AMATH 482 Homework 4

Vineet Palepu

March 10, 2021

1. Abstract

In this paper, we examine the application of linear discriminant analysis for performing digit classification. We used the linear discriminant analysis to classify handwritten digits between two classes, three classes, and all 10 classes. We then compared these results to common techniques applied like support vector machines and decision tree classifiers. We then analyzed the performance of these techniques and how certain factors such as the digits being classified and the number of features affected the performance.

2. Introduction and Overview

We are given a list of images containing handwritten digits and are tasked with developing an algorithm to recognize the digits. We perform the singular value decomposition which helps us determine the rank of the data. Then, we project the data onto 3 of the right singular vector modes to help visualize the dataset. Once we have done this, we perform linear discriminant analysis by projecting our data onto a line that best separates the chosen digits. We then test out various combinations of digits seeing how accurate our classifier is at discriminating between pairs of digits, three digits, and then finally all 10 digits. For comparison, we also tested the accuracy of support vector machines and decision tree classifiers, which were once considered state of the art techniques for classification.

3. Theoretical Background

3.1. Singular Value Decomposition

The Singular Value Decomposition (SVD) is a matrix factorization of the form shown in Equation (1), where U and V are unitary matrices and Σ is a diagonal matrix. Because unitary matrices do not stretch or compress a vector and diagonal matrices only stretch vectors, the SVD can be visualized as a rotation, followed by a dilation, followed by another rotation.

$$A = U\Sigma V^* \tag{1}$$

$$A \in \mathbb{R}^{m\times n}, U \in \mathbb{R}^{m\times m}, \Sigma \in \mathbb{R}^{m\times n}, V \in \mathbb{R}^{n\times n}$$

In the SVD, U is the matrix of left singular vectors of A, V is the matrix of right singular vectors of A, and Σ is the matrix of singular values of A, with the singular values located on the diagonal. Additionally, the singular values are ordered from largest to smallest, thus ensuring that every matrix has a unique SVD. Important to note is that all matrices, no matter the size or shape, have a singular value decomposition.

Further, the SVD is useful for forming low-rank approximations of a matrix. If the matrix A is $m \times n$ and $rank(A) = r$, then it can be expressed as shown in Equation (2), where $u_i v_i^*$ is the outer product of $m \times 1$ and $1 \times n$ matrices. Further, if we instead sum up to N instead of r, where $N \leq r$, we can form the best rank N approximation of A.

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^* \tag{2}$$

In order to compute the SVD of a matrix, Equation (3) shows us that V and Σ^2 are the eigenvalues and eigenvectors of A^*A. This can also be done for AA^* as shown in Equation (4), showing that U and Σ^2 are the eigenvalues and eigenvectors of AA^*.

$$A^*A = (U\Sigma V^*)^*(U\Sigma V^*) \tag{3}$$

$$= V\Sigma U^* U\Sigma V^*$$
$$= V\Sigma^2 V^*$$
$$A^*AV = V\Sigma^2$$

$$AA^* = U\Sigma^2 U^* \tag{4}$$
$$AA^*U = U\Sigma^2$$

The singular value decomposition is useful in this application because it provides us with a basis where variation among the data is maximized, a feature useful for linear discriminant analysis.

3.2. Linear Discriminant Analysis

Linear discriminant analysis (LDA) is a technique where data is projected onto a new basis, usually a line, where it can more easily be analyzed. This projection is the one that maximizes the distance between the different classes' data, while minimizing the variance within a class. Once this is done, a threshold can be established that best separates the two sets of data. The projection is calculated by find a vector w as shown in Equation (5), where S_B is the between class scatter matrix, defined in Equation (7) and S_w is the within class scatter matrix, defined in Equation (8). Note that w is actually calculated by finding the vector corresponding to the maximum eigenvalue of the generalized eigenvalue problem shown in Equation (6). μ_j is the mean of the column vectors of the j^{th} class, μ is the mean of the column vectors of all classes, and x_{ji} is the i^{th} sample of the j^{th} class.

$$w = argmax \frac{w^T S_B w}{w^T S_w w} \tag{5}$$

$$S_B w = \lambda S_w w \tag{6}$$

$$S_B = \sum_{j=1}^{N} (\mu_j - \mu)(\mu_j - \mu)^T \tag{7}$$

$$S_w = \sum_{j=1}^{N} \sum_{i=1}^{M} (x_{ji} - \mu_j)(x_{ji} - \mu_j)^T \tag{8}$$

When it comes to determining the classes, if there are only two classes, a threshold can be calculated that maximizes the accuracy of the classifier on the data. If multiple classes are used, a single threshold cannot be used, so a different technique must be applied. In our case, we simply calculated the mean that was closest. This has similar results to calculating a threshold for only two classes, while still performing adequately with more classes.

3.3. Support Vector Machines

A support vector machine is a type of supervised machine learning algorithm that classifies data by finding a hyperplane in N-dimensional space that separates the data points by class. The specific hyperplane chosen is one that maximizes the margin, or distance between each point in a class and the plane.

3.4. Decision Tree Classifier

A decision tree classifier is a supervised machine learning algorithm that classifies examples by continually splitting the data according to some parameters.

4. Algorithm Implementation and Development

Our first step is to load the MNIST training and test datasets and labels. The code to do this can be found in the function $loadMNIST()$, located in Appendix B. We then reshape and rescale the data so that each pixel value is between 0 and 1.

Next, we computed the SVD of the entire dataset (with the mean subtracted) and visualized the first few left singular vectors, the columns of U, shown in Figure 1. The interpretation of the U matrix is that it can be thought of as the most important distinguishing features of the different digits. To get an idea of what the rank of the dataset was, we plotted the singular values, the diagonal elements of Σ shown in Figure 2. Note that even the smallest singular value is larger than 0. We can interpret the Σ matrix as how important each of the distinguishing features is. That is, a high singular value in the first row and column means the first feature contains a significant portion of the variance in the dataset. Next, we wanted to visualize different low-rank approximations of the dataset. We zeroed out varying numbers of the smallest significant values and visualized a random digit to see how recognizable it was. Some approximations are shown in Figure 3. Finally, we wanted to project the dataset onto 3 of the columns of V and visualize it. The V matrix can be interpreted as follows: the columns vectors of V show us how each of the digits is represented in the new basis. We chose the first three columns as those have the largest variance, and plotted each digit in a different color, shown in Figure 4.

Now that we had a decent understanding of the dataset, we set out to creating the classifier. We started with a classifier for distinguishing two digits. To do so, we take the data for each of the two selected digits, combine them, and compute the SVD. We then calculate the within and between class scatter matrices, and compute the generalized eigenvectors as in Equation (6). From this, we calculate the maximal eigenvalue and eigenvector pair. We then project our data onto this vector, w, and calculate the mean of each digit's projection. Then, we calculate the threshold value. This code can be found in the function $digit\_trainer()$ in Appendix B.

To make predictions based on this model, we project the data onto the vector by calculating $w^T * U^T * data$ and then check the value against our threshold, and based on this, classify the digit. This code is found in the function $predict()$ in Appendix B. Finally, we can calculate the accuracy on the test dataset. This is important to ensure that our model isn't overfitting, and that it is generalizable to future data. One thing to note is that we didn't subtract the mean from our data as is typical with PCA, as we found that it had minimal effect on the accuracy.

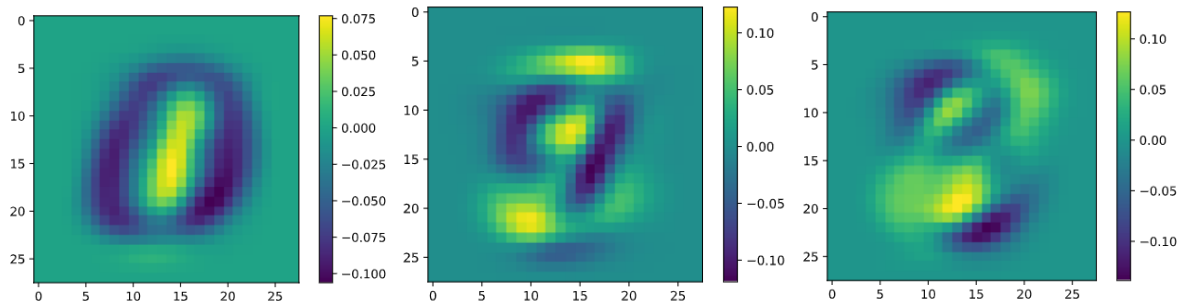

Figure 1. This shows the first three columns of U visualized. Parts of different digits can be seen in each of the three visualizations shown above.

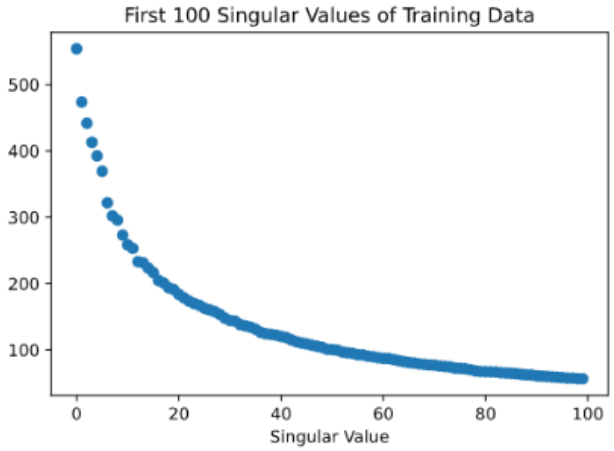

Figure 2. Here we see the first 100 singular values from the SVD of our training dataset. While the first 10 singular values are significantly larger than the rest, even the 100th singular value still is much larger than 0, and thus contributes some information about the digit.

Now, we wanted to find out which digits were easiest and most difficult to differentiate. So, we trained a classifier on every possible combination of two digits. To do this, we looped through every combination, training a classifier, and then calculating and storing the training set accuracy as well as the test set accuracy. We then sorted these accuracies to figure out the easiest and most difficult digits to classify.

Next, we want to try classifying amongst three or more digits. We repeat the same process as above, but with minor modifications. Rather than providing a threshold value, we instead return the mean of each of the digits' projection. The code for training a 3 or more digit classifier can be found in the function $digit\_trainer\_multiple()$ in Appendix B. Then, to predict, we simply calculate the projection onto the w vector as usual, but instead of comparing with a threshold value, compare to find which mean is closest. The code for predicting digits for 3 or more classes can be found in the function $predict\_multiple()$ in Appendix B.

Finally, to benchmark our LDA classifier against typical methods, we trained a support vector machine and decision tree classifier to distinguish amongst all the digits.

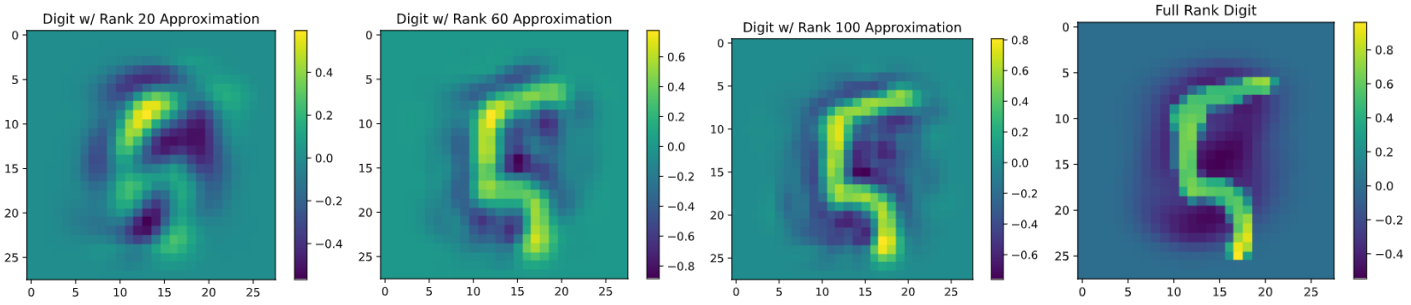

Figure 3. Here we see different low-rank approximations of a single training example. As can be seen, the higher the rank of the approximation, the closer it is to the full rank digit (right).

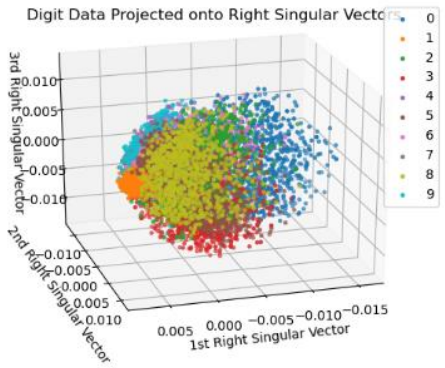

Figure 4. Here we have the projection of the data onto the first three right singular vectors. As we can see, each digit is grouped together on the projection.

5. Computational Results

After training the two digit classifier on all combinations using 20 features, we found that the most difficult digits to differentiate were 3 and 5, with training accuracy of 90.4% and test accuracy of 92.2%. The pairs 7 and 9, 5 and 8, and 4 and 9 also had similar accuracies of around 92% to 93% on the training set and test set. The easiest digits to classify were 0 and 1, having accuracies of 99.7% on the training set and 99.9% on the test set. The full matrix of digit accuracies is shown in Figure 5. An interesting note is that the classifier on the easiest digits had an accuracy of above 99% for as few as 2 features. It was only when using only the most significant singular value (using 1 feature) that accuracy dropped to around 93%. However, the difference is much more noticeable for the difficult to classify digits. When using 2 features for differentiating between 3 and 5, the accuracy dropped from above 90% all the way to 64%. Still, with as few as 5 features, the accuracy was around 90%. Additionally, adding more features didn't increase the accuracy. Even with 500 out of 784 features, the accuracy for classifying 0 vs 1 was identical to 20 features. However, for the digits 3 and 5, the accuracy rose to 96.4% on the test set when using 500 features.

When training for 3 digits, accuracy varied depending on the digits chosen. Certain combination such as 0,1,4 had training and test accuracies of 92.8% and 92.6%. However, other combinations like 3,5,8 had lower accuracies of 71.0% and 73.0% for the training and test dataset, respectively. We also tested classification of all the digits at once, and found accuracies of 33.9% and 33.7% for the training and test set respectively. While this is certainly better than random chance, accuracy is significantly lower than when comparing only 2 or 3 digits. This is likely due to the fact that as the number of digits increases, the projections are likely closer together, or even overlapping, and thus more difficult to differentiate. Additionally, if we increase the number of features to 500, we find a test accuracy of 44%, though computation time is increased. Interestingly, when using all 784 features, despite computation time being significantly longer, we got an accuracy of 22%. This was unexpected and we are unsure as to why exactly this occurs, though we hypothesize that it may be due to the smaller features consisting of "noise" that is unimportant to the digit. A similar pattern of decreased accuracy using all features was found for classifying between 0 and 1.

The SVM classifier achieved an accuracy of 97.92% on the test set when trained on all 10 digits. This was significantly higher than the accuracy for our classifier, which only got around 34%. The decision tree classifier was similarly successful, though not as much, getting an accuracy of 87.34% on the test dataset.

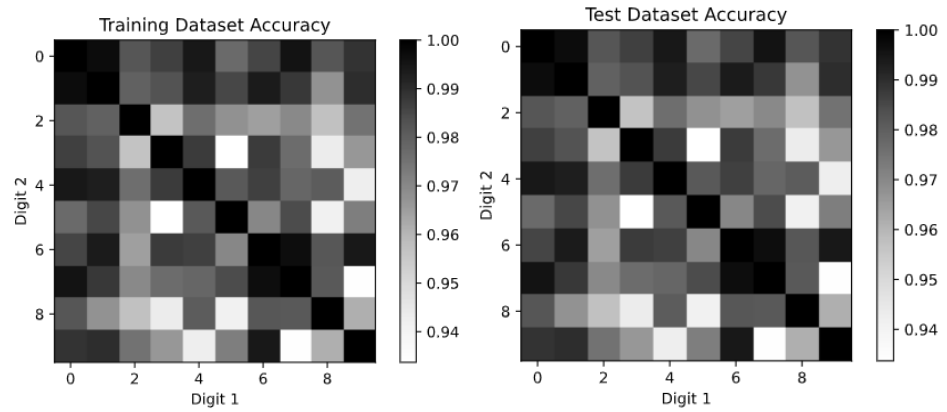

Figure 5. Here, we visualize the different training (left) and test (right) dataset accuracies for each combination of digits when trained with 20 features. The diagonal is 1 as predicting for the same digit always gives the correct answer. Also note that each matrix is symmetric across the diagonal.

When trained on the hardest digits, 3 and 5, the SVM classifier got an accuracy of 99.36% on the test dataset, and the decision tree classifier got 95.74%. Both these values are significantly higher than the test accuracy of our LDA classifier, which was 92.22%.

For the easiest digits, 0 and 1, the SVM had an accuracy of 99.95% on the test set and the tree classifier had an accuracy of 99.67%. In this case, the SVM has a slightly higher accuracy than the 99.91% of the LDA, and the LDA had a slightly higher accuracy than the tree classifier.

6. Summary and Conclusions

We were given the task of training a classifier using LDA to differentiate between different handwritten digits. We were successfully able to differentiate between pairs of digits with accuracies of above 90%. When differentiating between three digits, accuracies were still relatively high, ranging from 70% to 90%. However, this technique was unsuccessful when attempting to classify amongst all 10 digits, with an accuracy of between 33% and 44% (or as low as 22% if using all features), depending on the number of features used. The number of features to use is an important parameter to consider when training an LDA classifier. Using more features can increase the accuracy (or paradoxically decrease it if too many are used), but also increases the time required for training, as well as predicting (though to a lesser degree). We conclude that very few features (<10) can be used when classifying between pairs of digits. As the complexity of the problem increases (for example if you are trying to differentiate difficult digits, or if you are trying to classify among a larger number, say 3, 4, or more digits), increasing the features can provide increases in accuracy. When comparing this method to state of the art methods, we find that the accuracy is lower, especially as the number of classes increases.

7. References

Chakure, A. Decision Tree Classification. *Medium* (2020). Available at: https://medium.com/swlh/decision-tree-classification-de64fc4d5aac.

Gandhi, R. Support Vector Machine - Introduction to Machine Learning Algorithms. *Medium* (2018). Available at: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

8. Appendix A: Functions Used

Python

NumPy

`np.abs` : Returns the absolute value of a matrix.

`np.amax` : Returns the largest value of a matrix. Used to normalize certain matrices.

`np.arange` : Generates a vector with consecutive integer values.

`np.argmax` : Gets the index of the maximal element

`np.argmin` : Gets the index of the minimal element

`np.linalg.svd` : Calculates the SVD of a matrix. Used to get the singular values and U, V matrices for the LDA.

`np.mean` : Calculates the mean of a values of a matrix along a specified axis

`np.unique` : Gets the unique elements of a matrix. Used to determine what digits are in a data subset

`np.vstack` : Vertically concatenates matrices.

MatPlotLib

`pyplot.plot` : Creates a graph of the data. We used this to visualize various data.

SciPy

`sp.linalg.eig` : Used solve the generalized eigenvalue problem to find the vector w to project onto

SciKitLearn

`sklearn.tree.DecisionTreeClassifier` : Used to create a decision tree classifier for the handwritten digit data.

`sklearn.svm.SVC` : Used to create a support vector machine for the handwritten digit data.

`fit` : Trains the model on the given data

`score` : Calculates the model accuracy on a dataset

9. Appendix B: Code

```
# %%
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
from scipy import linalg

# Load the mnist data from file
# From https://stackoverflow.com/a/53226079
def loadMNIST( prefix, folder ):
    intType = np.dtype( 'int32' ).newbyteorder( '>' )
    nMetaDataBytes = 4 * intType.itemsize

    data = np.fromfile( folder + "/" + prefix + '-images-idx3-
ubyte', dtype = 'ubyte' )
    magicBytes, nImages, width, height = np.frombuffer( data[:nMetaDataBytes].tob
ytes(), intType )
    data = data[nMetaDataBytes:].astype( dtype = 'float32' ).reshape( [ nImages,
width, height ] )

    labels = np.fromfile( folder + "/" + prefix + '-labels-idx1-ubyte',
                          dtype = 'ubyte' )[2 * intType.itemsize:]

    return data, labels


# %%
# Load Data from Files
trainingImages, trainingLabels = loadMNIST( "train", "./datasets" )
testImages, testLabels = loadMNIST( "t10k", "./datasets" )

train_labels = trainingLabels
test_labels = testLabels

# Swap axes
train_data = np.moveaxis(trainingImages, 0, -1)
test_data = np.moveaxis(testImages, 0, -1)

# Flatten images
train_data = np.reshape(train_data, (train_data.shape[0] * train_data.shape[1], t
rain_data.shape[2]))
test_data = np.reshape(test_data, (test_data.shape[0] * test_data.shape[1], test_
data.shape[2]))

# Rescale to [0, 1]
train_data = train_data / np.max(train_data)
test_data = test_data / np.max(test_data)
```

```
# %%
U, S, V = np.linalg.svd(train_data - np.mean(train_data, axis=1)[:, None], full_m
atrices=False)
#U, S, V = np.linalg.svd(train_data, full_matrices=False)


# %%
plt.imshow(np.reshape(U[:, 3], (28, 28)))
plt.colorbar()
plt.show()


# %%
plt.plot(S[0:100], 'o')
plt.title("First 100 Singular Values of Training Data")
plt.xlabel("Singular Value")
plt.show()


# %%
# Create low-rank approximation and visualize digits for comparison
rank = int(784/2)
img_num = 100

S_partial = np.copy(S)
S_partial[rank:] = 0
reconstruction = U @ np.diag(S_partial) @ V
original = U @ np.diag(S) @ V
plt.title("Full Rank Digit")
plt.imshow(np.reshape(original[:, img_num], (28, 28)))
plt.colorbar()
plt.show()
plt.title("Digit w/ Rank " + str(rank) + " Approximation")
plt.imshow(np.reshape(reconstruction[:, img_num], (28, 28)))
plt.colorbar()
plt.show()


# %%
projections = []
for i in range(10):
    projections.append(V[0:3, train_labels == i])
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(10):
    # Sample 1000 points for each digit to make visualization easier
    indices = np.random.randint(projections[i].shape[1], size = 1000)
```

```
    data = projections[i][:, indices]
    #data = projections[i][:, :]

    ax.scatter(data[0, :], data[1, :], data[2, :], s=6)
plt.title("Digit Data Projected onto Right Singular Vectors")
ax.set_xlabel("1st Right Singular Vector")
ax.set_ylabel("2nd Right Singular Vector")
ax.set_zlabel("3rd Right Singular Vector")
ax.legend(("0", "1", "2", "3", "4", "5", "6", "7", "8", "9"), loc="left")
plt.show()


# %%
def digit_trainer(digit1_data, digit2_data, num_features):
    U, S, V = np.linalg.svd(np.hstack((digit1_data, digit2_data)), full_matrices=
False)
    U = U[:, 0:num_features]

    SV = np.diag(S) @ V
    digit1 = SV[0:num_features, :digit1_data.shape[1]]
    digit2 = SV[0:num_features, digit1_data.shape[1] + 1:]


    d1_m = np.mean(digit1, axis=1)
    d2_m = np.mean(digit2, axis=1)

    S_w = 0
    for i in range(digit1.shape[1]):
        S_w += (digit1[:, i] - d1_m).reshape(-
1, 1) @ (digit1[:, i] - d1_m).reshape(-1, 1).T

    for i in range(digit2.shape[1]):
        S_w += (digit2[:, i] - d2_m).reshape(-
1, 1) @ (digit2[:, i] - d2_m).reshape(-1, 1).T

    S_b = (d1_m - d2_m).reshape(-1, 1) @ (d1_m - d2_m).reshape(-1, 1).T

    eigenvalues, eigenvectors = sp.linalg.eig(S_b, S_w)
    max_eig_ind = np.argmax(np.abs(eigenvalues))
    w = eigenvectors[:, max_eig_ind]
    w = w / np.linalg.norm(w, ord=2)

    proj_d1 = w.T @ digit1
    proj_d2 = w.T @ digit2

    if np.mean(proj_d1) > np.mean(proj_d2):
        w = -w
        proj_d1 = -proj_d1
```

```
        proj_d2 = -proj_d2

    sort_d1 = np.sort(proj_d1)
    sort_d2 = np.sort(proj_d2)

    t1 = len(sort_d1) - 1
    t2 = 0

    while sort_d1[t1] > sort_d2[t2]:
        t1 -= 1
        t2 += 1

    threshold = (sort_d1[t1] + sort_d2[t2]) / 2

    return U, S, V, threshold, w


def predict(data, U, threshold, w, digits):
    prediction_val = w.T @ U.T @ data
    predictions = np.zeros(prediction_val.shape)
    predictions[prediction_val < threshold] = digits[0]
    predictions[prediction_val >= threshold] = digits[1]
    return predictions

def accuracy(predictions, labels):
    return np.sum(predictions == labels) / len(predictions)


# %%
# Train classifier on all combinations of two digits
num_features = 20
train_acc = np.zeros((10, 10))
test_acc = np.zeros((10, 10))
for i in range(10):
    for j in range(i + 1, 10):
        print("Testing " + str(i) + " vs " + str(j))

        train1 = train_data[:, train_labels == i]
        train2 = train_data[:, train_labels == j]

        train = train_data[:, (train_labels == i) | (train_labels == j)]
        train_l = train_labels[(train_labels == i) | (train_labels == j)]

        test = test_data[:, (test_labels == i) | (test_labels == j)]
        test_l = test_labels[(test_labels == i) | (test_labels == j)]

        U, S, V, threshold, w = digit_trainer(train1, train2, num_features=num_fe
atures)
```

```
        train_predictions = predict(train, U, threshold, w, [i, j])
        train_acc[i, j] = accuracy(train_predictions, train_l)

        test_predictions = predict(test, U, threshold, w, [i, j])
        test_acc[i, j] = accuracy(test_predictions, test_l)


# %%
train_acc = np.ma.masked_array(train_acc, mask=(train_acc == 0))
test_acc = np.ma.masked_array(test_acc, mask=(test_acc == 0))


# %%
# Print Training Dataset Accuracy
np.set_printoptions(suppress=False, precision=4)
train_acc_sort = np.unravel_index(np.ma.argsort(train_acc, axis=None, fill_value=
10), train_acc.shape)
train_acc_sort = np.vstack(train_acc_sort).T[:45, :]
train_acc_val = np.zeros(train_acc_sort.shape[0])
for i in range(45):
    train_acc_val[i] = train_acc[train_acc_sort[i, 0], train_acc_sort[i, 1]]
print("Train Accuracies:")
print(np.hstack((train_acc_sort, train_acc_val.reshape(-1, 1))))


# %%
# Print Test Dataset Accuracy
test_acc_sort = np.unravel_index(np.ma.argsort(test_acc, axis=None, fill_value=10
), test_acc.shape)
test_acc_sort = np.vstack(test_acc_sort).T[:45, :]
test_acc_val = np.zeros(test_acc_sort.shape[0])
for i in range(45):
    test_acc_val[i] = test_acc[test_acc_sort[i, 0], test_acc_sort[i, 1]]
print("Test Accuracies:")
print(np.hstack((test_acc_sort, test_acc_val.reshape(-1, 1))))


# %%
# Generate accuracy matrices
train_acc_mat = np.array(train_acc) + np.array(train_acc.T)
for i in range(train_acc_mat.shape[0]):
    train_acc_mat[i, i] = 1
print(train_acc_mat)

test_acc_mat = np.array(train_acc) + np.array(train_acc.T)
for i in range(test_acc_mat.shape[0]):
    test_acc_mat[i, i] = 1
print(test_acc_mat)
```

```
# %%
# Display accuracy matrices
plt.imshow(train_acc_mat, cmap="Greys")
plt.title("Training Dataset Accuracy")
plt.xlabel("Digit 1")
plt.ylabel("Digit 2")
plt.colorbar()
plt.show()
plt.imshow(test_acc_mat, cmap="Greys")
plt.title("Test Dataset Accuracy")
plt.xlabel("Digit 1")
plt.ylabel("Digit 2")
plt.colorbar()
plt.show()


# %%
def digit_trainer_multiple(digit_data, labels, num_features):
    U, S, V = np.linalg.svd(digit_data, full_matrices=False)
    U = U[:, 0:num_features]

    SV = np.diag(S) @ V

    included_digits = np.unique(labels)
    num_digits = len(included_digits)

    digits = []
    digits_mean = []

    for i in range(num_digits):
        digits.append(SV[0:num_features, labels == included_digits[i]])
        digits_mean.append(np.mean(digits[i], axis=1))

    all_digits_sum = 0
    num_samples = 0
    for i in range(num_digits):
        all_digits_sum += np.sum(digits[i], axis=1)
        num_samples += digits[i].shape[1]

    all_digits_mean = all_digits_sum / num_samples

    S_w = 0

    for j in range(num_digits):
        for i in range(digits[j].shape[1]):
            S_w += (digits[j][:, i] - digits_mean[j]).reshape(-
1 ,1) @ (digits[j][:, i] - digits_mean[j]).reshape(-1, 1).T

    S_B = 0
```

```
    for j in range(num_digits):
        S_B += (digits_mean[j] - all_digits_mean).reshape(-
1, 1) @ (digits_mean[j] - all_digits_mean).reshape(-1, 1).T


    eigenvalues, eigenvectors = sp.linalg.eig(S_B, S_w)
    max_eig_ind = np.argmax(np.abs(eigenvalues))
    w = eigenvectors[:, max_eig_ind]
    w = w / np.linalg.norm(w, ord=2)

    projected_digits = []

    for j in range(num_digits):
        projected_digits.append(w.T @ digits[j])

    projected_digits_mean = []
    for j in range(num_digits):
        projected_digits_mean.append(np.mean(projected_digits[j]))

    return U, S, V, projected_digits_mean, w, included_digits

def predict_multiple(data, U, projected_digits_mean, w, digits):
    prediction_val = w.T @ U.T @ data
    predictions = np.zeros(prediction_val.shape)
    for i in range(len(prediction_val)):
        digit_index = np.argmin(np.abs(prediction_val[i] - projected_digits_mean)
)
        predictions[i] = digits[digit_index]

    return predictions

# Taket the subset of the dataset provided that contains only the digits found in
 the parameter digits
def data_subset(data, labels, digits):
    subset_data = []
    subset_labels = []
    for digit in digits:
        subset_data.append(data[:, labels == digit])
        subset_labels.append(labels[labels == digit])

    return np.hstack(subset_data), np.hstack(subset_labels)



# %%
data, labels = data_subset(train_data, train_labels, [0, 1])

U, S, V, projected_digits_mean, w, digits = digit_trainer_multiple(data, labels,
num_features=784)
```

```
predictions = predict_multiple(data, U, projected_digits_mean, w, digits)
print(accuracy(predictions, labels))

data, labels = data_subset(test_data, test_labels, digits)

predictions = predict_multiple(data, U, projected_digits_mean, w, digits)
print(accuracy(predictions, labels))


# %%
# Train SVM
from sklearn import svm
svm_model = svm.SVC()
svm_model.fit(train_data.T[:, :], train_labels[:])


# %%
# Test SVM
svm_model.score(test_data.T[:, :], test_labels[:])


# %%
# Train tree classifier
from sklearn import tree
tree_model = tree.DecisionTreeClassifier()
tree_model.fit(train_data.T[:, :], train_labels[:])


# %%
# Test tree classifier
tree_model.score(test_data.T[:, :], test_labels[:])


# %%
# Compare SVM performance on digits 3, 5
svm_hardest = svm.SVC()
svm_hardest.fit(train_data.T[(train_labels == 3) | (train_labels == 5), :], train
_labels[(train_labels == 3) | (train_labels == 5)])
svm_hardest.score(test_data.T[(test_labels == 3) | (test_labels == 5), :], test_l
abels[(test_labels == 3) | (test_labels == 5)])


# %%
# Compare tree performance on digits 3, 5
tree_hardest = tree.DecisionTreeClassifier()
tree_hardest.fit(train_data.T[(train_labels == 3) | (train_labels == 5), :], trai
n_labels[(train_labels == 3) | (train_labels == 5)])
```

```
tree_hardest.score(test_data.T[(test_labels == 3) | (test_labels == 5), :], test_
labels[(test_labels == 3) | (test_labels == 5)])


# %%
# Compare SVM performance on digits 0, 1
svm_easiest = svm.SVC()
svm_easiest.fit(train_data.T[(train_labels == 0) | (train_labels == 1), :], train
_labels[(train_labels == 0) | (train_labels == 1)])
svm_easiest.score(test_data.T[(test_labels == 0) | (test_labels == 1), :], test_l
abels[(test_labels == 0) | (test_labels == 1)])


# %%
# Compare tree performance on digits 0, 1
tree_easiest = tree.DecisionTreeClassifier()
tree_easiest.fit(train_data.T[(train_labels == 0) | (train_labels == 1), :], trai
n_labels[(train_labels == 0) | (train_labels == 1)])
tree_easiest.score(test_data.T[(test_labels == 0) | (test_labels == 1), :], test_
labels[(test_labels == 0) | (test_labels == 1)])
```