Handwritten Digit Recognition

Karla Cornejo Arguta

December 8, 2024

Pattern recognition is widely used in the betterment of programs; there are different types of patterns that can be recognized through computers, such as image recognition, speech recognition, biometrics, medical diagnosis, network security, and text analysis. In this project, we focus on image recognition and we use this to read handwritten digits and train different classification programs such as simple classification, SVD classification, and K-Nearest Neighbors Classification. We assess the accuracy of each on two different handwritten digits datasets one from the US Postal Service and another from the Modified National Institute of Standards and Technology database.

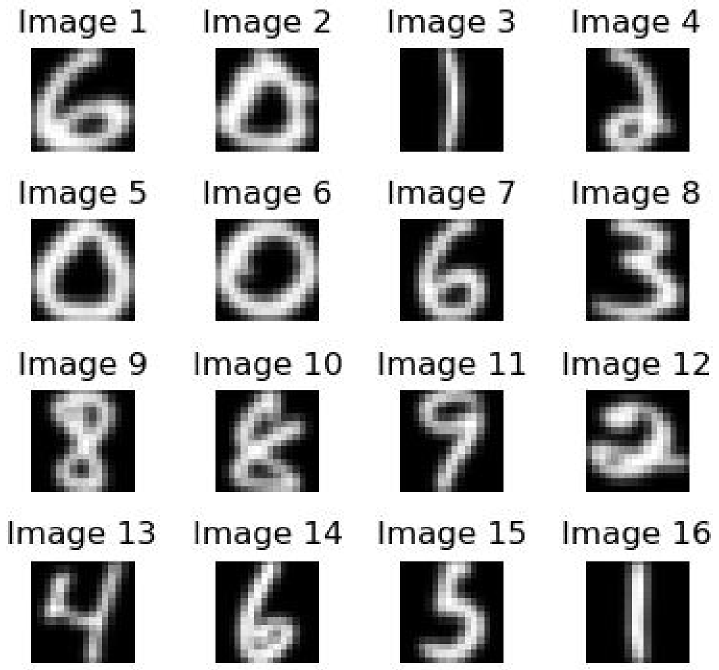
1 Introduction

Handwritten Digit Recognition seeks to train a model that recognizes an image of a handwritten digit as a single digit through the analysis of the pixels in each picture. If the pixels are similar, then the image must belong to a label. This works the same for any image recognition; the program reads the patterns in the pixels, recognizes the differences and similarities in each, and then classifies the image as the class that has the least error. In this project, we are looking at two different data sets: the USPS Data set and the MNIST data set. Both are encoded as flattened images of a handwritten number. We are training a Simple Classification Model and an SVD Classification model in the USPS data set. Then we assess the accuracy of the SVD model in a larger data set such as the MNIST data set and compare accuracy to that of K-Nearest Neighbors which is a slightly more accurate classification model than SVD. In doing this, we want to test the accuracy of each model for a moderately large data set to a large data set.

2 Data Set

The data used was obtained at the Center of Excellence in Document Analysis and Recognition (CEDAR) as part of a project sponsored by the US Postal Service. The data consists of 9298 handwritten digits between 0 and 9; each digit contains a 16×16 pixel image with a label identifying the digit. For the implementation of classification, the data was split into two equal sets, one used to train the classification model and the other to test the classification model. Pixel values are normalized to be between -1 and 1, and each digit image is represented as a one-dimensional vector of length 256. From this, we obtain four different matrices, two pattern matrices of size 256×4649 each, and two label matrices of size 10×4649 each. One pattern and one label matrix is used as part of the training data; each column in the pattern matrix represents an image, and each column on the label matrix represents a label for the corresponding image in the pattern matrix. Below, in figure 1, we can see a visual representation of the first 16 images in the training data set.

Figure 1: First 16 images in training data set

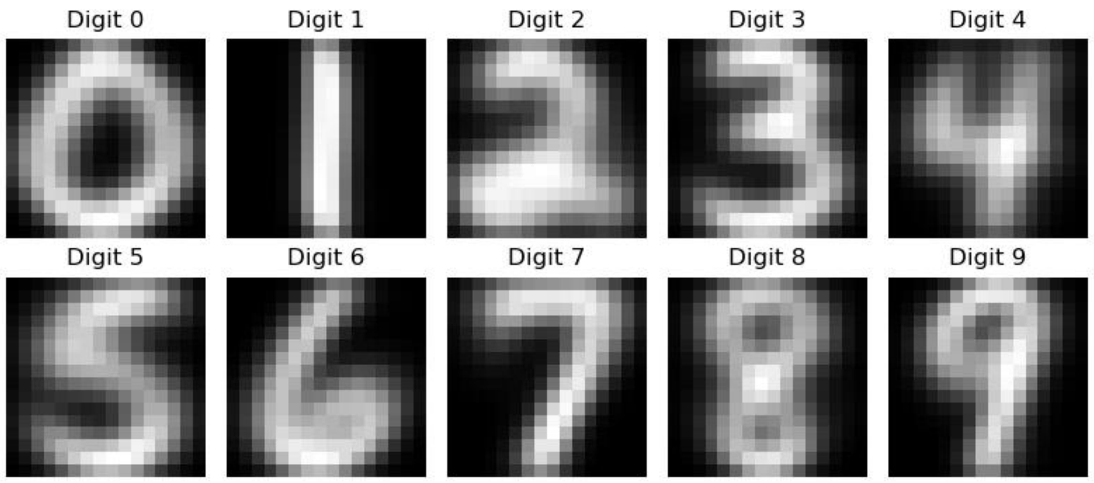


3 Methods

3.1 Simple Classification

For the simple classification method, we compute the mean (average) of each digit in the training data set and from the test data set, we classify each digit as j where $0 \leq j \leq 9$ if the digit in the data set is close to the mean of each digit j. An image of the means for each digit is displayed in figure 2:

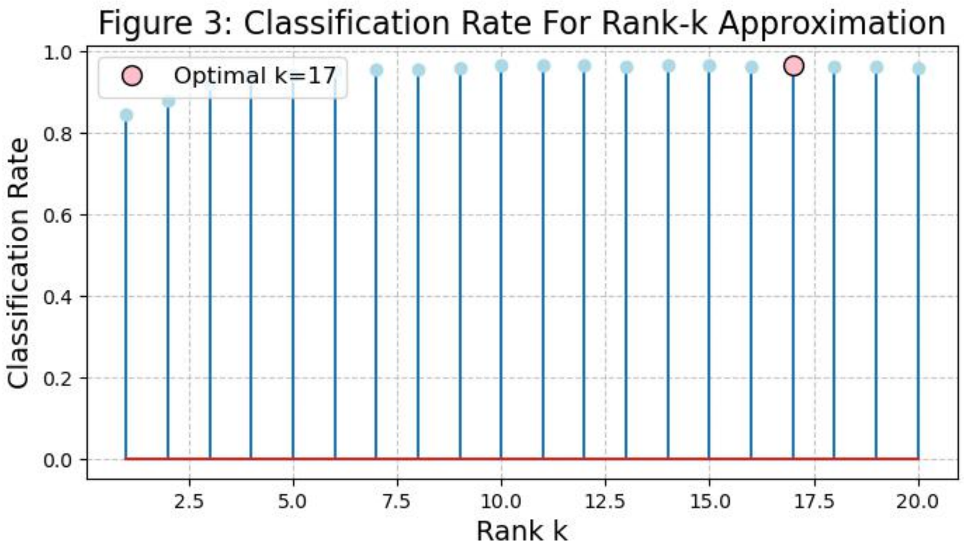
Figure 2: Digit Mean Images



The intuition is that if each vector in the test data set is close to a mean digit it will have similar pixel data and the images will look similar. In our program, we do this classification by first computing the average of each vector i in the training data set that corresponds to the label where the digit $= j$. We achieve this by obtaining the vector in the training patterns data set corresponding to the boolean value where the elements in the training label are 1 through a for loop. For our classifier, we classify each test vector as a digit j where $0 \leq j \leq 9$ by subtracting the means from each test pattern vector and then computing the Euclidean distance between each digit. In other words, the 2-norm such that $||x_i - y_j||_2$ where x_i is the i vector in the test data set and y_j is the mean vector for digit j. After this, we obtain the classifier result by obtaining the minimum across each vector norm and classify x_i as digit j.

3.2 SVD Classification

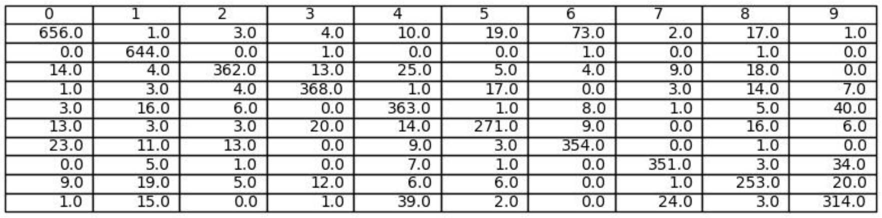
For our next method, we used SVD classification. This method is more complex but also more precise. Singular value decomposition comprises of decomposing a matrix into three different matrices $U, \sum, V^T$, the left singular vectors, singular values, and the right singular vectors, respectively. For the classification method, we only need the left singular vectors U. First, we pool the decomposition up to the kth-value, where k is the optimal value that gives the most accurate decomposition to recognize the digits using the U matrix and test data. After we have the kth-value decomposition we train the data by computing the 2-norm of residual errors. The way this can be done is by obtaining the k expansion coefficient that is such $U_k^{(j)}(U_k^{(j)T} x_i$ which gives the best answer to the least squares problem if x_i belongs to class j (in this case x_i is digit j). Using this expansion coefficient we subtract it from y_i which represents each test digit, the idea is that if the 2-norm of this difference is significantly smaller than any other, then digit i belongs to class j. We can compute the 2-norm of residual errors by $||y_i - U_k^{(j)}(U_k^{(j)T} x_i||_2$. After obtaining this error we obtain the minimum value in each digit and classify the digit as j where $0 \leq j \leq 9$ corresponding to the row index of the minimum value. Then we obtain the accuracy rate at which the classifier accurately predicts the label, running this k times we can obtain the kth optimal value at which the accuracy rate is the highest. Rendering the program for $1 \leq k \leq 20$ we obtained optimal value $k = 17$ as seen below in figure 3:



4 Results

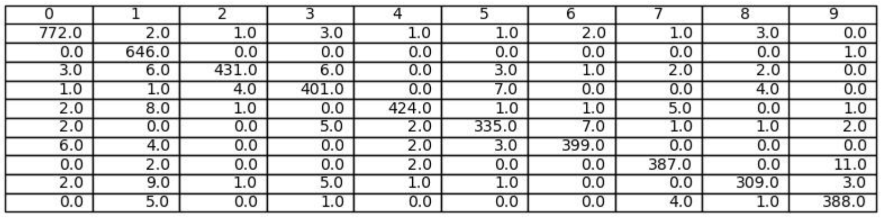
From our Simple Classification method, we obtained the accuracy table in Figure 4 and an accuracy rate of 84.66%. From our SVD Classification Method, we obtained the accuracy table in Figure 5 and an accuracy rate of 96.62% for our optimal k-value of 17.

Figure 4: Confusion Table for Simple Classification



0	1	2	3	4	5	6	7	8	9
656.0	1.0	3.0	4.0	10.0	19.0	73.0	2.0	17.0	1.0
0.0	644.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	0.0
14.0	4.0	362.0	13.0	25.0	5.0	4.0	9.0	18.0	0.0
1.0	3.0	4.0	368.0	1.0	17.0	0.0	3.0	14.0	7.0
3.0	16.0	6.0	0.0	363.0	1.0	8.0	1.0	5.0	40.0
13.0	3.0	3.0	20.0	14.0	271.0	9.0	0.0	16.0	6.0
23.0	11.0	13.0	0.0	9.0	3.0	354.0	0.0	1.0	0.0
0.0	5.0	1.0	0.0	7.0	1.0	0.0	351.0	3.0	34.0
9.0	19.0	5.0	12.0	6.0	6.0	0.0	1.0	253.0	20.0
1.0	15.0	0.0	1.0	39.0	2.0	0.0	24.0	3.0	314.0

Figure 5: Confusion Table for SVD Classification



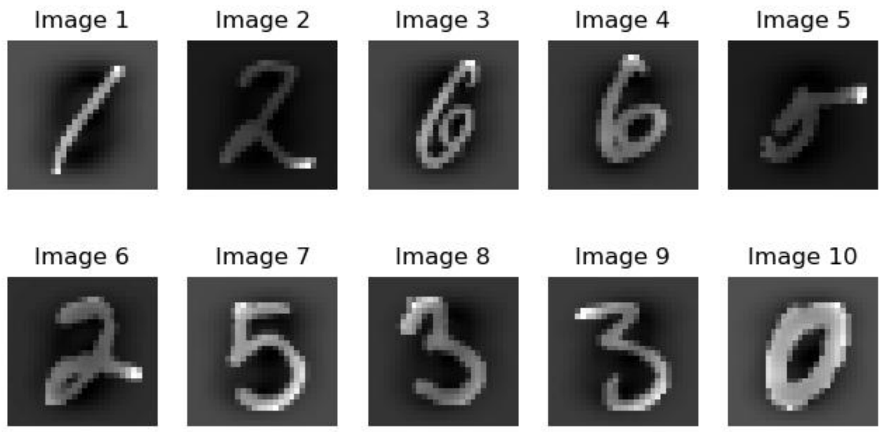
0	1	2	3	4	5	6	7	8	9
772.0	2.0	1.0	3.0	1.0	1.0	2.0	1.0	3.0	0.0
0.0	646.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0
3.0	6.0	431.0	6.0	0.0	3.0	1.0	2.0	2.0	0.0
1.0	1.0	4.0	401.0	0.0	7.0	0.0	0.0	4.0	0.0
2.0	8.0	1.0	0.0	424.0	1.0	1.0	5.0	0.0	1.0
2.0	0.0	0.0	5.0	2.0	335.0	7.0	1.0	1.0	2.0
6.0	4.0	0.0	0.0	2.0	3.0	399.0	0.0	0.0	0.0
0.0	2.0	0.0	0.0	2.0	0.0	0.0	387.0	0.0	11.0
2.0	9.0	1.0	5.0	1.0	1.0	0.0	0.0	309.0	3.0
0.0	5.0	0.0	1.0	0.0	0.0	0.0	4.0	1.0	388.0

Here we can see that the SVD Classification method works better at accurately predicting the label of a test image over a Simple Classification method. The table for SVD Classification has larger numbers accross the diagonal indicating higher accuracy as well as having an accuracy rate higher than the Simple Classification. This is because classifying using the SVD uses residual errors over euclidean distance leading to smaller errors. However, it is still not 100% accurate.

5 Additional Program Testing

Knowing that SVD Classification is superior to Simple Classification, how does it compare when using larger data sets? Is it still extremely accurate, or does accuracy decrease? We did additional testing using the MNIST data set, which is also composed of handwritten digits. This data set contains 70,000 data points of pattern vectors, each of length 784. Each digit vector has a corresponding label, which is encoded as a digit j where $0 \leq j \leq 9$. We did a similar split as with the USPS data set, where we used 50% of the data to train the model and 50% to test the model. This left us with two pattern data sets, each of size $784x35000$, and two label data sets, each encoded as matrix size $35000x1$. The first 10 images in the training data set are shown in Figure 6.

Figure 6: First 10 images in MNIST training data set



To test the accuracy of SVD, we used SVD Classification and K-Nearest Neighbor Classification. The K-Nearest Neighbor Classification has been shown to be slightly more accurate than the SVD classification in the USPS Data Set (Saito, Lecture 21). We followed the SVD Classification Process above to obtain the accuracy of SVD classification and optimal value. For the K-NN Classification method, we take the distance of the test digit y_j to all of the training digits and then look at the k-nearest neighbors of y_j and take the majority label of the training data to assign a label to the test digit. In Python, we used the KNeighborsClassifier from the sklearn.neighbors package to compute the classification. After, we applied this from k = 1 to k = 10 to obtain the optimal k-value. We obtained the k-values shown in Figure 7 from both SVD and K-NN classifications. Here, we see that the optimal value in SVD Classification is 13 and the optimal value for K-Nearest Neighbors is 3.

Figure 7: Classification Rates and Optimal k using SVD and KNN Classification on MNIST Data Set

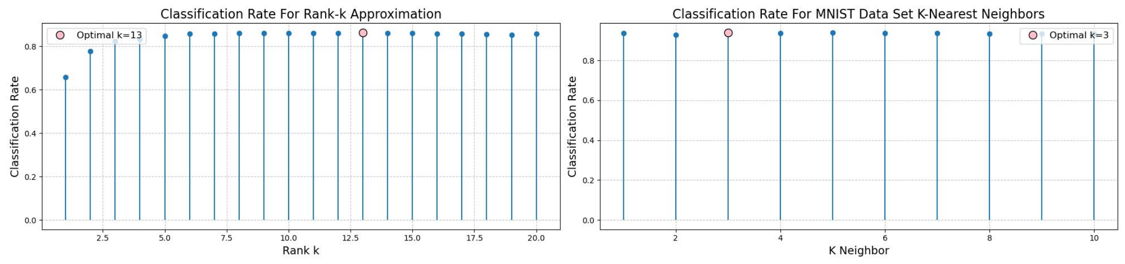
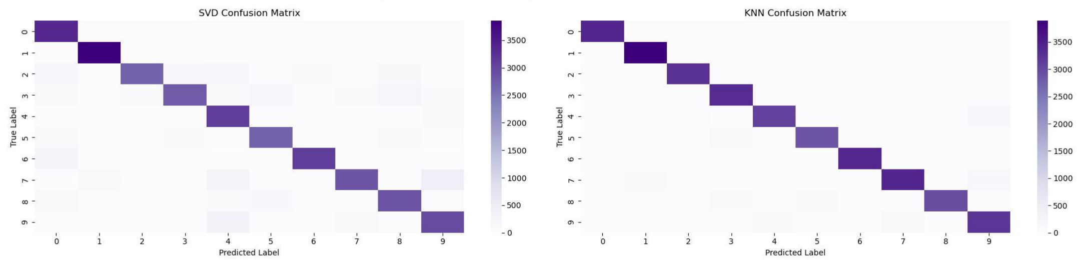


Figure 8: Confusion Matrices using SVD and KNN Classification on MNIST Data Set



Following suit, we computed the accuracy tables (shown as a heatmap due to large-scale numbers) in Figure 8. Here we can see that the heatmap for the SVD accuracy has slightly more color distributed outside the diagonal, indicating more errors than K-NN. The accuracy rate obtained from SVD was 86.35% and from K-NN was 93.90%. Overall both prediction rates are high, but the SVD Classification rate was significantly lower than the K-NN Classification in this case. Indicating that SVD Classification might not be the best option in this instance.

6 Conclusions

For large data sets, doing SVD might still lead to accurate results, but better recognition programs might work better and lead to more accurate results, such as the K-Nearest Neighbor method. However, when data sets are smaller yet relatively large, Singular Value Decomposition yields highly accurate results and carries less computation time than K-Nearest Neighbors. Though accuracy is slightly lower, computation time decreases with SVD, yielding SDV Classification as an appropriate method for relatively large data sets. For more complicated programs that require pattern recognition, there might be better classification models that both decrease computation time and increase accuracy. Such as for the MNIST data the convolutional neural network algorithm is usually used to analyze the data set.

Sources

Eldén, L. (2007). 10. Classification of Handwritten Digits. In Society for Industrial and Applied Mathematics eBooks (pp. 113–128). https://doi.org/10.1137/1.9780898718867.ch10

Saito, M. (2017). Lecture 21: Classification of Handwritten Digits. University of California, Davis. https://www.math.ucdavis.edu/~saito/courses/167.s17/Lecture21.pdf

Subasi, A. (2020). Other classification examples. In Elsevier eBooks (pp. 323–390). https://doi.org/10.1016/b978-0-12-821379-7.00005-9

Code Appendix

train_patterns = The patterns in the USPS training data set, matrix with 4649 vectors of length 256.
test_patterns = The patterns used to test the algorithm in the USPS data set, the same size as the training data set.
train_labels = The labels associated with the training patterns in the USPS data set.
test_labels = The labels associated with the testing patterns in the USPS data set.
x_train = The patterns in the MNIST training data set.
x_test = The patterns in the MNIST test data set.
y_train = The labels associated with the MNIST training data set.
y_test = The labels associated with the MNIST test data set.

[ ]:
```
#!/usr/bin/env python
# coding: utf-8

# # Handwritten Digit Recognition

# ###  **a)** Download the handwritten digit database from Canvas: usps.mat.␣
↪Then, load this file into your Python session.
# ### This file contains 4 arrays: `train_patterns`, `test_patterns` of size␣
↪256 × 4649, and `train_labels`,
# ### `test_labels` of size 10 × 4649. The `train_patterns` and `test_patterns`␣
↪contain a raster scan of the 16 × 16 gray level pixel
# ### intensities, which have been normalized to range within [-1, 1]. The␣
↪train labels and test labels variables contain the ground
# ### truth information of the digit images. That is, if the handwritten digit␣
↪image `train_patterns[:, j]` truly represents digit $i$,
# ### then `train_labels[i, j]` is +1, and all the other entries of␣
↪`train_labels[:, j]` are -1.

# In[3]:


# Autograder Code cell number 0
from scipy import io
import numpy as np

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']



# In[4]:
```

```
## importing the usps data set and extracting the training and test sets
data_dict = io.loadmat('/Users/karlacornejoargueta/Downloads/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']


# ### Now, display the first 16 images in train patterns using plt.subplots(4,␣
 ↪4) and `plt.imshow` functions in Python. Note:
# ### You need to reshape each column into a matrix of size 16 × 16 in order to␣
 ↪display it correctly.

# ### In your report you should explain what `train_patterns, test_patterns,␣
 ↪train_labels,` and `test_labels` represent and state
# ### the dimensions.

# In[7]:


## using matplotlib to plot the images corresponding to the first 16 images
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
fig1, axes = plt.subplots(4, 4, figsize=(5, 5))
for i, ax in enumerate(axes.flat):
    col = train_patterns[:, i]     ## extracting columns 1-16 and reshaping to␣
 ↪be 16x16
    col = col.reshape(16,16)
    ax.imshow(col, cmap = 'gray')
    ax.set_title(f'Image {i+1}')
    ax.axis('off')
fig1.suptitle("Figure 1: First 16 images in training data set")

plt.tight_layout()
plt.show()
fig1.savefig("figure1.jpg")


# ### (b) Now, compute the mean digits in the train patterns, put them in a␣
 ↪matrix called `train_aves` of size 256 × 10, and display
# ### these 10 mean digit images using `plt.subplots(2, 5)` and `plt.imshow`.

# ### Note: You can gather (or pool) all the images in `train_patterns`␣
 ↪corresponding to digit $ j (0 \leq j \leq 9)$ by the following:
```

```
# ```
# train_patterns[:, train_labels[j, :] == 1]
# ```

# In[10]:


# Autograder Code cell number 1
# The autograder won't check the images and intermediate results

## computing train_aves as a matrix that stores the averages for each digit
train_aves = np.zeros((256, 10))
for j in range(10):
    digits = train_patterns[:, train_labels[j, :] == 1]  ## matching digits to␣
 ↪the label
    for i, col in enumerate(digits):
        train_aves[i,j]= np.mean(col) ## computing the averages


# In[11]:


## using matplotlib to show the mean images
fig2, axes = plt.subplots(2, 5, figsize=((8, 4)))
for i, ax in enumerate(axes.flat):
    ave = train_aves[:,i].reshape(16,16) ## reshaping into 16x16 pixels
    ax.imshow(ave, cmap = 'gray')
    ax.set_title(f'Digit {i}')
    ax.axis('off')
fig2.suptitle("Figure 2: Digit Mean Images")

plt.tight_layout()
plt.show()
fig2.savefig("figure2.jpg")


# ### (c) Let's conduct the simplest classification experiments as follows:
#
# ### (c.1) First, prepare a matrix called `test_classif` of size 10 × 4649 and␣
 ↪fill this matrix by computing the Euclidean distance
# ### (or its square) between each image in the `test_patterns` and each mean␣
 ↪digit image in `train_patterns.`

# In[13]:


# Autograder Code cell number 2
```

```
# The autograder won't check the intermediate results

## initializing a matrix 10x4649 to include all classifiers
test_classif = np.zeros((10, 4649))

for i in range(10):
    for j in range(test_patterns.shape[1]):
        differences = test_patterns[:,j] - train_aves[:, i] ## taking␣
 ↪differences from test_patterns and the mean images and then 2-norm
        test_classif[i, j] = np.sum(differences ** 2, axis=0)



# ### (c.2) Then, compute the classification results (the predicted label for␣
 ↪`test_patterns`) by finding the position index of the
# ### minimum of each column of `test_classif`. Put the results in a vector␣
 ↪`test_classif_res` of size 1 × 4649

# **Note:** You can find the position index giving the minimum of the j-th␣
 ↪column
# of `test_classif` by `np.argmin` function.

# In[16]:


# Autograder Code cell number 3
# The autograder won't check the intermediate results

## taking the minimum of the classifications for the classification results
test_classif_res = np.zeros((1,4649))
test_classif_res = np.argmin(test_classif, axis=0)


# ### (c.3) Finally, compute the confusion matrix `test_confusion` of size 10 ×␣
 ↪10, and the overall classification rate
# ### (i.e., the ratio of the `test_patterns` with the correct predicted label).

# **Note:** First gather the classification results corresponding to the␣
 ↪***k***th digit by
#
# ```
# tmp = test_classif_ref[np.where(test_labels[k,:] == 1)[0]]
# ```

# This `tmp` array contains the results of your classification of the test␣
 ↪digits whose true digit is $k
```

```
# (0 \leq k \leq 9).$ In other words, if your classification results were␣
 ↪perfect, all the entries of the `tmp` would
# be ***k.*** But in reality, this simplest classification algorithm makes␣
 ↪mistakes, so `tmp` contains values other than ***k.***
# You need to count how many entries have the value ***j*** in `tmp`, $j = 0 :␣
 ↪9.$ That would give you the ***k***th row of the
# `test_confusion` matrix.

# In[20]:


# Autograder Code cell number 4
# The autograder will check the classification rate and test_confusion
# Please name the two variables as ``rate`` and ``test_confusion``

## initializing confusion matrix
test_confusion = np.zeros((10,10))

for k in range(10):
    tmp = test_classif_res[np.where(test_labels[k,:] == 1)[0]] ## checking if␣
 ↪the classification label is the same as the test label
    for j in range(10):
        test_confusion[k, j] = np.sum(tmp == j) ## computing classification␣
 ↪matrix
correct = np.trace(test_confusion)   ## obtaining classification rate from␣
 ↪matrix
total = np.sum(test_confusion)
rate = correct / total
print(rate)


# In[21]:


import pandas as pd

## using pandas to display confusion table
df = pd.DataFrame(test_confusion)

fig, ax = plt.subplots(1, 1, figsize=(10, 3))
ax.axis('off')
ax.table(cellText=df.values, colLabels=df.columns, loc='center')
fig.suptitle("Figure 4: Confusion Table for Simple Classification")
plt.savefig('figure4.jpg')
```

```
# ### (d) Finally, let’s conduct the SVD-based classification experiments.
# ### (d.1) Pool all the images corresponding to the jth digit␣
↪`train_patterns`, compute the rank k SVD of that set of images
# ### (i.e., the first k singular values and vectors), and put the left␣
↪singular vectors (or the matrix U) of jth digit into the array
# ###`train_u` of size 256 × k × 10. For j = 0 : 9, you can do the following:
#
# ```
# train_u[:,:,j], tmp, tmp2 = svds(train_patterns[:,train_labels[j,:]==1], k)
# ```

# In[23]:


from scipy.sparse.linalg import svds
k = 15
## define a function to get the k-th rank svd of traing matrix
def compute_train_svd(train_patterns,train_labels, k):
    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)  ## using svds function from scipy.sparce.linalg
    return train_u
train_u = compute_train_svd(train_patterns,train_labels, k) ## u matrix for␣
↪training data with k =15 for example


# **Note:** We do not need the singualr values and right singular vectors in␣
↪this experiment. The `svds` is the same function we used
## in Homework 7.

# ### (d.2) Now, compute the expansion coefficients of each test digit image␣
↪with respect to the k singular vectors of each train digit
# ### image set. In other words, you need to compute $k \times 10$ numbers for␣
↪each test digit image. Put the results in the 3D array
# ### `test_svd` of size $k × 4649 × 10$. This can be done by:
#
# ```
# for j in range(10):
#     test_svd[:, :, j] = train_u[:, :, j].T @ test_patterns
# ```

# In[26]:


## computing part of the expansion coefficient
```

```
def expansion_coeff(train_u, test_patterns):
    test_svd = np.zeros((k,4649,10))
    for j in range(10):
        test_svd[:, :, j] = train_u[:, :, j].T @ test_patterns
    return test_svd
test_svd = expansion_coeff(train_u, test_patterns)


# ### (d.3) Next, compute the error between each original test digit image and␣
 ↪its rank $k$ approximation using the $j$th digit
# ### images in the training dataset. The idea of this classification is that␣
 ↪if a test digit image should belong to class of $j^*$th
# ### digit if the corresponding rank $k$ approximation is the best␣
 ↪approximation (i.e., the smallest error) among 10 such approximations.
# ### Prepare a matrix `test_svdres` of size 10 × 4649, and put those␣
 ↪approximation errors into this matrix.
#

# **Note:** The rank $k$ approximation of test digits using the $k$ left␣
 ↪singular vectors of the $j$th digit training images
# can be computed by
# ```
# train_u[:,:,j] @ test_svd[:,:,j]
# ```

# In[29]:


## computing the errors between the expansion coefficient and the test patterns
def error(test_patterns, train_u, test_svd):
    test_svdres = np.zeros((10,4649))
    for j in range(10):
        rank_k_approx = train_u[:,:,j] @ test_svd[:,:,j]
        dif = test_patterns - rank_k_approx
        for i in range(test_patterns.shape[1]):
            test_svdres[j, i] = np.linalg.norm(dif[:, i]) ** 2  ## using the␣
 ↪norms as classification
    return test_svdres
test_svdres = error(test_patterns, train_u, test_svd) ## computing for the data␣
 ↪with k = 15


# ### (d.4) Create a function `usps_svd_classification`. For $k$ = 1 : 20, run␣
 ↪the function and compare the overall classification
# ### rate and report your results by plotting a figure of the overall␣
 ↪classification rates versus the rank $k$.
```

```
# ### Which $k$ has the best classification rate?

# In[31]:



# Autograder Code cell number 5
# Note that Autograder will only read the code cell number 0 and forget about␣
↪the results from cell number 1 to 4

from scipy.sparse.linalg import svds

def usps_svd_classification(train_patterns, test_patterns, train_labels,␣
↪test_labels, k):
    """
    Implemented the rank k SVD-based classification on the usps dataset

    INPUT:

    - train_pattern: The training patterns in the data set.
    - test_pattern: The test patterns in the data set.
    - train_label: The training labels in the data set.
    - test_label: The test label in the data set.
    - k: the k-th value for optimal classification.

    OUTPUT:

    - rate: the overall classification rate of the test_patterns
    - test_predict: the predicted label for the test_patterns
    """
    ## implementing same steps as above
    train_u = np.zeros((256, k, 10))
    test_svdres = np.zeros((10,4649))
    test_svd = np.zeros((k,4649,10))
    correct = 0

    for j in range(10):
        train_u[:,:,j], tmp, tmp2 = svds(train_patterns[:,train_labels[j,:
↪]==1], k) ## k rank svd
        test_svd[:, :, j] = train_u[:, :, j].T @ test_patterns
        rank_k_approx = train_u[:,:,j] @ test_svd[:,:,j]    ## expansion␣
↪coefficient
        dif = test_patterns - rank_k_approx                 ## difference from␣
↪expansion coefficient
        for i in range(test_patterns.shape[1]):
            test_svdres[j, i] = np.linalg.norm(dif[:, i]) ** 2  ## 2-norm of␣
↪the difference
```

```
    test_predict = np.argmin(test_svdres, axis=0)         ## prediction label␣
 ↪using the minimum error

    for i in range(test_patterns.shape[1]):
        true_label = np.where(test_labels[:, i] == 1)[0][0]    ## computing the␣
 ↪count of correct labels
        if test_predict[i] == true_label:
            correct += 1
    rate = correct / test_patterns.shape[1]             ## computing accuracy␣
 ↪rate

    return rate, test_predict




# In[32]:


# Autograder Code cell number 6
# The autograder won't check the images
# The autograder will check the classification rates for k from 1 to 20
# Please name the variable as ``rate_list``, which is a numpy.array of size 20.

## initializes matrix of 1x20
rate_list = np.zeros(20)
## computes accuracy rates for k from 1 to 20 using the function above
for k in range(1,21):
    rate, test_predict = usps_svd_classification(train_patterns, test_patterns,␣
 ↪train_labels, test_labels, k)
    rate_list[k-1] = rate


# In[33]:


## using matplot to plot optimal value
optimal_k_index = np.argmax(rate_list)  ## optimal value is the maximum␣
 ↪accuracy rate in the list above

fig3 = plt.figure(figsize=(8, 4))
markerline, stemlines, baseline = plt.stem(range(1, len(rate_list) + 1),␣
 ↪rate_list)  ## creates a stem plot
markerline.set_color('lightblue')
```

```
## highlights the optimal value
plt.plot(
    range(1, len(rate_list) + 1)[optimal_k_index],
    rate_list[optimal_k_index],
    'o',
    color='pink',
    markeredgecolor='black',
    markersize=10,
    label=f'Optimal k={optimal_k_index + 1 }'      ## adding label for optimal␣
 ↪value
)

plt.title("Figure 3: Classification Rate For Rank-k Approximation", fontsize=16)
plt.xlabel("Rank k", fontsize=14)
plt.ylabel("Classification Rate", fontsize=14)

plt.grid(True, linestyle='--', alpha=0.7)
plt.legend(fontsize=12, loc=2)

plt.show()
fig3.savefig("figure3.jpg")


# ### (d.5) Finally, compute the confusion matrix and the overall␣
 ↪classification rate using this best SVD-based classification method
# ### by following the same strategy as Parts (c.2) and (c.3) above. Let's name␣
 ↪this confusion matrix `test_svd_confusion`.

# In the results section of your report, you should at least have the plot from␣
 ↪part (d.4) and report the confusion matrix and the
# overall classification rate for this dataset by the different methods (the␣
 ↪simplest classification and only pick the best ***k*** SVD).

# In[36]:



# Autograder Code cell number 7
# The autograder will check the classification rate and test_confusion for best␣
 ↪k
# Please name the two variables as ``rate`` and ``test_confusion``

## running the svd classification function for optimal value
true_labels = np.argmax(test_labels, axis=0)
rate, test_predict = usps_svd_classification(train_patterns, test_patterns,␣
 ↪train_labels, test_labels, 17)
```

```
## initializing confusion matrix
test_confusion = np.zeros((10,10))

for i in range(len(test_predict)):          ## creates confusion table in which␣
 ↪the predicted labels are the same as the true labels
    true_class = true_labels[i]
    predicted_class = test_predict[i]

    test_confusion[true_class, predicted_class] += 1


# In[37]:


df = pd.DataFrame(test_confusion)

## using panda to visualize the table
fig5, ax = plt.subplots(1, 1, figsize=(10, 3))
ax.axis('off')
ax.table(cellText=df.values, colLabels=df.columns, loc='center')
fig5.suptitle("Figure 5: Confusion Table for SVD Classification")
plt.savefig('figure5.jpg')
print(rate_list[16])


# ### (e) Try the method on another larger handwritten-digit dataset (for␣
 ↪example, MNIST dataset) and test other machine learning
# ### algorithm on the dataset.

# In[39]:


## using sklearn to import MNIST data set and K-NN Classifier
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score


# In[40]:


## loading the MNIST dataset
mnist = datasets.fetch_openml("mnist_784")
x = mnist.data.to_numpy()
y = mnist.target.astype(int)
```

```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.5,␣
 ↪random_state=42)

## scaling and transposing data to make computations easier and more accurate
scaler = StandardScaler()
x_train = scaler.fit_transform(x_train).transpose()
x_test = scaler.transform(x_test).transpose()

y_test = y_test.reset_index(drop=True)
y_train = y_train.reset_index(drop=True)


# In[41]:


## plotting the first 10 images using matplotlib
y_train_np = y_train.to_numpy() if isinstance(y_train, pd.Series) else y_train␣
 ↪## changing to numpy array so it can be visualized
fig5, axes = plt.subplots(2, 5, figsize=(8, 4))
for i, ax in enumerate(axes.flat):
    col = x_train[:, i]
    col = col.reshape(28,28)        ## reshaping into 28x28 images
    ax.imshow(col, cmap = 'gray')
    ax.set_title(f'Image {i+1}')
    ax.axis('off')

fig5.suptitle("Figure 6: First 10 images in MNIST training data set")
plt.show()
fig5.savefig("figure6.jpg")


# In[42]:


def mnist_svd_classification(train_set, test_set, train_labels, test_labels, k):
    """
    Implement rank-k SVD-based classification on the MNIST dataset

    INPUT:
    - train_set: The training patterns in the data set.
    - test_set: The test patterns in the data set.
    - train_label: The training labels in the data set.
    - test_label: The test label in the data set.
    - k: rank for the SVD approximation.

    OUTPUT:
```

```
    - rate: classification rate (accuracy) on the test set.
    - test_predict: predicted labels for the test patterns.
    """
    ## adapting the svd classification to fit the MNIST data set
    test_svdres = np.zeros((10, test_set.shape[1]))
    correct = 0

    U_all = np.zeros((train_set.shape[0], k, 10))

    for j in range(10):
        X_j = train_set[:, train_labels == j] ## using the training data

        U_all[:, :, j], _, _ = svds(X_j, k=k)  ## computing SVD and saving U

        test_svd = U_all[:, :, j].T @ test_set

        rank_k_approx = U_all[:, :, j] @ test_svd  ## expansion coefficient
        dif = test_set - rank_k_approx

        test_svdres[j, :] = np.linalg.norm(dif, axis=0) ** 2 ## residual error


    test_predict = np.argmin(test_svdres, axis=0)  ## classifying on the␣
 ↪minimum error

    correct = np.sum(test_predict == test_labels)
    rate = correct / test_set.shape[1]          ## obtaining accuracy rate

    return rate, test_predict


# In[ ]:


## obtaining accuracy rate for k from 1 to 20 to find optimal value
rate_list_svd = np.zeros(20)
for k in range(1,21):
    rate, test_predict = mnist_svd_classification(x_train, x_test, y_train,␣
 ↪y_test, k)
    rate_list_svd[k-1] = rate


# In[ ]:


def knn_mnist_classification(x_train, x_test, y_train, y_test, n=3):
    """
    Implement KNN-based classification on the MNIST dataset

    INPUT:
    - x_train: The training patterns in the data set.
    - x_test: The test patterns in the data set.
    - y_train: The training labels in the data set.
    - y_test: The test label in the data set.
    - n: number of neighbors for the KNN classifier (default is 3).

    OUTPUT:
    - rate: classification rate (accuracy) on the test set.
    - test_predict: predicted labels for the test patterns.
    """

    ## computing knn classification to the MNIST data set

    x_train = x_train.transpose()  ## transposing the data for the dimensions␣
 ↪to match with the sklearn knn function
    x_test = x_test.transpose()

    knn = KNeighborsClassifier(n_neighbors=n) ## using sklearn to obtain␣
 ↪classifier with n neighbors

    knn.fit(x_train, y_train)  ## fitting the classifier onto the training data

    test_predict = knn.predict(x_test)  ## predicting based on the test data

    rate = accuracy_score(y_test, test_predict)  ## obtaining the rate of␣
 ↪accuracy

    return rate, test_predict


# In[ ]:


## computing the knn classification for n = 1 to 10 to find optimal value
rate_list_knn = np.zeros(10)
for k in range(1,11):
    rate, test_predict = knn_mnist_classification(x_train, x_test, y_train,␣
 ↪y_test, k)
    rate_list_knn[k-1] = rate


# In[ ]:
```

```
## visualizing optimal values for both knn and svd classification
optimal_k_index_svd = np.argmax(rate_list_svd)  ## taking the maximum accuracy␣
 ↪rate in each list
optimal_k_index_knn = np.argmax(rate_list_knn)

fig7, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 5))

ax1.stem(range(1, len(rate_list_svd) + 1), rate_list_svd, markerfmt='o',␣
 ↪basefmt=" ") ## plotting a stem plot for svd on MNIST
ax1.plot(
    range(1, len(rate_list_svd) + 1)[optimal_k_index_svd],
    rate_list_svd[optimal_k_index_svd],
    'o',
    color='pink',
    markeredgecolor='black',
    markersize=10,
    label=f'Optimal k={optimal_k_index_svd + 1}'
)

ax1.set_title("Classification Rate For Rank-k Approximation", fontsize=16)
ax1.set_xlabel("Rank k", fontsize=14)
ax1.set_ylabel("Classification Rate", fontsize=14)
ax1.grid(True, linestyle='--', alpha=0.7)
ax1.legend(fontsize=12, loc=2)

ax2.stem(range(1, len(rate_list_knn) + 1), rate_list_knn, markerfmt='o',␣
 ↪basefmt=" ") ## plotting a stem plot for knn on MNIST
ax2.plot(
    range(1, len(rate_list_knn) + 1)[optimal_k_index_knn],
    rate_list_knn[optimal_k_index_knn],
    'o',
    color='pink',
    markeredgecolor='black',
    markersize=10,
    label=f'Optimal k={optimal_k_index_knn + 1}'
)

ax2.set_title("Classification Rate For MNIST Data Set K-Nearest Neighbors",␣
 ↪fontsize=16)
ax2.set_xlabel("K Neighbor", fontsize=14)
ax2.set_ylabel("Classification Rate", fontsize=14)
ax2.grid(True, linestyle='--', alpha=0.7)
ax2.legend(fontsize=12, loc=1)

fig7.suptitle("Figure 7: Classification Rates and Optimal k using SVD and KNN␣
 ↪Classification on MNIST Data Set")
plt.tight_layout()
plt.show()
fig7.savefig("figure7.jpg")


# In[ ]:


## initializing the confusioon tables for both svd and knn and their optimal␣
 ↪values
rate_svd, test_predict_svd = mnist_svd_classification(x_train, x_test, y_train,␣
 ↪y_test, np.argmax(rate_list_svd)+1)
rate_knn, test_predict_knn = knn_mnist_classification(x_train, x_test, y_train,␣
 ↪y_test, np.argmax(rate_list_knn)+1)
test_confusion_svd = np.zeros((10,10))
test_confusion_knn = np.zeros((10,10))


# In[ ]:


## computing both tables
for i in range(len(test_predict_svd)):
    true_class = y_test[i]
    predicted_class = test_predict_svd[i]

    test_confusion_svd[true_class, predicted_class] += 1 ## for svd

for i in range(len(test_predict_knn)):
    true_class = y_test[i]
    predicted_class = test_predict_knn[i]

    test_confusion_knn[true_class, predicted_class] += 1 ## for knn


# In[ ]:


import seaborn as sns
fig8, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 5))

## using seaborn to visualize tables as heatmaps
## data set is too large to represent in a regular table
sns.heatmap(test_confusion_svd, annot=False, cmap='Purples', xticklabels=np.
 ↪arange(10), yticklabels=np.arange(10),ax = ax1)

ax1.set_title('SVD Confusion Matrix')
ax1.set_xlabel('Predicted Label')
```

```
ax1.set_ylabel('True Label')  ## SVD


## visualizing knn
sns.heatmap(test_confusion_knn, annot=False, cmap='Purples', xticklabels=np.
 ↪arange(10), yticklabels=np.arange(10),ax = ax2)

ax2.set_title('KNN Confusion Matrix')
ax2.set_xlabel('Predicted Label')
ax2.set_ylabel('True Label')

fig8.suptitle("Figure 8: Confusion Matrices using SVD and KNN Classification on␣
 ↪MNIST Data Set")
plt.tight_layout()
plt.show()
fig8.savefig("figure8.jpg")
```