Department of Informatics

M.Sc. in Data Science

Numerical optimization and Large Scale Linear Algebra

Classification of Handwritten Digits

Instructors: Paris Vassalos

Authors: Antonios Mavridis (p3352215)

04/06/2023

Table of Contents

1 Introduction

Handwritten digit classification is an essential task in the field of pattern recognition and machine learning. The ability to automatically recognize and classify handwritten digits has numerous practical applications, such as optical character recognition, postal automation, and document analysis. In this report, we present an algorithm for the classification of handwritten digits using the Singular Value Decomposition (SVD) technique.

The primary objective of this assignment is to construct an algorithm that accurately classifies handwritten digits based on a training set. The algorithm utilizes the SVD of each class matrix and determines the optimal number of singular vectors to use as a basis for classification. We measure the classification accuracy by considering the relative residual vector in the least squares problem.

The specific tasks involved in this assignment are as follows:

1. Tuning the algorithm for classification accuracy: We explore the relationship between the number of basis vectors used and the percentage of correctly classified digits. This analysis helps determine the optimal number of basis vectors to achieve the highest classification accuracy.
2. Assessing the difficulty of classifying digits: We investigate whether all digits are equally easy or difficult to classify. Additionally, we examine some of the difficult cases, noting that poorly written digits pose challenges in classification.
3. Analyzing the singular values of different classes: We evaluate the singular values of the classes and explore the possibility of using varying numbers of basis vectors for different classes. Through experiments, we assess whether it is beneficial to utilize fewer basis vectors for specific classes.

The dataset provided consists of training and test data:

1. "dzip.txt" and "azip.txt": These files contain the training data. "dzip.txt" holds the corresponding labels (digits), while "azip.txt" stores the training images represented as vectors.
2. "dtest.txt" and "testzip.txt": These files contain the test data, with "dtest.txt" containing the labels and "testzip.txt" storing the test images.

To facilitate the visualization of images, we will create a Python function similar to "Ima2.m" provided in the assignment. In addition to the required tasks, we will explore an optional two-stage algorithm with SVD. This approach aims to optimize the test phase by comparing the unknown digit only to the first singular vector in each class. We will analyze the effectiveness of this variant and determine how frequently the second stage is unnecessary.

Moreover, we will enhance the image viewing capability by modifying the "ima2.m" function to display 20 x 20 images.

In conclusion, this report presents an algorithm for handwritten digit classification using SVD. Through experimentation and analysis, we aim to achieve accurate classification results while exploring potential optimizations for the classification process.

2 Question 1

According to the publication "Matrix Methods in Data Mining and Pattern Recognition," precise procedures must be followed to compute the SVD of each class matrix and subsequently classify the unknown test digits.

To implement the algorithm, we begin by importing the necessary data from Excel spreadsheets, including the training and test datasets. These datasets consist of handwritten digit images, represented as matrices, along with their corresponding labels.

In our code, we start by constructing matrix A for each class. We collectively generate ten matrices, labeled A1 to A10, where each matrix corresponds to a specific digit class. These matrices have 256 rows, representing 16x16 pixels, and a varying number of columns, corresponding to the samples in each class.

Next, we compute the SVD for each A_i matrix, resulting in the matrices U, S, and V. The S matrix contains the singular values, while the U matrix consists of the left singular vectors derived from the eigenvectors of the product of A_i and its transpose (AAT).

To classify an unknown digit, we have collectively defined a function named clf(kmin, kmax). This function takes the minimum and maximum values of k, representing the number of basis vectors to consider. By calculating the norm of the residual error ($||I - UU^T||$) between the unknown digit and the class matrices, we can determine the most suitable digit class based on the lowest residual error.

The function returns a DataFrame named predK, which contains the predicted digit for each different number of singular vectors considered.

We then evaluate the accuracy of the predictions for each value of k and store the results in a DataFrame named accuracyK. We utilize the scikit-learn accuracy_score function to compare the predicted labels with the true labels from the test dataset.

Finally, we collectively plot the accuracy for each different number of singular vectors considered and display the confusion matrix and classification report for the best value of k.

By analyzing the accuracy results and visualizing the performance metrics, we can assess the effectiveness of our algorithm in classifying handwritten digits. Our goal is to collectively determine the optimal number of basis vectors to achieve the highest classification accuracy.

```
#Import the data from each excel spreadsheet in xtrain,ytrain,xtest,ytest

ytest=pd.read_excel('.\Dataset\data.xlsx',sheet_name='dtest',header=None)
ytrain=pd.read_excel('.\Dataset\data.xlsx',sheet_name='dzip',header=None)
xtrain=pd.read_excel('.\Dataset\data.xlsx',sheet_name='azip',header=None)
xtest=pd.read_excel('.\Dataset\data.xlsx',sheet_name='testzip',header=None)

# Create matrix A which contains 10 matrices, each one of them corresponding to matrix Ai

where i=0,..,9 classes
#Each matrix Ai has 256 rows
#Matrix A1 contains images that are classified only as 1 in ytrain etc.

A=[]
for i in range(10):
    A.append(xtrain.loc[:,ytrain.iloc[0,:]==i])


#Calculate U,S,V by applying SVD in each one of the Ai
#Again each of U,S,V contain Ui,Si,Vi for each one of the classes
```

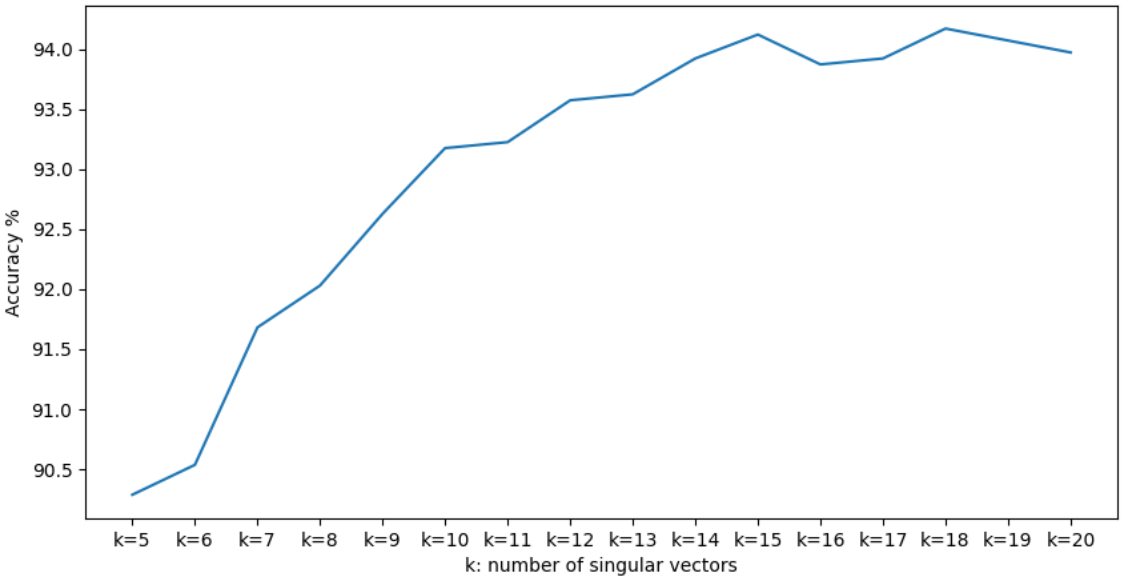
```
U=[]
S=[]
V=[]
for i in range(10):
    u,s,v=svd(A[i],full_matrices=False)
    U.append(u)
    S.append(s)
    V.append(v)

#Define function which takes the minimum and maximum value of k
#Calculate Identity matrix as I
#For each number of basis vectors calculate the norm of residual error (||I-UU^T||)
#Classify the digit as the one with the lowest residual
#predK contains the predictions for each different number of singular vectors

def clf(kmin,kmax):
    I=np.eye(len(xtest))
    predK=pd.DataFrame()
    for k in range(kmin,kmax):
        pred=[]
        for i in range(len(xtest.columns)):
            residuals=[]
            for j in range(10):
                u=U[j][:,:k]
                residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
            minres=np.argmin(residuals)
            pred.append(minres)
        pred=pd.DataFrame(pred)
        predK[str(k)]=pred
    return predK

#Run function clf

predK=clf(5,21)

#Calculate the accuracy of predictions for each different number of singular vectors

accuracyK=pd.DataFrame()
for k in range(5,21):
    accuracy=accuracy_score(ytest.iloc[0,:],predK[str(k)])
    accuracyK['k='+str(k)]=[accuracy]

print(accuracyK)


#Plot the accuracy for each different numbe of singular vectors

plt.figure(figsize=(10,5))
plt.plot(accuracyK.columns,accuracyK.iloc[0,:]*100)
plt.ylabel('Accuracy %')
plt.xlabel('k: number of singular vectors')


#Confusion matrix for the best k
#Columns show the predicted class
#Rows show the true class

print(confusion_matrix(ytest.iloc[0,:], predK.iloc[:,np.argmax(accuracyK)]))
```

```
#Classification report for the best k

print(classification_report(ytest.iloc[0,:], predK.iloc[:,np.argmax(accuracyK)]))
```

Below are represented the accuracy of predictions for each different number of singular vectors:

```
       k=5       k=6       k=7       k=8       k=9      k=10      k=11  \
0  0.90284  0.905331  0.916791  0.920279  0.926258  0.931739  0.932237

       k=12      k=13      k=14      k=15      k=16      k=17      k=18  \
0  0.935725  0.936223  0.939213  0.941206  0.938714  0.939213  0.941704

       k=19      k=20
0  0.940708  0.939711
```

Below is represented the accuracy for each different number of singular vectors:



Below are represented the Confusion matrix for the best k:

```
[[355   0   2   0   1   0   0   0   0   1]
 [  0 259   0   0   3   0   2   0   0   0]
 [  8   1 178   2   5   0   0   1   3   0]
 [  2   0   3 150   1   6   0   1   2   1]
 [  2   1   0   0 185   2   0   3   0   7]
 [  7   1   1   5   0 141   0   0   2   3]
 [  2   1   0   0   2   1 163   0   1   0]
 [  0   1   1   0   3   0   0 141   0   1]
 [  2   0   1   6   0   1   0   0 153   3]
 [  0   2   0   1   4   0   0   3   2 165]]
```

Below are represented the Classification report for the best k:

	precision	recall	f1-score	support
0	0.94	0.99	0.96	359
1	0.97	0.98	0.98	264
2	0.96	0.90	0.93	198
3	0.91	0.90	0.91	166
4	0.91	0.93	0.92	200
5	0.93	0.88	0.91	160
6	0.99	0.96	0.97	170
7	0.95	0.96	0.95	147
8	0.94	0.92	0.93	166
9	0.91	0.93	0.92	177
accuracy			0.94	2007
macro avg	0.94	0.94	0.94	2007
weighted avg	0.94	0.94	0.94	2007

Finally, the results indicate that employing the first 18 basis singular vectors resulted in a greater classification accuracy of 94%. This is further supported by the plot of accuracy for various k. According to the categorization report for the best k=18, the f1-score is 94%. The classes with the greatest f1-scores are digits 1 and 6, with f1-scores of 98% and 97%, respectively, while the classes with the lowest f1-scores are digits 3 and 5, with f1-scores of 91%.

3 Question 2

In this section, we investigate whether all the digits in our classification problem are equally easy or difficult to classify. Additionally, we explore some of the challenging cases where misclassifications occur, particularly focusing on poorly written digits.

Upon analyzing the confusion matrix and the classification report, it becomes apparent that certain digits pose more challenges in terms of accurate classification. Specifically, the digits 3 and 5 exhibit lower f1-scores, indicating a higher level of difficulty in correctly identifying them.

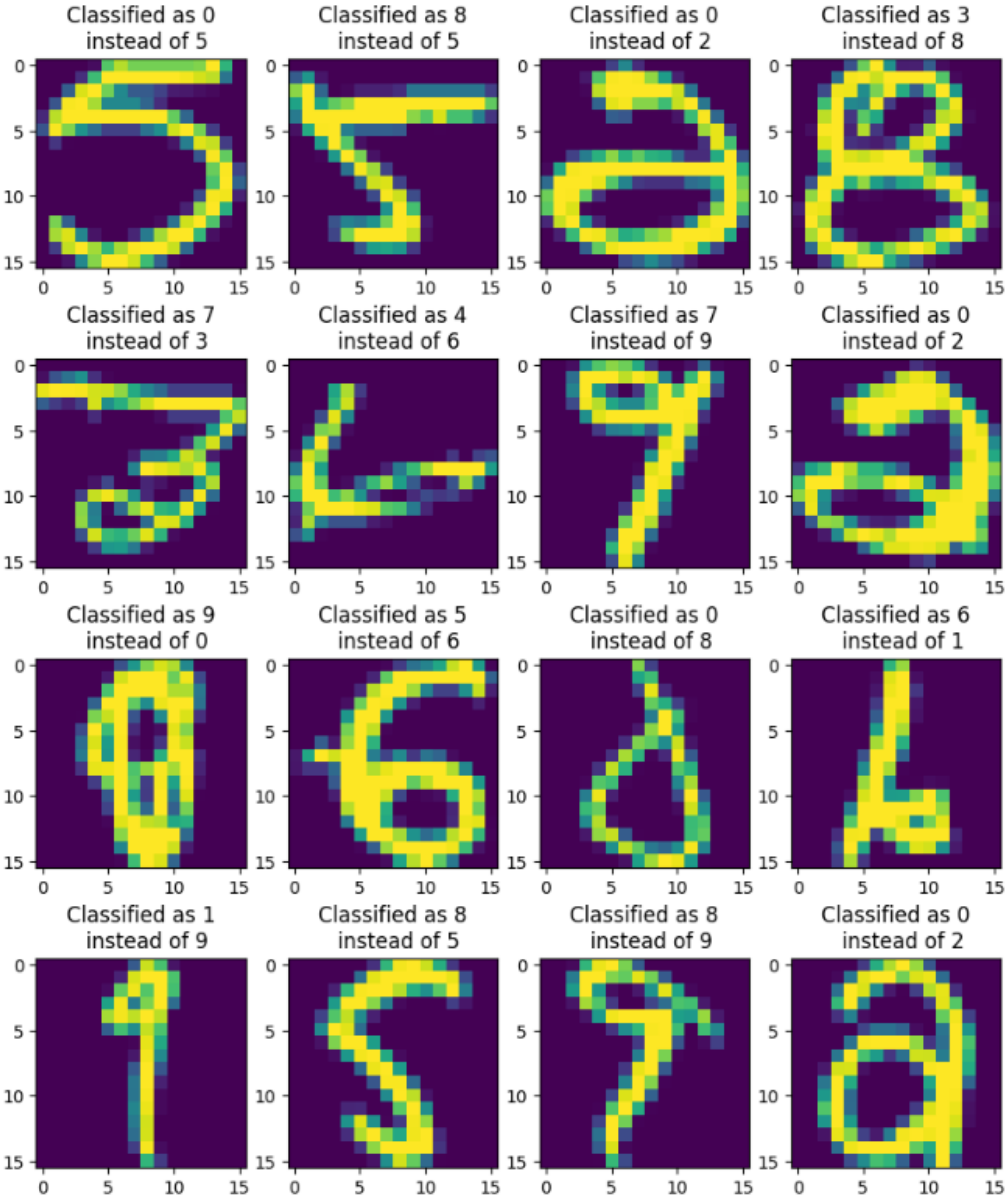
To gain further insights into these challenging cases, we examine the misclassified digits. By visualizing a selection of these misclassifications, we can observe the nature of the digit images that lead to incorrect predictions. Furthermore, we specifically analyze the digits 3 and 5 with the lowest f1-scores, aiming to identify common characteristics or irregularities in their handwritten representations.

Through this examination, we expect to uncover patterns or distinct features that contribute to the misclassification of certain digits. Understanding the difficulties and observing the poorly written digits will provide valuable insights into the limitations and potential areas of improvement for our handwritten digit classification algorithm.

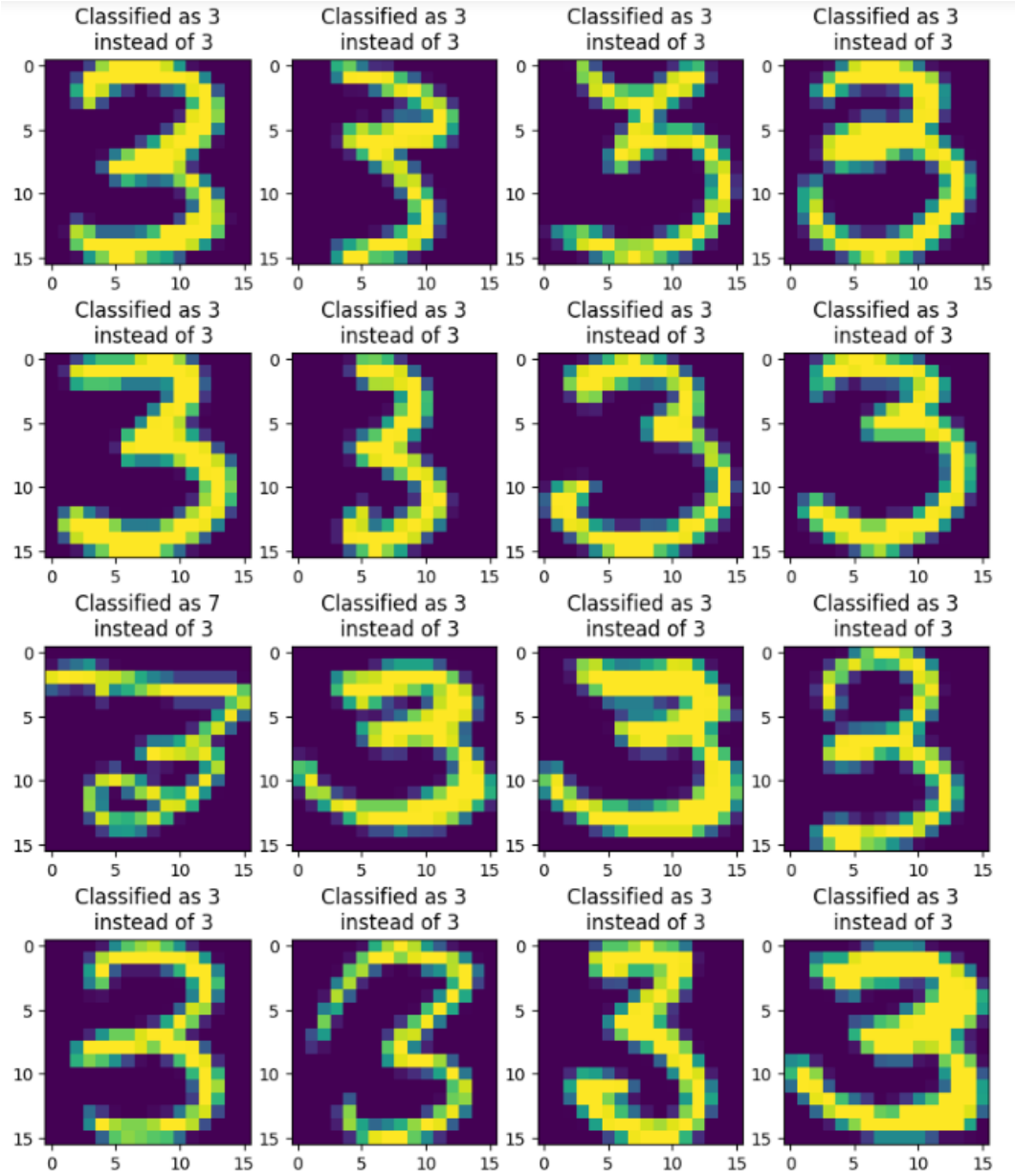
```
#Create the function ima2.n that displays the digits

def ima2(A):
    image=np.array(A)
    image=image.reshape(16,16)
    plt.imshow(image)

#Array misclass contains the indexes of wrongly classified digits

misclass=np.where(ytest.iloc[0,:]!=predK.iloc[:,13])

#Plot the first 16 digits that were wrongly classified

plt.figure(figsize=(10,12))
for i in range(16):
    plt.subplot(4,4,i+1)
    ima2(xtest.iloc[:,misclass[0][i]])
    plt.title('Classified as '+str(predK.iloc[misclass[0][i],13])\
    +'\n '+' instead of '+str(ytest.iloc[0,misclass[0][i]]))

#Array class3 contains the indexes of the three digit

class3=np.where(ytest.iloc[0,:]==3)

#Plot the first 16 digits that were the digit 3

plt.figure(figsize=(10,12))
for i in range(16):
    plt.subplot(4,4,i+1)
    ima2(xtest.iloc[:,class3[0][i]])
    plt.title('Classified as '+str(predK.iloc[class3[0][i],13])\
    +'\n '+' instead of '+str(ytest.iloc[0,class3[0][i]]))

#Array class5 contains the indexes of the five digit

class5=np.where(ytest.iloc[0,:]==5)
```

```
#Plot the first 16 digits that were the digit 5

plt.figure(figsize=(10,12))
for i in range(16):
    plt.subplot(4,4,i+1)
    ima2(xtest.iloc[:,class5[0][i]])
    plt.title('Classified as '+str(predK.iloc[class5[0][i],13])\
    +'\n '+' instead of '+str(ytest.iloc[0,class5[0][i]]))
```

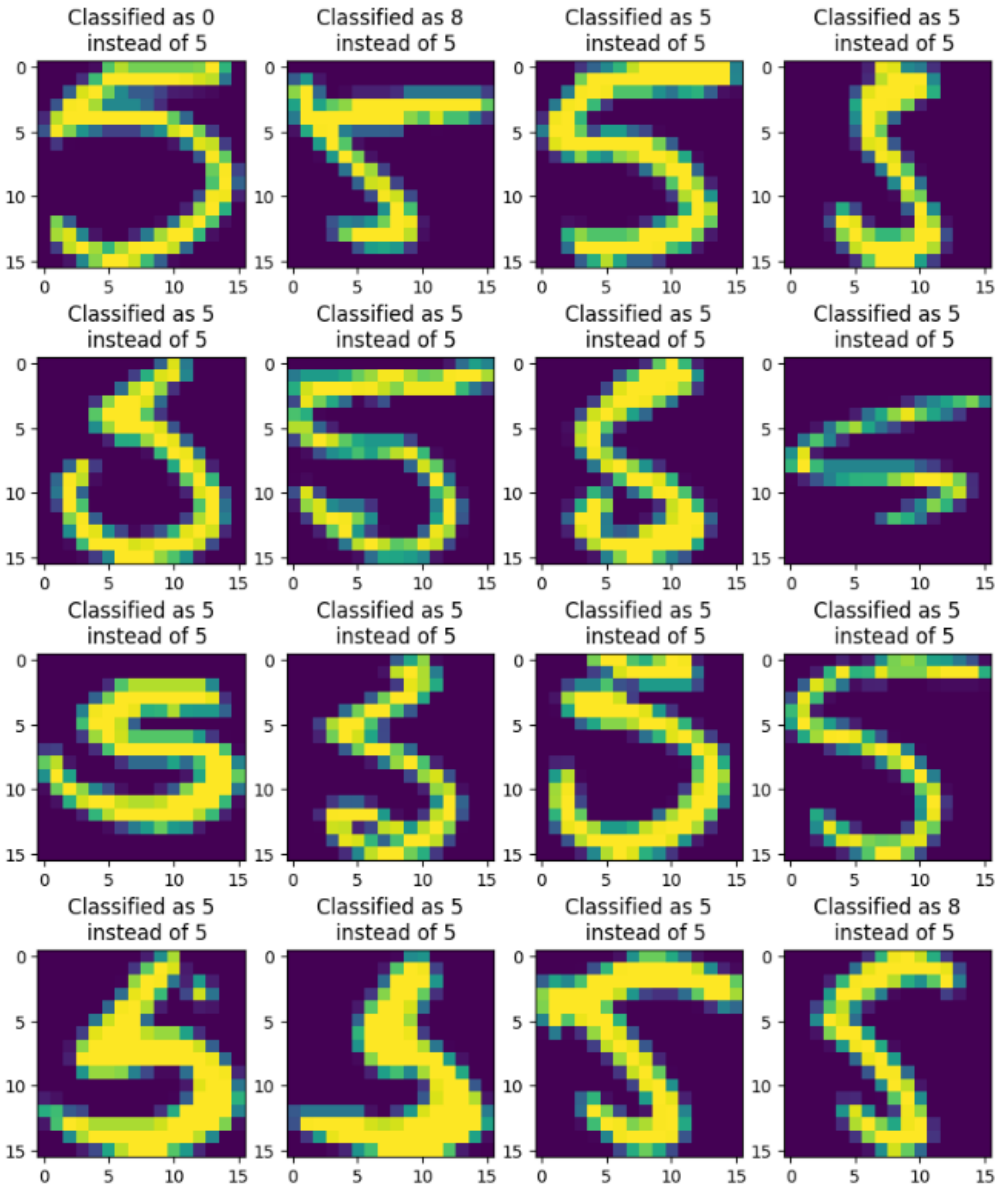
The below plot represents the first 16 digits that were wrongly classified:



The below plot represents the first 16 digits that were the digit 3:



The below plot represents the first 16 digits that were the digit 5:



Indeed, we observe that the digits that were wrongly classified were poorly written.

4 Question 3

In this section, we aim to explore the possibility of using different numbers of basis vectors for different classes in our digit classification problem. By examining the singular values of the different classes, we can determine if it is motivated to employ varying numbers of basis vectors.

Based on our previous knowledge, we have observed that certain classes have significantly larger first singular values compared to the rest. This observation suggests that these classes may retain the majority of the relevant information with only the first few singular vectors. Consequently, using fewer basis vectors for these classes might be sufficient and potentially improve efficiency without sacrificing classification accuracy. Specifically, we hypothesize that classes with higher classification outcomes, such as digits zero and one, may benefit from using fewer singular vectors due to their larger initial singular values.

To investigate this hypothesis, we perform a few experiments to assess the impact of using fewer basis vectors in one or two specific classes. We analyze the norm of the residual error ($||I - UU^T||$) for different numbers of basis vectors and classify the digits based on the lowest residual. By comparing the accuracy of predictions for each different number of singular vectors, we can evaluate if it pays off to use fewer basis vectors in specific classes.

Now let's delve into the code that implements these experiments. The code consists of three main parts, each focusing on a different scenario: testing digit zero, testing digit one, and testing both digits zero and one together.

In the first part, we examine the hypothesis with digit zero. For each number of basis vectors (k) ranging from 1 to 18, we calculate the norm of the residual error for digit zero while keeping the number of singular vectors for all other digits fixed at 18. The digit is then classified based on the lowest residual, and the predictions for each different number of singular vectors are stored in the DataFrame predKzero. Additionally, we calculate the accuracy of predictions for each k and store the results in the DataFrame accuracyKzero.

The second part follows a similar procedure but focuses on testing digit one. We calculate the norm of the residual error for digit one while keeping the number of singular vectors for all other digits fixed at 18. The predictions for each different number of singular vectors are stored in the DataFrame predKone, and the accuracy of predictions for each k is stored in accuracyKone.

In the third part, we test both digits zero and one together. We calculate the norm of the residual error for both digits while keeping the number of singular vectors for all other digits fixed at 18. The predictions for each different number of singular vectors are stored in the DataFrame predKone_zero_one, and the accuracy of predictions for each k is stored in accuracyKone_zero. This allows us to evaluate the performance of using different numbers of basis vectors for both digits combined.

By analyzing the accuracy results and identifying the optimal k values, we can determine if it is indeed motivated to use different numbers of basis vectors for different classes.

```
# Check the first singular values

for i in range(10):
    print('Singular values for class'+str(i)+'\n',S[i][:10])

#For each number of basis vectors calculate the norm of residual error (||I-UU^T||) for

digit ZERO
#Classify the digit as the one with the lowest residual
#predKzero contains the predictions for each different number of singular vectors for

digit ZERO, while all the other digits have k=18

class0=np.where(ytest.iloc[0,:]==0)
```

```
I=np.eye(len(class0))
predKzero=pd.DataFrame()
for k in range(1,19):
    pred=[]
    for i in range(len(xtest.columns)):
        residuals=[]
        for j in range(10):
            if j==0:
                u=U[j][:,:k]
                residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
            else:
                u=U[j][:,:18]
                residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
        minres=np.argmin(residuals)
        pred.append(minres)
    pred=pd.DataFrame(pred)
    predKzero[str(k)]=pred

#Calculate the accuracy of predictions for each different number of singular vectors

for digit zero while all the other digits have k=18

accuracyKzero=pd.DataFrame()
for k in range(1,19):
    accuracy=accuracy_score(ytest.iloc[0,:],predKzero[str(k)])
    accuracyKzero['k='+str(k)]=[accuracy]

print(accuracyKzero)

print('\n The optimal k for digits zero is',np.argmax(accuracyKzero)+1)


#For each number of basis vectors calculate the norm of residual

error (||I-UU^T||) for digit ONE
#Classify the digit as the one with the lowest residual
#predKone contains the predictions for each different number of singular vectors for digit ONE,

while all the other digits have k=18


class1=np.where(ytest.iloc[0,:]==1)
I=np.eye(len(class1))
predKone=pd.DataFrame()
for k in range(1,19):
    pred=[]
    for i in range(len(xtest.columns)):
        residuals=[]
        for j in range(10):
            if j==1:
                u=U[j][:,:k]
                residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
            else:
                u=U[j][:,:18]
                residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
        minres=np.argmin(residuals)
        pred.append(minres)
    pred=pd.DataFrame(pred)
```

```
    predKone[str(k)]=pred


#Calculate the accuracy of predictions for each different number of singular vectors

for digit one while all the other digits have k=18

accuracyKone=pd.DataFrame()
for k in range(1,19):
    accuracy=accuracy_score(ytest.iloc[0,:],predKone[str(k)])
    accuracyKone['k='+str(k)]=[accuracy]

print(accuracyKone)

print('\n The optimal k for digit one is',np.argmax(accuracyKone)+1)

#For each number of basis vectors calculate the norm of residual error

(||I-UU^T||) for digit ONE and ZERO
#Classify the digit as the one with the lowest residual
#predKone_zero_one contains the predictions for each different number of singular vectors

for digit ONE and ZERO, while all the other digits have k=18

class01=np.where((ytest.iloc[0,:]==1) & (ytest.iloc[0, :] == 0))


I=np.eye(len(class01))

predKone_zero_one=pd.DataFrame()
for k in range(1,19):
    pred=[]
    for i in range(len(xtest.columns)):
        residuals=[]
        for j in range(10):
            if j==1:
                u=U[j][:,:k]
                residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
            elif j==0:
                u=U[j][:,:k]
                residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
            else:
                u=U[j][:,:18]
                residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
        minres=np.argmin(residuals)
        pred.append(minres)
    pred=pd.DataFrame(pred)
    predKone_zero_one[str(k)]=pred


#Calculate the accuracy of predictions for each different number of singular vectors

for digit one and zero while all the other digits have k=18

accuracyKone_zero=pd.DataFrame()
for k in range(1,19):
    accuracy=accuracy_score(ytest.iloc[0,:],predKone_zero_one[str(k)])
    accuracyKone_zero['k='+str(k)]=[accuracy]
```

```
print(accuracyKone_zero)

print('\n The optimal k for digits zero and one is',np.argmax(accuracyKone_zero)+1)
```

The first singular values are:

```
Singular values for class0
 [184.44749844  97.59388422  62.05061337  54.14931248  41.06020067
  40.36847138  36.28294233  30.02308652  29.00107349  24.94954536]
Singular values for class1
 [234.04151731  41.9309063   24.84924265  16.29480667  13.61095501
  12.52178131  11.22883717  10.66849318   8.4133817    8.32127045]
Singular values for class2
 [138.2836996   57.39739384  46.39569635  40.96003126  37.50530652
  33.53160003  32.237997    28.40611951  27.12012507  26.73498766]
Singular values for class3
 [126.63823606  39.77367291  33.32245701  29.87372361  27.9646299
  24.76781289  24.39030584  21.460863    19.05978766  18.59508072]
Singular values for class4
 [123.02706507  41.62602412  34.93431159  32.29075793  26.81182829
  24.5655462   21.86754262  20.44929199  20.08172813  18.45554627]
Singular values for class5
 [94.28412631 37.85299086 35.77494073 28.02067178 24.0466634  22.43784723
 20.63123172 20.36365239 17.06151968 16.81156101]
Singular values for class6
 [141.95961664  52.1946615   37.96584617  32.01451394  26.28505545
  25.13724003  21.2005021   20.93588569  19.34289072  18.88883469]
Singular values for class7
 [160.19698391  46.71035896  36.52512533  34.68763686  31.4956071
  22.46554732  20.34818397  19.1925414   17.58541538  16.78508287]
Singular values for class8
 [133.06780611  41.64353682  34.45996324  30.49563174  28.71666418
  25.47900665  22.6150183   21.89173764  20.91189761  19.47011758]
Singular values for class9
 [141.22390944  45.72512732  30.05843737  29.45708405  22.93443324
  20.50031445  18.80548273  16.97604746  15.2117038   14.80121696]
```

The optimal k for digits zero is:

```
        k=1       k=2       k=3       k=4       k=5       k=6       k=7  \
0  0.221226  0.219233  0.201295  0.182362  0.188341  0.197309  0.178376

        k=8       k=9      k=10      k=11      k=12      k=13      k=14  \
0  0.17439  0.169905  0.168909  0.169407  0.164923  0.164923  0.164923

       k=15      k=16      k=17      k=18
0  0.164425  0.164425  0.160438  0.161435

 The optimal k for digits zero is 1
```

The optimal k for digit one is:

```
        k=1       k=2       k=3       k=4      k=5       k=6       k=7  \
0  0.20578  0.206776  0.204285  0.201295  0.20279  0.204285  0.181863

        k=8      k=9      k=10      k=11      k=12      k=13      k=14  \
0  0.164923  0.15994  0.157947  0.167414  0.165919  0.161435  0.160937

       k=15      k=16      k=17      k=18
0  0.159442  0.156951  0.163926  0.161435

 The optimal k for digit one is 2
```

The optimal k for digits zero and one is:

```
        k=1       k=2       k=3       k=4       k=5       k=6      k=7  \
0  0.266567  0.263577  0.240658  0.219731  0.226208  0.237668  0.20279

        k=8       k=9      k=10      k=11      k=12      k=13      k=14  \
0  0.178376  0.167414  0.165421  0.175386  0.170404  0.165919  0.165421

       k=15     k=16      k=17      k=18
0  0.164923  0.16293  0.163428  0.161435

 The optimal k for digits zero and one is 1
```

5 Optional Task: Two-Stage Algorithm with SVD

In this optional task, we wanted to explore the implementation of a two-stage algorithm to save computational operations in the test phase. Our goal was to investigate whether this modified approach could achieve comparable results while potentially reducing the need for the second stage of the algorithm.

We decided to start by creating a first stage in the algorithm where we compared the unknown digit only to the first singular vector in each class. We calculated the residual error for each class and checked if one class exhibited a significantly smaller residual compared to the others. If such a difference was found, we classified the digit as belonging to that class. This allowed us to quickly and simply classify the digit without further computations.

However, if the residuals in the first stage did not show a significant difference among classes, we proceeded to the second stage, which followed the algorithm implemented in previous sections. In this stage, we compared the unknown digit with all training digits and classified it as the closest match.

Our main objective with this task was to evaluate the performance of the two-stage algorithm compared to using a fixed number of singular vectors for all classes. Additionally, we were interested in determining how frequently the second stage was necessary.

Now let's examine the code that implements this optional task. We started by initializing variables and setting the tolerance value for the significance of residuals (tol). We then proceeded with the two-stage algorithm.

For each test digit, we calculated the residuals for the first stage by considering only the first singular vector from each class. If the difference between the two smallest residuals was below the tolerance (tol), indicating no significant difference, we proceeded to the second stage. In the second stage, we recalculated the residuals using 18 singular vectors from each class. The minimum residual determined the predicted class for the test digit. We stored the predictions for each test digit in the DataFrame predKoptional.

Finally, we calculated the accuracy of the predictions and printed the result. We also counted the number of times the second stage was necessary and displayed that count. We found that the results for this modified algorithm were unsatisfactory compared to using other numbers of singular vectors. We achieved an accuracy of 63%, and the second stage was required six times. These findings suggested that while the two-stage algorithm may save computational operations in certain cases, it did not perform as well as using a fixed number of singular vectors for all classes in terms of classification accuracy.

```
predKoptional=pd.DataFrame()
pred=[]

#Tolerance of the significance of residuals
tol=0.005

#Number of second stages
count=0

#Two-stage algorithm
for i in range(len(xtest.columns)):
    residuals=[]
    for j in range(10):
        u=U[j][:,:1]
        #Use one singular vector
        residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
    if sorted(residuals)[1]-sorted(residuals)[0]<tol:
        residuals=[]
```

```
        count=count+1
        for n in range(10):
            u=U[j][:,:18]
            #Use 18 singular vectors in second stage if necessary
            residuals.append(norm(np.dot(I-np.dot(u,u.T),xtest.iloc[:,i]),2))
    minres=np.argmin(residuals)
    pred.append(minres)
pred=pd.DataFrame(pred)
predKoptional=pred

#Calculate the accuracy of predictions

accuracy=accuracy_score(ytest.iloc[0,:],predKoptional)

print('The accuracy is ',accuracy*100, '%')
print('The number that the second stage is necessary is',count)
```