Uppsala University

Scientific Computing for Data Analysis

1TD352 62042

Miniproject 2

Authors:
Alexander Cremer
— Ask Olaussen —
Emil Kinman Maly

May 14, 2025

1 Introduction

In this mini project we try to gain a deeper understanding into how matrices in python can be used to solve different tasks. In this case to classify handwritten numbers, after training the model on given training sets. The handwritten numbers will be represented by matrices, which the program then will use to evaluate and classify. The SVD method in particular will be important here and we will look into what the three components of the SVD tells us about the classifications of the numbers. The amount of test samples are pretty large, so discovering the accuracy of the method depending on how many test samples we use can be of interest to reduce unnecessary computations.

2 Approach/method

This report describes the construction of an algorithm that, based on a training set of handwritten digits, is capable of classifying a set of handwritten unknown digits between 0 and 9. In the following section, we explain the basic principles of the mathematical and computational theories that make this possible.

At its core, the problem is about breaking down the training images into small components. By using singular value decomposition (SVD), we can isolate the main patterns in variation, thus allowing us to create templates for each of the digits. To then classify a new unknown image, we project it onto each of these templates and measure the residual, allowing us to identify the most likely digit. For the sake of clarity, the whole process has been divided into two principal components: Pattern recognition and Classification. These are presented in the following setions.

2.1 Pattern recognition - "training" the program to recognize digits

The data we allow the computer to process (in this case, training images of hand-drawn numbers) can be broken down into small components by letting each image be represented by an m x n matrix, where each (i, j)-th entry corresponds to the brightness of the pixel at location (i, j). In the case of this project, the m x n matrix of size 28 x 28 is reshaped to a vector of size 784 x 1, thus letting each image of a digit be represented by a vector. We now do the same with a large number of different training images of the same digit, and place all these vectors together. This leaves us with a matrix A, with size 784 x n, where n is the number of training images we choose to use. The columns that make up the matrix A correspond to different hand-drawn variants of the same digit, which means that we get an almost linear dependence in A. As we know from linear algebra, this means that the data in A effectively occupies a much lower-dimensional subspace than the full 784-dimensional image space. If we now have a way of identifying the most significant variations in the way the studied digit is drawn, we should be able to project each image onto these directions and thereby find the most significant patterns. This is exactly what SVD does. By decomposing A as $A = U\Sigma V^T$, we can then let a number of the most significant orthogonal vectors U1, U2, ..., Uk, called singular images, make out the foundation of what we define as the subspace of the digit. We can then repeat the same procedure for all 10 digits, thus creating a model that encapsulates the fundamental features of every digit.

2.2 Classification – using residuals to identify the most likely digit

The next part is about letting the program process new unknown data (in this case, a set of test digits unknown to the computer) and decide which of the previously constructed models have the best resemblance using residuals, meaning the actual data - model predictions. In the same way as before, we can express each image of a handwritten digit as a vector of size 784 x 1.

The least squares solution gives us:

$$\boldsymbol{x}^* = \min_{\mathbf{x}} \|U_k\mathbf{x} - \boldsymbol{\delta}\|_2^2$$

We now take the gradient and set to zero to get:

$$\nabla_x^2 \|U_k\boldsymbol{x} - \boldsymbol{\delta}\|_2^2 = U_k^T U_k \boldsymbol{x} - U_k^T \boldsymbol{\delta}$$

$$\Rightarrow U_k^T U_k \boldsymbol{x} = U_k^T \boldsymbol{\delta}$$

Since U_k has orthonormal columns:

$$[U_k^T U_k = I]$$

$$\Rightarrow \boldsymbol{I}\boldsymbol{x} = U_k^T \boldsymbol{\delta}$$

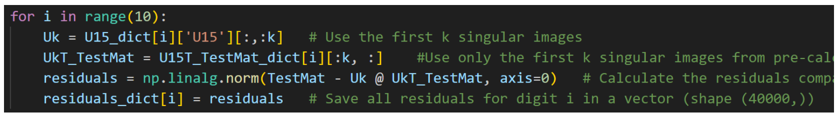
$$\Leftrightarrow \boldsymbol{x} = U_k^T \boldsymbol{\delta}$$

Residual = actual data - model prediction

$$\text{residual} = \|r\|_2 = \|\boldsymbol{\delta} - U_k\boldsymbol{x}^*\|_2$$

$$\Leftrightarrow \|r\|_2 = (I - U_k U_k^T)\boldsymbol{\delta}\|_2$$

This method can be used in the code as well, but calculating residuals for one digit at a time can take a very long time, especially when working with tens of thousands of digits, as in this case. A much more efficient approach is to perform the residual calculation by matrix multiplication with a matrix containing all 40000 test digits. In that way, we project all 40000 test images at the same time onto one of the digit bases. This process is then repeated for all ten digit bases and their resulting residuals are stored in corresponding vectors of shape (40000,). The implementation in the code is shown in figure 1:



```python
for i in range(10):
    Uk = U15_dict[i]['U15'][:,:k]   # Use the first k singular images
    UkT_TestMat = U15T_TestMat_dict[i][:k, :]    #Use only the first k singular images from pre-cal
    residuals = np.linalg.norm(TestMat - Uk @ UkT_TestMat, axis=0)   # Calculate the residuals comp
    residuals_dict[i] = residuals   # Save all residuals for digit i in a vector (shape (40000,))
```

Figure 1: Implementation of test image matrix projection onto the digit bases

The ten residual vectors are then stacked together row wise, leaving us with a matrix R of shape 10 x 40000. For each column in R (containing the 10 residual values for one test image), we save the row number (representing a digit) with the smallest residual value. This is done using the numpy function np.argmin(R, axis=0).

3 Results

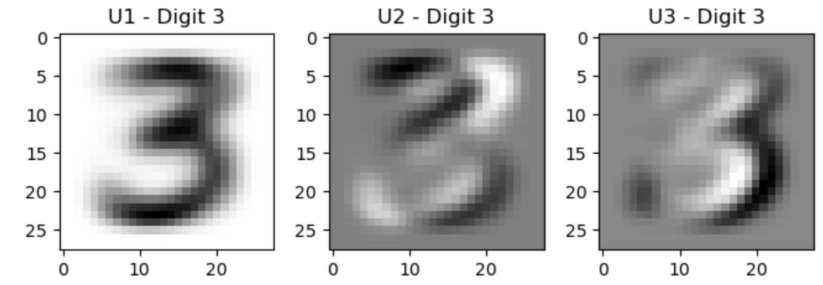


Figure 2: First three singular images of the digit 3

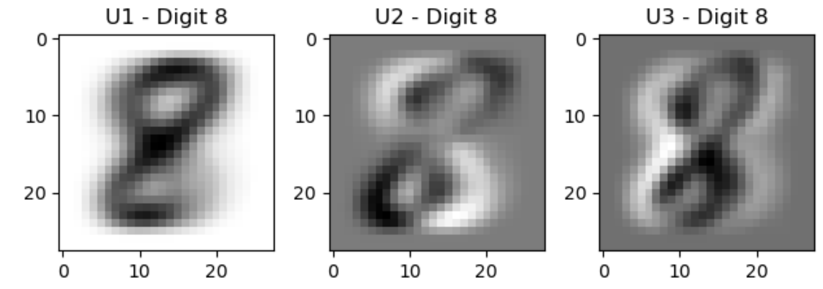


Figure 3: First three singular images of the digit 8

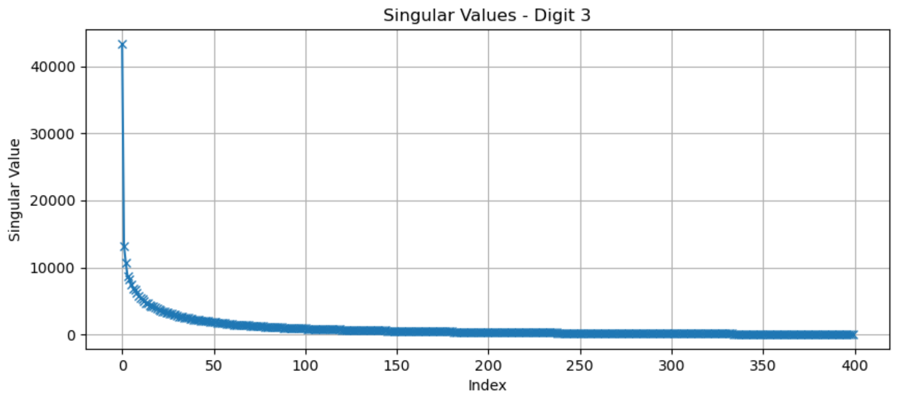


Figure 4: Singular values of digit 3

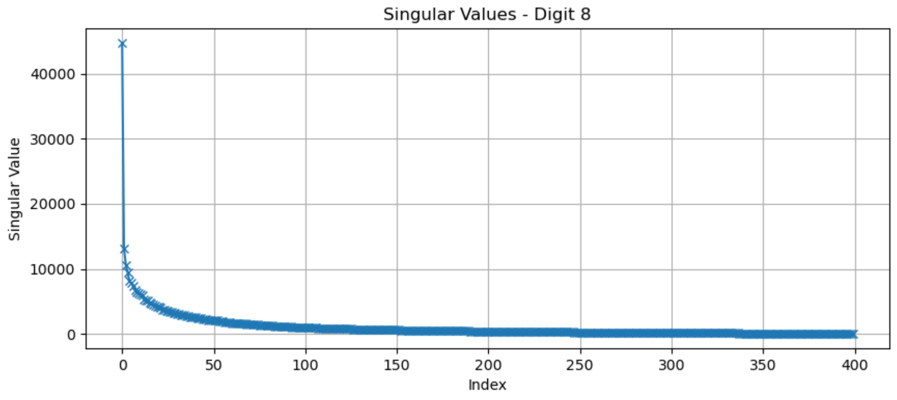


Figure 5: Singular values of digit 8

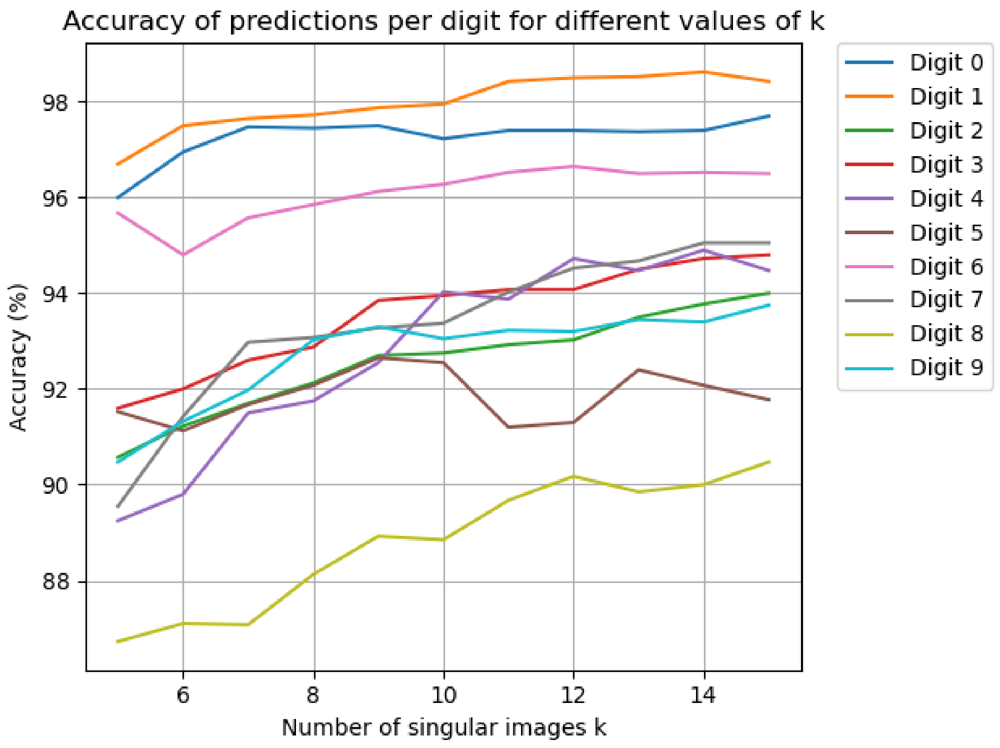


Figure 6: Prediction accuracy for each digit separately with different values of k, based on 400 training images.

k	**Total Prediction Accuracy (%)**
5	91.8075
6	92.3250
7	93.0200
8	93.4075
9	93.8750
10	94.0000
11	94.1350
12	94.3575
13	94.5250
14	94.6475
15	94.6950

Table 1: Total prediction accuracy for different values of k, where k is the number of singular images, based on 400 training images.

4 Discussion

Figure 3 and 4 in the results section shows the 400 first singular values for the digits three and eight. The singular values corresponds to how normal these features are for the digits. As we see both graphs have a sharp decline in the beginning. This is a wanted result because that means we could approximate the digit quite well with few singular images. If the decline would have been more gradual we would have needed to take many more singular images into account when classifying the numbers to get a high succesrate.

Looking at the singular images of the same digits in figure 1 and 2, we can see what our three best approximations would look like. These images show the most common features of the digits. One example is the point in the middle of the three, which is very characteristic. At the edges however we can see that it differs more between how the digit is written. Some of the same tendencies can be seen for the digit eight. The shade of the background of the images how strong that particular image is. This is taken from the singular values shown in figure 3 and 4. From these graphs we can see that there is a huge difference between the first and the second. That means that most digits could get recognized from this alone, but more singular images would have a higher accuracy.

Figure 6 shows the accuracy of the model for the different digits with k-values from 5 to 15. The differences between the digits are worth noting.

For example the algorithm catches the digit one much easier than eight. This can in part be seen by the singular images for eight, which are more blurry and not as define as the ones for the digit three. Indicating that there may be a greater variety in how eight is written than the other numbers, it could also be that the training set for eight is of lesser quality than the others. Given that the training sets are quite large, it is reasonable to think that they not vary to much in quality and that the worse accuracy is down to that the digit itself is harder to read because there are more fluctuations in the way it is written. That could in part be because eight is a soft digit without any cute angels which would be easier to recognise.

In table 1 we find the total accuracies of all the digits combined with k-values between five and 15. Already with k=5 we have a method that guesses right more than nine times out of ten. Which is quite good. If we increase k a bit we see that it is possible to gain valuable accuracy still. However the trend seem to flatten and by the time k reaches about ten only small improvements can be made. So to use a k-value between eight and twelve can be useful for this method in order to make the algorithm work fast but still maintain high accuracy. We want to keep k as low as possible while having good accuracy. There are additional risks to increasing the k-value. The singular images closer to 400 are very blurry, which could potentially lead to classifying numbers in the wrong categories. Depending on what the method is used for getting a false digit could have much graver consequences than missing out on a digit.

5 Appendix

6 Code

```python
import numpy as np
import matplotlib.pyplot as plt


TrainMat = np.load('TrainDigits.npy')
TrainLab = np.load('TrainLabels.npy')
TestLab = np.load('TestLabels.npy')
TestMat = np.load('TestDigits.npy')


```

```
##### Task 2 #####
#Plot singular values and images for digits 3 and 8 using 400
     training images

n = 400   #The ammount of training images used

#Extract first 400 images for each digit 0 - 9
#Compute SVD and save the information in a dictionary
digit_data = {}   #Dictionary to save digit data
for i in range(10):
    index = (TrainLab == i); #Find train digits of type i
    Ai_all = TrainMat[:,index[0]] #All train digits of type i
    Ai = Ai_all[:,0:n] #The first n train digits of type i

    #SVD for digits 0 - 9:
    Ui, Si, VTi = np.linalg.svd(Ai)

    #Save in dictionary
    digit_data[i] = {
        'A': Ai,
        'U': Ui,
        'S': Si,
        'VT': VTi
    }


#Plot singular values (S) for digit d
def singular_values(d):
    plt.figure(figsize=(10, 4))
    plt.plot(digit_data[d]['S'], marker='x')
    plt.title(f'Singular Values - Digit {d}')
    plt.xlabel("Index")
    plt.ylabel("Singular Value")
    plt.grid(True)
    plt.show()

#Plot singular images (U1, U2, U3) for digit d
def singular_images(d):
    plt.figure()
    plt.suptitle(f'Singular images U1, U2 & U3 for digit {d}'
    )
    for i in range(3):
        plt.subplot(1, 3, i+1)
        plt.title(f'U{i+1} - Digit {d}')
```

```python
        u_i = digit_data[d]['U'][:,i] #The first digit in the training set
        u = np.reshape(u_i, (28, 28)).T #Reshaping a vector to a matrix
        plt.imshow(u, cmap ='gray') #Plot of the digit
    plt.tight_layout()
    plt.show()


#Run the plots
singular_values(3)
singular_images(3)
singular_values(8)
singular_images(8)



##### Task 3 #####

#Store the first 15 singular imgages of each digit in a dictionary
U15_dict = {}
for i in range(10):
    U15_dict[i] = {'U15': digit_data[i]['U'][:,:15]} #The first 15 singular images of each digit (0 - 9)


#np.shape(TestMat) = (784, 40000)

#Calculate the accuracy of predicitons for values k = 5, 6, ... , 15
def accuracies_with_k(k_start, k_stop):

    all_accuracies = []
    U15T_TestMat_dict = {}      #Dictionary for pre-calculated U15.T @ TestMat

    #Pre-calculations of the part of the matrix multiplication that is not dependent on k. Saves us computational power.
    for i in range(10):
        U15T_TestMat_dict[i] = U15_dict[i]['U15'].T @ TestMat #Shape (15, 40000)

    for k in range(k_start,k_stop):
```

```python
    residuals_dict = {}

    #Loop that projects all 40000 test images at the same time on the base of every class (digit 0-9)
    for i in range(10):
        Uk = U15_dict[i]['U15'][:,:k]   #Use the first k singular images
        UkT_TestMat = U15T_TestMat_dict[i][:k, :]    #Use only the first k singular images from pre-calculated dictionary. Shape (k, 40000)
        residuals = np.linalg.norm(TestMat - Uk @ UkT_TestMat, axis=0)   #Calculate the residuals compared to the matrix of the test digit
        residuals_dict[i] = residuals   #Save all residuals for digit i in a vector (shape (40000,))

    #Stack all 10 vectors row wise (R[0,:] contains residuals_dict[0], R[1,:] contains residuals_dict[1] and so on.)
    R = np.vstack([residuals_dict[i] for i in range(10)])  #shape (10, 40000)

    predictions = np.argmin(R, axis=0)    #For each column (test image) in R, we take the row (representing a digit) with the smallest residual


    #Create an array with accuracies for every digit with k singular images
    accuracies = np.array([np.mean(predictions[TestLab[0] == d] == d) * 100 for d in range(10)])
    all_accuracies.append(accuracies)

    #Explenation of the code above
    #TestLab[0] == d  -->   creates a template for where the digit d is placed in the test set. For example  [False, False, True, False] for digit 2 in [3, 7, 2, 1]
    #predictions[TestLab[0] == d]   -->   gives us the values in the predictions array at indices where the digit should be (indices where d is located in test set)
    #predictions[TestLab[0] == d] == d   --> checks if the values match. Gives us True/False depending on if the prediction was correct/wrong
    #np.mean works here because True = 1, False = 0
    # * 100 to get in %

    #Calculate total accuracy (all digits combined) (optional)
    """
        correct = np.sum(predictions == TestLab[0])   # The ammount of predictions that equal their respective test labels (correct predictions)
        total = len(TestLab[0])   # Total ammount of images tested
        accuracy = correct / total * 100  # Accuracy in %
        print(f' With k = {k}, we get a prediction accuracy of {accuracy}%')
    """

    return np.array(all_accuracies)



def plot_accuracies(k_start, k_stop):
    all_accuracies = accuracies_with_k(k_start, k_stop)
    for d in range(10):
        plt.plot(range(k_start, k_stop), all_accuracies[:, d], label=f'Digit {d}')
    plt.xlabel('Number of singular images k')
    plt.ylabel('Accuracy (%)')
    plt.title('Accuracy of predictions per digit for different values of k')
    plt.grid(True)
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
    plt.tight_layout()
    plt.show()

plot_accuracies(5, 16)
```

Listing 1: Code for mini-project 2