Hand Digit Recognition Success Rate')
plt.xlabel('Rank k-Approximation')
plt.ylabel('Classification Succcess Rate')
plt.show()
```





[58]:
```
max_val, max_index = max((val, index) for index, val in enumerate(classif_rates))
print('The maximum classification rate is', max_val, 'at k =', max_index+1)
```

```
The maximum classification rate is 0.9665924276169265 at k = 15
```

[54]:
```
# Using k=17, the maximum classification rate:

sci_k = 15
sci_train_u = np.zeros((64, sci_k, 10))

for j in range(10):
    sci_train_u[:,:,j], tmp, tmp2 = svds(x_train_new[:,y_train_new[j,:] == 1], sci_k)
```

[55]:
```
sci_test_svd = np.zeros((sci_k, 898, 10))

for j in range(10):
    sci_test_svd[:, :, j] = sci_train_u[:, :, j].T @ x_test_new
```

[56]:
```
# Compute the approximations:
sci_rank_k_approx = np.zeros((64, 898, 10))

for j in range(10):
    sci_rank_k_approx[:, :, j] = sci_train_u[:, :, j] @ sci_test_svd[:, :, j]
```

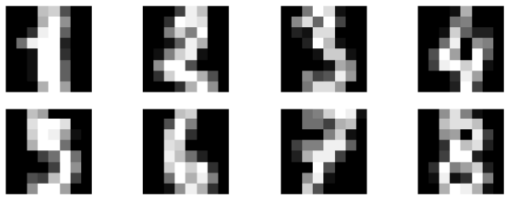
[57]:
```
sci_test_svdres = []

for i in range(898):  # 898 train_patterns is 256x4649
    original = x_test_new[:, i]
    row = []
    for j in range(10):
        sci_rank_k_approx_col = sci_rank_k_approx[:, i, j]
        diff = np.linalg.norm(original - sci_rank_k_approx_col, ord=2)
        row.append(diff)
    sci_test_svdres.append(row)

sci_test_svdres = np.column_stack(sci_test_svdres)
```

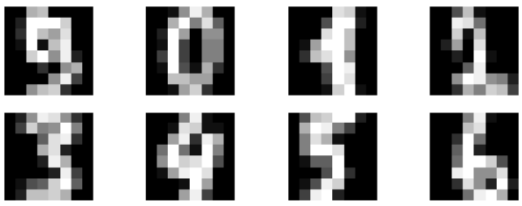
[58]:
```
sci_test_svd_pred = []

for i in range(898):  # 898 cols in sci_test_svdres
    index = np.argmin(sci_test_svdres[:, i])
    sci_test_svd_pred.append(index)

sci_test_svd_pred = np.reshape(sci_test_svd_pred, (1, 898))
```

[59]:
```
# for each valid digit (0, 1, ..., 9) count how many of those digits we correctly guessed in test_classif_res and turn it into a
# vector with the frequency of each digit at it's column location in v.
sci_test_svd_confusion = []

for k in range(10):  # 10 because 10 total digits (0, 1, ..., 9)

    tmp = sci_test_svd_pred[0, np.where(y_test_new[k, :] == 1)[0]]
    tmp_flat = tmp.flatten()
    unique_elements, counts = np.unique(tmp_flat, return_counts=True)

    # Add the frequencies of each digit into the row_k
    row_k = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    for val, count in zip(unique_elements, counts):
        row_k[val] = count
    sci_test_svd_confusion.append(row_k)

sci_test_svd_confusion = np.array(sci_test_svd_confusion)
sci_test_svd_confusion
```

[59]:
```
array([[88,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 89,  1,  0,  0,  0,  0,  0,  1,  0],
       [ 0,  0, 86,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  1, 87,  0,  0,  0,  1,  2,  0],
       [ 1,  0,  0,  0, 85,  0,  0,  2,  0,  4],
       [ 0,  0,  0,  0,  0, 86,  0,  0,  0,  5],
       [ 0,  0,  0,  0,  0,  0, 91,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, 89,  0,  0],
       [ 0,  8,  0,  0,  0,  0,  0,  0, 79,  0],
       [ 0,  0,  0,  3,  0,  1,  0,  0,  0, 88]], dtype=int64)
```

[48]:
```
# Display the original figures in Sci-Kit

fig, axes = plt.subplots(4, 4)
for i in range(4):
    for j in range(4):
        col = i * 4 + j + 1
        axes[i, j].imshow(np.reshape(x_train_new[:, col], (8, 8)), cmap = 'grey')
        axes[i, j].axis('off')
plt.show()
```





```
[50]: # Make train_aves_x from the Sci-Kit data

train_aves_x = []
fig, axes = plt.subplots(2, 5)

for i in range(2):
    for j in range(5):
        digit = 5*i + j
        digit_mean = np.mean(x_train_new[:,y_train_new[digit,:]==1], axis=1)

        # Reshape digit_mean to have 1 column
        digit_mean = np.reshape(digit_mean, (64, 1))
        train_aves_x.append(digit_mean)

        # Reshape digit_mean for image
        digit_img = np.reshape(digit_mean, (8, 8))
        axes[i, j].imshow(digit_img, cmap = 'grey')
        axes[i, j].axis('off')
plt.show()
train_aves_x = np.concatenate(train_aves_x, axis=1)
train_aves_x.shape
```

[50]: (64, 10)

```
[39]: # Testing random things

n_samples
```

[39]: 1797

```
[44]: print(x_train_new.shape)
x_train_new
```

(64, 898)

```
[44]: array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  1.,  0.],
       [ 5.,  0.,  0., ...,  2., 12.,  0.],
       ...,
```

```
       ...,
       [ 0., 10., 16., ..., 14.,  0.,  3.],
       [ 0.,  0.,  9., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

[45]:
```
print(x_test_new.shape)
x_test_new
```

```
(64, 898)
```

[45]:
```
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 1.,  6.,  0., ...,  6.,  1.,  2.],
       ...,
       [12.,  6.,  2., ...,  6.,  6., 12.],
       [ 1.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

[46]:
```
print(y_train_new.shape)
y_train_new
```

```
(10, 898)
```

[46]:
```
array([[ 1., -1., -1., ..., -1., -1., -1.],
       [-1.,  1., -1., ..., -1., -1., -1.],
       [-1., -1.,  1., ..., -1., -1., -1.],
       ...,
       [-1., -1., -1., ..., -1., -1., -1.],
       [-1., -1., -1., ..., -1., -1., -1.],
       [-1., -1., -1., ...,  1., -1., -1.]])
```

[47]:
```
print(y_test_new.shape)
y_test_new
```

```
(10, 898)
```

[47]:
```
array([[-1., -1., -1., ...,  1., -1., -1.],
       [-1., -1., -1., ..., -1., -1., -1.],
       [-1., -1., -1., ..., -1., -1., -1.],
       ...,
       [-1., -1., -1., ..., -1., -1., -1.],
       [ 1.,  1., -1., ..., -1.,  1., -1.],
       [-1., -1., -1., ..., -1., -1.,  1.]])
```