# Classificiation of Handwritten Digits by SVD

Georgiy Kiselev
Student id: 919184826

December 9, 2023

## 1 Introduction

This paper seeks to examine the application of Sparse Singular Value Decomposition towards the classification of handwritten digits. By leveraging the inherent sparsity of handwritten digit datasets—wherein only a few features significantly contribute to distinguishing between digits—Sparse SVD holds the potential to capture these essential discriminative features efficiently. This, in turn, leads to a classification model that is not only computationally efficient but also yields precise and reliable results. The process is interesting because it will help develop an intuitive understanding of the SVD method for the reader, as well as provide a relatively computationally inexpensive method for the classification problem while providing significantly accurate results. Primarily, it will demonstrate a real-world use case for this mathematical process in order to develop an appreciation for its applications and benefits. The paper is structured thus: an introduction to the data, followed by the methods used for classification, an examination of the results obtaining from applying said methods, and a conclusion formally summarizing the paper. A code appendix is included for reproducibility of results. Additionally, the Python libraries scipy, numpy, matplotlib.pyplot and sklearn are required for the implementation.

## 2 Data Set

The data set used in this paper was created as part of a project by the United States Postal Service, where envelopes were scanned for handwritten digits 0-9, which were added to the data as $16x16$ black and white images. The data set contains 9,298 of these images, with a roughly equal representation of each digit class. Half (4,649) of these images will be used for training, and the other half for testing. That is, the singular values used in classification will be extracted from the former, and the method will be tested for accuracy on the latter. To be usable in code, each image is represented as a length 256 vector of normalized values ranging from -1 to 1 based on the color value of each pixel in the image. The dataset also includes training and testing labels, which are one-hot encoded vectors of length 10 corresponding to each image. These label vectors have a -1 at every index except the index representing the number shown in the image, which has a 1. For the context of the code, the data will be used in an (X,y) representation, with X being the digit images and y being their corresponding labels. Thus, the code appendix references train_patterns, test_patterns, train_labels and test_labels as X_train_img, X_test_img, y_train and y_test respectively.

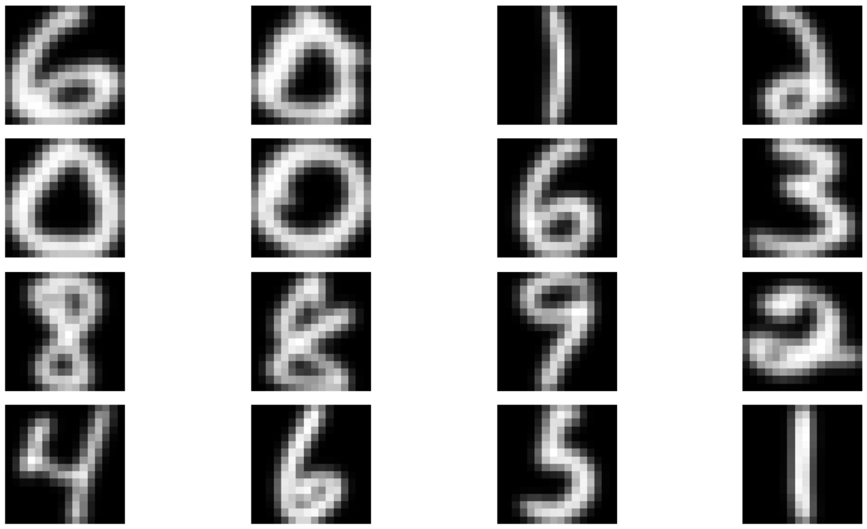


Figure 1: Handwritten digits from the dataset

The training method examined in section 3 will utilize the averages of each digit 0-9. The code appendix creates a matrix called train_aves of size 256x10 containing mean digit images for each class. The plot below shows these averaged figures.

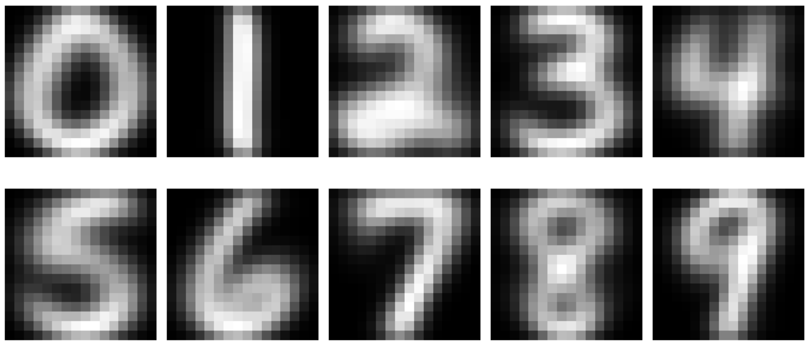


Figure 2: Mean of each digit 0-9

While the averaged figures are more 'blurry' than their individual components, the primary features of each digit are still clearly distinguishable.

## 3 Methods

### 3.1 Simple Classification Algorithm

Firstly, we examine a simple method of classification based on the Euclidean Distance formula below, where p and q are points in Euclidean space. Equivalently, this metric can be computed via the 2-norm of q-p:

$$d(p,q) = \sqrt(\sum_{i=1}^{n}((q_i-p_i)^2)) \tag{1}$$

We compute the distance between the pixel values of each test image and the averaged of each digit class. The index of the minimum distance between the test image and an averaged value will be selected as the prediction label. Essentially, we seek to find which digit the test image is most similar to based on each of its 256 pixel values. Though this is a basic approach, it should still demonstrate significant results.

We first form a matrix of size 10x4649 (test_classif in the code appendix), the entries of which consists of the Euclidean distance between each image in X_test_img and each mean digit image from X_train_img. As described above, the index of the minimum distance for each image will be chosen as the predicted label for it. The results of this procedure are given by the vector test_classif_res and will be examined for accuracy in a later section. However, this method only seeks to provide a performance benchmark that seeks to demonstrate the superiority of the next method.

## 3.2 Sparse SVD Based Classification

Because we continue to strive for increased performance, we suggest the utilization of sparse SVD for our classification problem. To accomplish this, we compute the first k left singular vectors $\{u_1, ..., u_k\}$ of each digit class. We select only the first k as a form of dimensionality reduction. That is, we discard irrelevant or noisy components that may cause inaccuracy in predictions. Essentially, using the first k vectors allows us to keep vectors representing directions along which the data varies the most with $\{u_1, ..., u_k\}$ capturing the most discriminative features of each class.

Once we have separated the training data into 9 groups based on the digit in the image, compute a sparse SVD of each group. In the code, the variable X_train represents the class-separated collection of training images.

$$\text{best rank-k approximation of } X^{(j)} = U_k^j \Sigma_k^j V_k^{jT} \tag{2}$$

A well selected k will allow the range of $X^j$ to be approximately equivalent to the range of $U_k^j$. This is significant because with $U_k^j$ forming part of $X^{(j)}$'s orthonormal basis, we are able to construct a k-term approximation of any training digit $x_i$ that is part of j digit class. More specifically, the best k-term approximation of $x_i$ of digit j is given by:

$$U_k^{(j)}(U_k^{(j)T} x_i) \tag{3}$$

However, some assumptions are made upon which this method depends. Firstly, each digit of each group in the training set needs to be 'well characterized'. That is, each digit within a class needs to share similar archetypal features with all other members of the class. For instance, all 1s must be a straight line in the center of the image. As such, approximating a digit of one class using the left singular vectors of a different class will result in a significantly large approximation error. Conversely, classifying a digit $y_i$ from the test set will return the minimum approximation error when $U_k^{(j)}$ is used, if $y_i$ is from digit class j.

To compute the approximation error for each test digit, utilize the formula below which finds the 2-norm of residual errors for each digit class. As mentioned previously, the minimum of these will be chosen as the predicted label.

$$E_j(y_i) := ||y_i - U_k^{(j)}(U_k^{(j)T} y_i)||_2, \text{ j} = 0, 1, ...9 \tag{4}$$

For this code implementation, an equivalent formula will be employed. Because the dimension of this image problem is quite small, ($d = 256$) it is acceptable to use.

$$E_j(y_i) := (I_d – U_k^{(j)} U_k^{(j)T}) y_i, \text{ j} = 0, 1, ...9 \tag{5}$$

The function usps_svd_classification in the code appendix will perform a comprehensive test of the classification method on a range of k values selected by the user, returning the accuracy rate of each k as well as an array of the best predictions.

## 4 Results

To demonstrate the efficacy of the methods described above, their predictions will be compared to the true labels via a confusion matrix. A confusion matrix is created thusly: The main diagonal represents the quantity of digits with true label j that were predicted to belong to j. Every other entry $a_{i,j}$ such that $i \neq j$ represents a sum amount of misclassified digits.

Firstly, examine the confusion matrix of the simple method to compare how well the algorithm performs in predicting the true labels of digits in the test set.

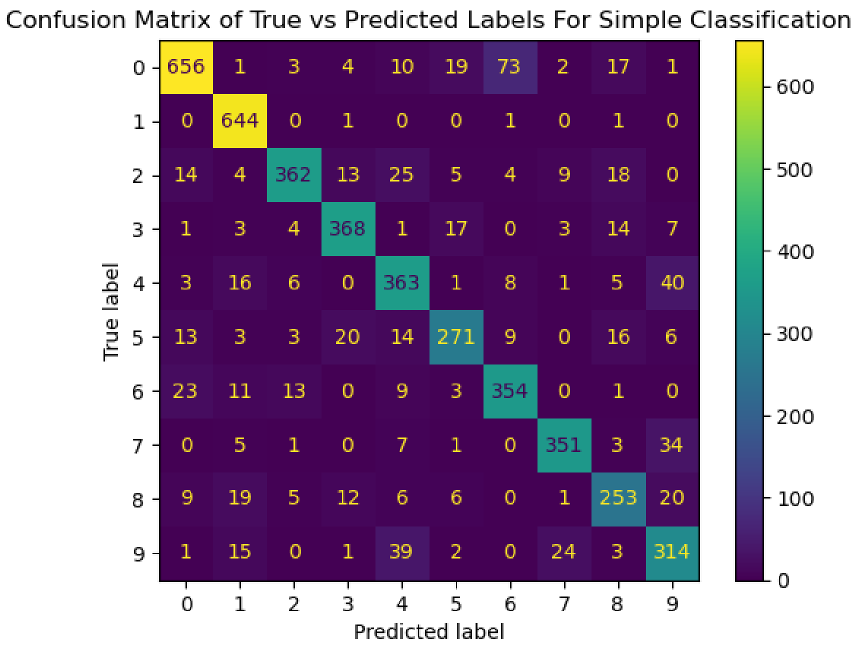


Figure 3: Simple Classification Confusion Matix

The simple method had an overall classification accuracy rate of 84.66%. It is clear that while this distance based algorithm does work somewhat, it is unable to capture and interpret the complex characteristics of the data's features. For instance the method stumbles on digits that are generally similar, like 0 and 6, 4 and 9, 3 and 8. Because each of these pairs have similar pixel values in similar locations, they are misclassified at a high rate.

We then utilize the SVD method and functions discussed above with a range of k values (1, 2, ... , 20) to find the first k left singular vectors of the training data. Then, make predictions by minimizing the approximation error on the test data. The figure below shows the accuracy rate of predictions made with each k.

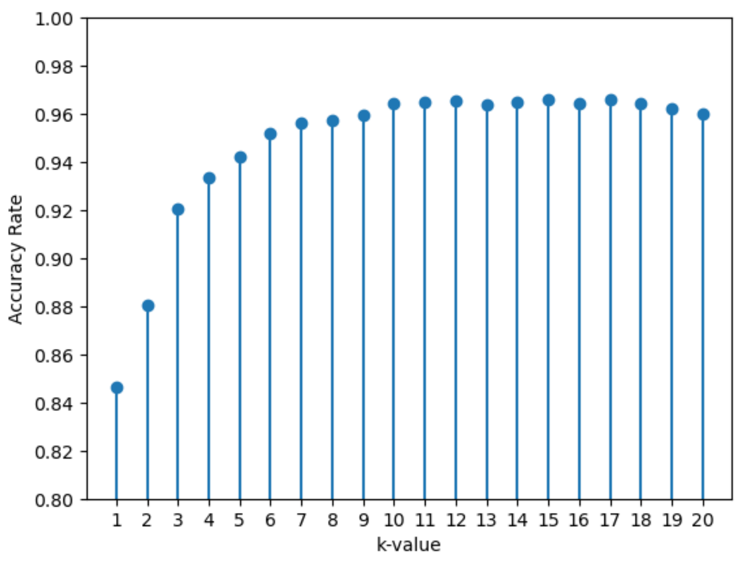
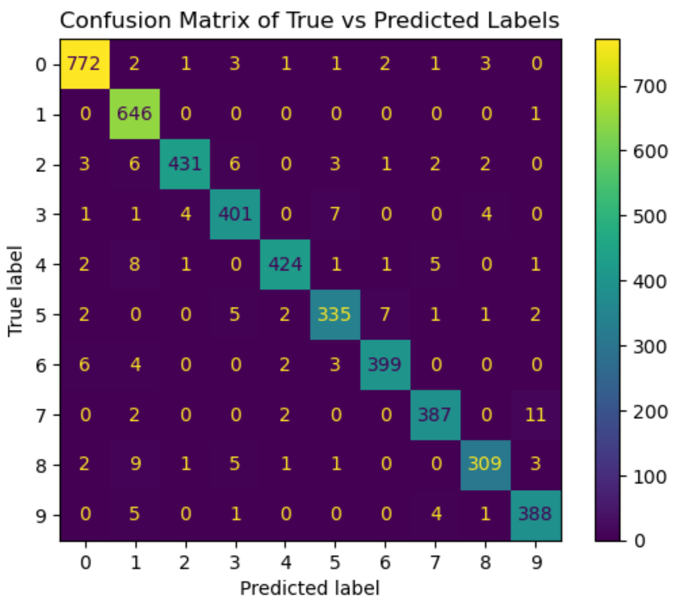


Figure 4: Accuracy rate of predictions made with each k

As expected, including many left singular vectors will eventually cause the accuracy to plateau, as not enough new information is granted by increasing k. The maximum accuracy rate of 96.62% is reached with $k = 17$.

As before, it would be pertinent to examine the confusion matrix created by comparing the the predictions generated by the optimal k = 17 against the true test labels.



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 772 | 2 | 1 | 3 | 1 | 1 | 2 | 1 | 3 | 0 |
| 1 | 0 | 646 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 2 | 3 | 6 | 431 | 6 | 0 | 3 | 1 | 2 | 2 | 0 |
| 3 | 1 | 1 | 4 | 401 | 0 | 7 | 0 | 0 | 4 | 0 |
| 4 | 2 | 8 | 1 | 0 | 424 | 1 | 1 | 5 | 0 | 1 |
| 5 | 2 | 0 | 0 | 5 | 2 | 335 | 7 | 1 | 1 | 2 |
| 6 | 6 | 4 | 0 | 0 | 2 | 3 | 399 | 0 | 0 | 0 |
| 7 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 387 | 0 | 11 |
| 8 | 2 | 9 | 1 | 5 | 1 | 1 | 0 | 0 | 309 | 3 |
| 9 | 0 | 5 | 0 | 1 | 0 | 0 | 0 | 4 | 1 | 388 |

Figure 5: SVD Base Classification Confusion Matrix for k = 17

Despite this much improved performance over the simple classification method examined in 3.1, we can see that similar errors still occur (just to a lesser degree). For instance, some 7s are misclassified as 9s, and some 8s are perceived to be 1s by the algorithm. Of course, this is understandable because we are not utilizing deep learning methods.

## 5 Conclusions

Utilizing the dataset of handwritten images put forward by the US Postal service, this paper scrutinized two classification methods. The first, a straightforward Euclidean distance-based approach, while demonstrating some predictive capability with a peak accuracy of 84.66%, struggled with distinguishing similar-looking digits due to its simplistic representation of features. The second method involved employing Sparse SVD. This method focuses on isolating the most important archetypal features for each digit class and predicting based on their presence or absence in the input. By extracting the first k left singular vectors for each digit class from the training data, this method achieved significant accuracy rates. The performance peaked at 96.62% accuracy with $k = 17$, showcasing the potential of Sparse SVD for classification tasks.

The examination of confusion matrices for both methods highlighted the superiority of Sparse SVD over the basic distance-based approach. However, even with Sparse SVD, some misclassifications persisted, particularly among digits with resemblances in their pixel values.

Despite the residual errors, the paper successfully demonstrated the effectiveness of Sparse SVD as a computationally inexpensive yet highly accurate method for handwritten digit classification. This exploration not only sheds light on the practical application of SVD but also underscores the importance of feature representation.

However, it is also important to consider some shortcomings of SVD as a classification algorithm. Primarily, it becomes computationally expensive when applied to high dimensional datasets when the rank used for approximation is also high. Additionally storing matrices in memory for large datasets becomes very demanding resource-wise. Thankfully, there are several other image classification methods and general techniques that provide solutions to the issues SVD based algorithms face. These include Support Vector Machines, Naive-Bayes Classifiers, Convolutional Networks and various clustering techniques.

## 6 Code Appendix

### 6.1 Importing Dependencies & Loading Data

```
from scipy import io
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse.linalg import svds
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

df = io.loadmat('usps.mat')
X_train_img, y_train = df['train_patterns'], df['train_labels']
X_test_img, y_test = df['test_patterns'], df['test_labels']
```

### 6.2 Helper Functions

```
#Function for decoding labels from one-hot encoding
# For each label, check index of 1 and return it as label. Outputs a list.
def oh_decode(oh_labels):
    decoded_labels = []
    for col in oh_labels.T:
        for i, e in enumerate(col):
            if e == 1:
                decoded_labels.append(i)

    return decoded_labels


#Function for grouping images based on the digit represented.
#Returns an array of arrays where the array at position 0 is all images of 0, etc.
def group_img(X, y):
    group_array = []
    labels = oh_decode(y) #Decodes corresponding labels

    for x in range(10): #Iterates through all images for each digit class, groups together
        x_matrix = []
        dim = np.unique(labels, return_counts=True)[1][x]
        for i, col in enumerate(X.T):
            if y[:,i][x] == 1:
                x_matrix.append(np.array(col.reshape(-1)))
        x_matrix = np.array(x_matrix)
        group_array.append(x_matrix)
    # output as array of arrays
    return group_array


#Creating new collections for training images based on ordered dataset created by group_img function
X_train = group_img(X_train_img,y_train)
X_test = group_img(X_test_img,y_test)
```

### 6.3 Plotting Individual Samples and Average Figures

```
#Naive approach to plotting the first 16 images of the training dataset.
fig, axs = plt.subplots(4, 4, figsize=(12, 6))
```

```
axs[0,0].imshow(np.reshape(X_train_img[:,0], (16, 16)), cmap='gray')
axs[0,0].axis("off")
axs[0,1].imshow(np.reshape(X_train_img[:,1], (16, 16)), cmap='gray')
axs[0,1].axis("off")
axs[0,2].imshow(np.reshape(X_train_img[:,2], (16, 16)), cmap='gray')
axs[0,2].axis("off")
axs[0,3].imshow(np.reshape(X_train_img[:,3], (16, 16)), cmap='gray')
axs[0,3].axis("off")
axs[1,0].imshow(np.reshape(X_train_img[:,4], (16, 16)), cmap='gray')
axs[1,0].axis("off")
axs[1,1].imshow(np.reshape(X_train_img[:,5], (16, 16)), cmap='gray')
axs[1,1].axis("off")
axs[1,2].imshow(np.reshape(X_train_img[:,6], (16, 16)), cmap='gray')
axs[1,2].axis("off")
axs[1,3].imshow(np.reshape(X_train_img[:,7], (16, 16)), cmap='gray')
axs[1,3].axis("off")
axs[2,0].imshow(np.reshape(X_train_img[:,8], (16, 16)), cmap='gray')
axs[2,0].axis("off")
axs[2,1].imshow(np.reshape(X_train_img[:,9], (16, 16)), cmap='gray')
axs[2,1].axis("off")
axs[2,2].imshow(np.reshape(X_train_img[:,10], (16, 16)), cmap='gray')
axs[2,2].axis("off")
axs[2,3].imshow(np.reshape(X_train_img[:,11], (16, 16)), cmap='gray')
axs[2,3].axis("off")
axs[3,0].imshow(np.reshape(X_train_img[:,12], (16, 16)), cmap='gray')
axs[3,0].axis("off")
axs[3,1].imshow(np.reshape(X_train_img[:,13], (16, 16)), cmap='gray')
axs[3,1].axis("off")
axs[3,2].imshow(np.reshape(X_train_img[:,14], (16, 16)), cmap='gray')
axs[3,2].axis("off")
axs[3,3].imshow(np.reshape(X_train_img[:,15], (16, 16)), cmap='gray')
axs[3,3].axis("off")

plt.tight_layout()
plt.show()

#Calculate the average of each digit class and append to an array containing averages.
train_aves = []
test_classif_res = []
for i in range(10):
    ave = np.mean(X_train[i], axis = 0)
    train_aves.append(ave)

train_aves = np.array(train_aves)

#Naive approach to plotting each average figure per digit class.
fig, axs = plt.subplots(2, 5, figsize=(12, 6))
axs[0,0].imshow(np.reshape(train_aves[0], (16, 16)), cmap='gray')
axs[0,0].axis("off")
axs[0,1].imshow(np.reshape(train_aves[1], (16, 16)), cmap='gray')
axs[0,1].axis("off")
axs[0,2].imshow(np.reshape(train_aves[2], (16, 16)), cmap='gray')
axs[0,2].axis("off")
axs[0,3].imshow(np.reshape(train_aves[3], (16, 16)), cmap='gray')
```

```
axs[0,3].axis("off")
axs[0,4].imshow(np.reshape(train_aves[4], (16, 16)), cmap='gray')
axs[0,4].axis("off")
axs[1,0].imshow(np.reshape(train_aves[5], (16, 16)), cmap='gray')
axs[1,0].axis("off")
axs[1,1].imshow(np.reshape(train_aves[6], (16, 16)), cmap='gray')
axs[1,1].axis("off")
axs[1,2].imshow(np.reshape(train_aves[7], (16, 16)), cmap='gray')
axs[1,2].axis("off")
axs[1,3].imshow(np.reshape(train_aves[8], (16, 16)), cmap='gray')
axs[1,3].axis("off")
axs[1,4].imshow(np.reshape(train_aves[9], (16, 16)), cmap='gray')
axs[1,4].axis("off")

plt.tight_layout()
plt.show()
```

## 6.4 Testing Simple Classification Algorithm

```
#Calculate predictions for the distance based approach
dists = []
test_classif = []
for i in range(X_test_img.shape[1]):
    for a in train_aves:
        dist = np.array(np.sqrt(sum((X_test_img[:,i] - a)**2))) #Distance formula
        dists.append(dist)
    test_classif.append(np.array(dists).T) #Append transposed vectors of distances.
    dists = []

predictions = []

#for each column in the array of distances, find minimum and take the label.
for i in range(X_test_img.shape[1]):
    predictions.append(np.argmin(test_classif[i], axis = 0))

test_classif_res = np.array(predictions).reshape(1, 4649)

#Create a confusion matrix
y_true = oh_decode(y_test) #Decode test labels
cm_labels = [0,1,2,3,4,5,6,7,8,9]
#Using sklearn for plotting the confusion matrix
test_confusion = confusion_matrix(y_true, predictions,
                                            labels = cm_labels)
test_confusion_display = ConfusionMatrixDisplay(confusion_matrix = test_confusion,
                                                                display_labels=cm_labels)

test_confusion_display.plot()
plt.title('Confusion Matrix of True vs Predicted Labels For Simple Classification')
plt.show()
```

## 6.5 SVD Classification & Function for finding best k

```
def usps_svd_classification(X_train, X_test_img, y_train, y_test, k_max):
    """
Implemented the rank k SVD-based classification on the usps dataset
INPUT:
- X_train: matrix of training images grouped by digit j
- X_test_img: matrix of test images NOT grouped by digit j (original order)
- y_train: one-hot encoded labels for the train data
- y_test: one-hot encoded labels for the test data
- k: maximum value for k that will be tested by algorithm
OUTPUT:
- accuracies: the overall classification accuracies of the test_patterns for each k
- test_predict: the predicted label for the test_patterns for the best k.
"""

    #function that selects minimum value for error of rank-k approximation
    def predict(X_test_img):
        classes = [U0,U1,U2,U3,U4,U5,U6,U7,U8,U9]
        preds = []
        for U in classes: #For each digit class, compute the error
            preds.append(np.linalg.norm((np.identity(len(U))-np.matrix(U)*np.matrix(U.T)).dot(
                                            X_test_img),ord=2)/np.linalg.norm(X_test_img,ord=2))
    #Computes the k-term approximation for each set of left singular values
        return preds.index(min(preds))
    accuracies = []

    y_true = oh_decode(y_test) #decodes labels to their respective values
    for k in range(1,k_max+1): #compute singular vectors for each digit based on K
        U0, _, _ = svds(X_train[0].T, k)
        U1, _, _ = svds(X_train[1].T, k)
        U2, _, _ = svds(X_train[2].T, k)
        U3, _, _ = svds(X_train[3].T, k)
        U4, _, _ = svds(X_train[4].T, k)
        U5, _, _ = svds(X_train[5].T, k)
        U6, _, _ = svds(X_train[6].T, k)
        U7, _, _ = svds(X_train[7].T, k)
        U8, _, _ = svds(X_train[8].T, k)
        U9, _, _ = svds(X_train[9].T, k)
        predictions = []
        for i in range(X_test_img.shape[1]):
            predictions.append(predict(X_test_img[:,i]))
        correct = 0
#calculates the accuracy rate for each digit class based on predictions
        for i, elem in enumerate(predictions):
            if elem == y_true[i]:
                correct += 1
        accuracy = correct / len(predictions)
        accuracies.append(accuracy)
        if accuracy == max(accuracies):
#If the accuracy is the best in the list currently, update to keep its corresponding predictions
            test_predict = predictions
            best_k = k

    return accuracies, test_predict
#run the function for each k in range of 20
accuracies, best_pred= usps_svd_classification(X_train, X_test_img, y_train, y_test, 20)
```

```
#plot each accuracy along with its corresponding k.
fig, ax = plt.subplots(facecolor='white')
ax.stem(range(1,21), accuracies)
y = np.array([0.8, 1.02])
plt.ylim(0.8,1)
ax.xaxis.set(ticks =list(range(1,21)), label_text = 'k-value' )
ax.yaxis.set(ticks=np.arange(y.min(), y.max(), 0.02), label_text='Accuracy Rate')
plt.yticks(np.arange(y.min(), y.max(), 0.02))
plt.show()
```

## 6.6 SVD Method Confusion Matrix

```
#Once again utilizing the sklearn confusion matrix display functionality
cm_labels = [0,1,2,3,4,5,6,7,8,9]
test_svd_confusion = confusion_matrix(y_true, best_pred, labels = cm_labels)
test_svd_confusion_display = ConfusionMatrixDisplay(confusion_matrix = test_svd_confusion,
                                                    display_labels=cm_labels)

#This time, plot the comparison between SVD method prediction and true labels
test_svd_confusion_display.plot()
plt.title('Confusion Matrix of True vs Predicted Labels')
plt.show()

#Snippet of leftover code to display the overall accuracy % of the best predictions.
correct = 0
for i, elem in enumerate(best_pred):
    if elem == y_true[i]:
        correct += 1

correct / len(best_pred)
```