# SVMs, LDA, and Decision Trees in PCA Space to Recognize MNIST Digits

Alex Mallen

**Abstract**

In this paper I explore various standard methods in machine learning and test their performance on a benchmark task in the machine learning field: classifying images of handwritten digits. I also demonstrate that principle component analysis is a powerful dimensionality reduction step to obtain a representation of the images that is much more relevant to classification than the original pixel values.

## 1 Introduction

The Modified National Institute of Standards and Technology (MNIST) database is a long-standing benchmark and entry point for machine learning. It consists of 70,000 images of handwritten digits from American census bureau employees and high school students. The testing set and the training set were originally from different populations, so MNIST is modified such that the two sets come from the same population. They were also standardized to 8-bit grayscale $28 \times 28$ images [11]. In their original 1998 paper, LeCun and colleagues were able to achieve 99.2% classification accuracy across all 10 digits using a 9th degree polynomial kernel Support Vector Machine (SVM) [10].

The goal of this paper is to test various machine learning methods—Linear Discriminant Analysis (LDA), SVM, and decision trees—to classify various subsets of these digits and gain insights into the strengths and weaknesses of each method. The vast majority of the pixels in these images convey no information about what digit is being displayed, so Principle Component Analysis (PCA) is used as a preprocessing step to reduce the dimensionality from $28 \times 28 = 784$ to a manageable size where each component relays maximal information about which digit is present.

## 2 Theory

As described in my previous paper [2], principle component analysis projects a 0-mean dataset onto an orthogonal basis such that the first $n$ basis vector (known as principle components) are the $n$ vectors that explain the most variance of your dataset. This is a very useful tool for dimensionality reduction, and is the first step performed in this paper. The singular value decomposition of a mean-subtracted matrix $X_c$ is given by

$$X_c = U\Sigma V^T \tag{1}$$

$\Sigma$ is a diagonal matrix of singular values, whose squared entries indicate the variance of the data along each of the principle components in $U$. The columns of $U$ form an orthogonal basis for the entire dataset sorted by importance. The first column of $U$ is the most defining image of the dataset, and explains the most variance, as it is multiplied by the largest singular value. The columns of $V^T$ represent each of the 60,000 images in the orthogonal basis formed by the columns of $U\Sigma$.

Linear discriminant analysis takes a set of labeled data of $n$ spatial dimensions and projects it onto a line in $\mathbb{R}^n$ such that the distribution of points along that line has maximal statistical separation between classes [3].

SVMs are a supervised machine learning algorithm that have a similar objective to LDA: they attempt to linearly separate the data. SVMs, however, are only directly capable of two-class separation. The objective of an SVM given data in $\mathbb{R}^n$ of two distinct classes is to find an $(n-1)$-dimensional hyperplane that minimizes the number and magnitude of misclassifications while maximizing the distance between the hyperplane and the $n+1$ closest points [4].

A decision tree is a supervised learning algorithm that asks a series of binary questions about the coordinates of the data in $n$ dimensional space in order to best classify it. The first step partitions the space into two according to some hyperplane orthogonal to one of the original basis vectors. Each step thereafter recursively partitions one of the partitions produced in the previous step in order to best classify the data [5].

## 3 Implementation

I implemented all methods in a Python Jupyter Notebook, with extensive use of the scikit learn package. In order to download and extract the MNIST digits, I used the torchvision library. This is performed only once. I then used the MNIST library to load the train and test sets as arrays.

### Principle Component Projection

I computed the average image of the 60,000 training images, and subtracted it from each one to obtain the matrix $X_c$. I then calculated the SVD of this matrix as in equation 1. I called `numpy.svd` with the flag `full_matrices=False` because $X_c$ is highly non-square, and would therefore have produced very large matrices (totaling to 26 GB), when only a small fraction of the entries would be nonzero.

I defined the rank of the decision space to be the number of principle components required in order to explain 90% of the variance in the data. I also made sure that this rank

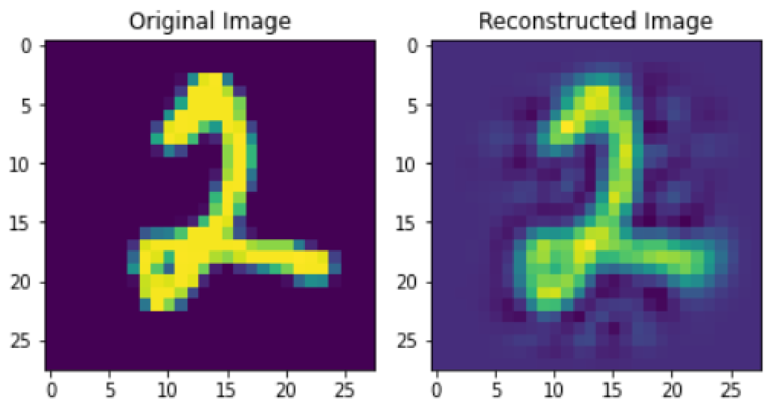

FIGURE 1: An example image reconstructed using the first 86 principle components.

was suitable by plotting the low-rank projection of each image for several random images, and assessing their quality, as in figure 1. In order to get the rank-$r$ projection of the training images, I computed the first $r$ rows of $\Sigma V^T$ instead of $U^T X_c$ because they are equivalent by 1 and the former is faster. For the 10,000 test images, however, I had to compute $U^T X_c$.

### Classification

Although I determined the rank of the data to be approximately 86, I determined that this was too many dimensions for classification, especially for a decision tree model. Instead, I used 10 dimensions, which accounts for 50% of the variance in my data.

I used the `sklearn.discriminant_analysis.LinearDiscriminantAnalysis` classifier class for linear discriminant classification, which is described in Appendix A. This can readily be used to distinguish between multiple classes.

For support vector machine classification, I used `sklearn.svm.SVC`. For multi-class classification, I initialized the classifier with the one-versus-one option, which is described in Appendix A.

For decision tree classification, I used `sklearn.tree.DecisionTreeClassifier`, which is described in Appendix A. I regularized this model with `min_samples_leaf=5`. Like LDA, this is readily usable to distinguish between multiple classes.

In order to find the easiest and hardest to distinguish digits for LDA, I looped over all $\binom{10}{2} = 45$ pairs of digits and trained an LDA classifier on all occurrences of those digits in the training set. I then evaluated the classifier's performance on all occurrences of those digits in the test set.

## 4 Results

In figure 2, it is apparent that the images of digits have significant structure in PCA space. While these first 3 principle components only account for 28% of the variance in the images,

| CLASSIFIER | 1 v 0 (LDA easy) | | 9 v 4 (LDA hard) | | 2 v 7 v 8 | | all | |
|---|---|---|---|---|---|---|---|---|
| | train | test | train | test | train | test | train | test |
| **LDA** | 0.0047 | 0.0019 | 0.155 | 0.159 | 0.070 | 0.073 | 0.233 | 0.227 |
| **linear SVM** | 0.0013 | 0.0014 | 0.048 | 0.054 | N/A | N/A | 0.057 | 0.064 |
| **decision tree** | 0.0019 | 0.0038 | 0.050 | 0.135 | N/A | N/A | 0.079 | 0.166 |

TABLE 1: Performance of each classifier on various tasks.

the first 86 principle components account for 90% of the variance. A complete description of the singular value spectrum and the explained variance can be seen in figure 3.

A 10-dimensional PCA-space was used for the classification. The results of these experiments are in table 1. The first two columns indicate which pairs of digits were the easiest and hardest to distinguish for the LDA classifier. A surprising result is that the classifier performed 2.5 times better on the test set than for the training set when comparing images of 1 and 0. This is likely due to randomness, since only 28/6000 images were misclassified in the training stage and only 2/1000 digits were misclassified in testing. Other than that, the results for LDA and SVM were extremely consistent between training and testing. A complete list of LDA's performances on pairs of digits can be found in the LDA section of the code in Appendix B.

It should be noted that direct comparison between classification methods on hardest and

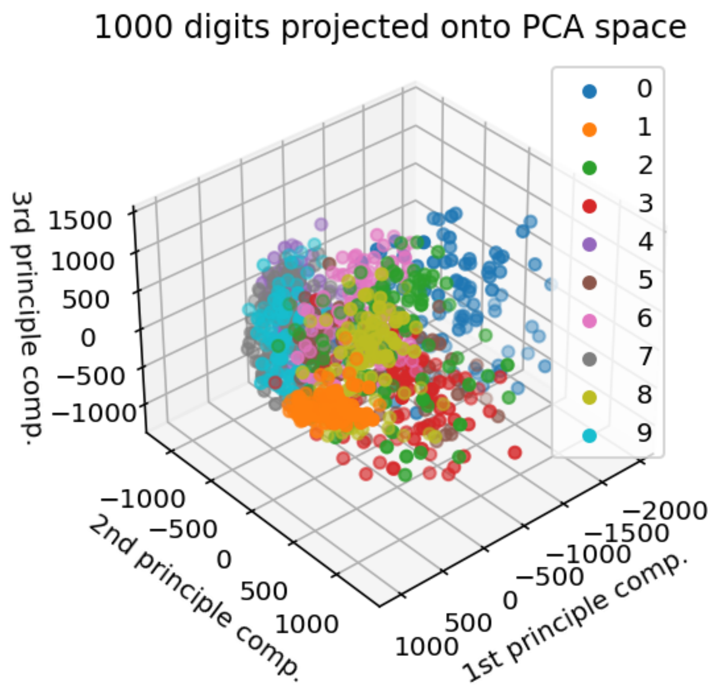

FIGURE 2: A projection of the mean-subtracted images onto the first 3 principle components. Only 1000 of the 60,000 points are shown for clarity.

easiest pairs of digits is misleading because these pairs of digits were selected based on the fact that LDA did well or poorly at classifying them. Even so, SVM performed best in both cases. While the decision tree outperformed LDA on the training sets in both cases, it did not show merit on the test set. This is a strong indication that decision trees overfit. More evidence of decision trees overfitting can be seen in the complete 10-digit classification. While the tree classifier had a competitive error rate of 7.9% in the training stage, it more than doubled that error rate to 16.6% on the test set. However, it still outperformed linear discriminant analysis (22.7%), likely because LDA reduces the dimensionality of the data down to 1, and is therefore not very powerful. The SVM classifier outperformed all others by a significant margin, with only a 6.4% error rate.

## 5 Conclusion

The 3 machine learning techniques explored in this paper are powerful but obsolete. LDAs and SVMs are both effective tools for distinguishing between two classes. In the case of 2 classes, LDA and SVM are doing essentially the same thing, but with a different objective for dividing the classification space. One reason that SVMs performed significantly better at multi-class separation is that the SVM classifier used 45 different SVMs, finding a unique hyperplane that best divides every pair of digits. On the other hand, multi-class LDA projects all of the data down onto 1 dimension, which effectively mandates that all the hyperplanes dividing the data are parallel. There is a tradeoff in that SVMs are slower. Decision trees proved to be more effective at multi-class clustering than LDAs but still not

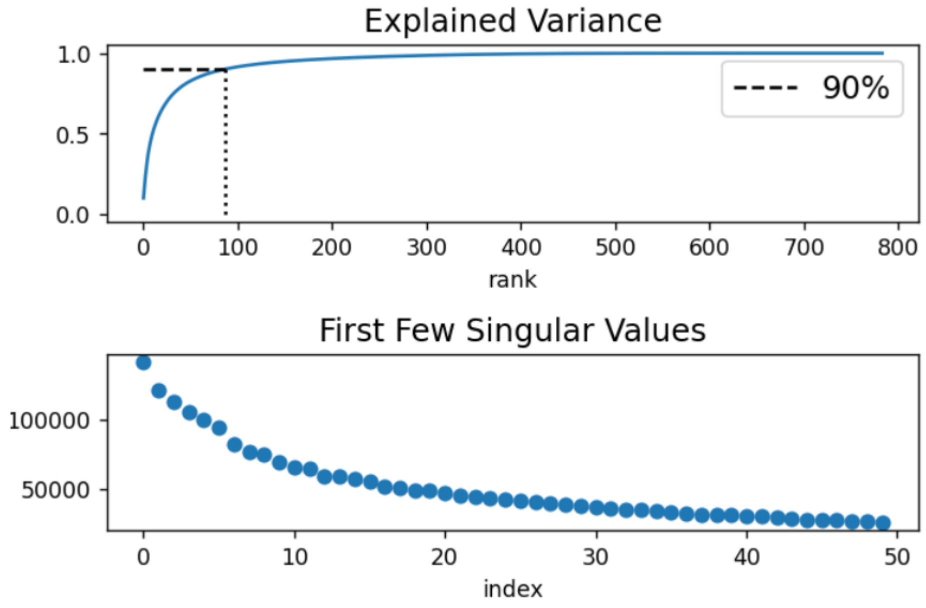

FIGURE 3: The singular value spectrum of the images. 90% of the variance can be explained by the first 86 principle components, roughly 10% of the original dimensionality of the images.

as effective as SVMs, but suffered severely from overfitting due to their ability to fit the training data perfectly with a highly non-smooth decision boundary.

The best performance at classifying all 10 digits was by the SVM model, with a 6.4% error rate. This still does not come close to the benchmark 0.8% set 20 years ago by LeCun et. al [10]. This indicates there is plenty of room for improvement from these simple models. Such improvements would likely come from abandoning the assumption of a linear or piecewise constant classification boundary. This can be achieved by performing a nonlinear pre-processing instead of PCA (which equates to using a nonlinear kernel in SVD) such as polynomial, or radial basis function. After applying linear classification in this higher dimensional space, a nonlinear decision boundary can be inferred in the original space. Neural networks are the leading candidate for such classification problems. While sophisticated neural networks have been able to achieve as low as 0.23% error rate, even a simple 3-layer neural network I created previously with 64 nodes in each hidden layer was able to achieve 4.9% error rate [10].

# Appendix A

## Important functions and their implementations

### `sklearn.svm.SVC`

This is a SVM classifier class that can be trained using `SVC.fit(train_data, train_labels)`, and then used to predict the labels of new data using `test_predics = SVC.predict(test_data)`. There are two options for using an SVM for multi-class classification, given by the option `decision_function_shape`. If "ovo" (one-versus-one is passed, SVC creates a unique support vector machine for every pair of classes. The default is "ovr" (one-versus-rest), which creates one support vector machine for each class. See [6].

### `sklearn.tree.DecisionTreeClassifier`

This is a decision tree classifier that can be trained using `DecisionTreeClassifier.fit (train_data, train_labels)`, and then used to predict the labels of new data using `test_predics = DecisionTreeClassifier.predict(test_data)`. Because decision trees are prone to overfitting, the parameters `min_samples_leaf` and `max_depth` can be used to regularize the model. See [7].

### `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`

This is a LDA classifier class that can be trained using `LinearDiscriminantAnalysis.fit (train_data, train_labels)`, and then used to predict the labels of new data using `test_predics = LinearDiscriminantAnalysis.predict(test_data)`. See [8].

# Appendix B

## Python code

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

# In[1]:


import numpy as np
from mnist import MNIST
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn import svm
from sklearn import tree

# I first used torchvision to download and extract the .gz compressed
# files, then used MNIST to read them as arrays. You can download and
    extract them however you like
# import torch
# import torchvision
# from torchvision import transforms, datasets


# In[2]:


mndata = MNIST('MNIST\\raw')

train_images, train_labels = mndata.load_training()
test_images, test_labels = mndata.load_testing()

train_images, train_labels = np.transpose(np.array(train_images)),
    np.array(train_labels)
test_images, test_labels = np.transpose(np.array(test_images)),
    np.array(test_labels)

avg_im = np.mean(train_images, axis=1, keepdims=True)
Xc = train_images - avg_im

U, S, Vh = np.linalg.svd(Xc, full_matrices=False) # economy otherwise
    requires 26 GB of ram

cumul_var = np.zeros(S.shape)
cumul_var[0] = S[0]**2
```

```
for i in range(1, S.shape[0]):
    cumul_var[i] = cumul_var[i-1] + S[i]**2

rank = 0
thresh = 0.9 * cumul_var[-1]
while cumul_var[rank] < thresh:
    rank += 1
print(rank)

plt.figure(dpi=125)
plt.subplot(211)
plt.plot(cumul_var / cumul_var[-1])
plt.plot([rank, rank], [0, 0.9], ":k")
plt.plot([0, rank], [0.9, 0.9], "--k", label="90%")
plt.xlabel("rank")
plt.title("Explained Variance", fontsize=14)
plt.legend(fontsize=14)
plt.subplot(212)
plt.scatter(np.arange(50), S[:50])
plt.title("First Few Singular Values", fontsize=14)
plt.xlabel("index")
plt.tight_layout()
plt.show()

U.shape, S.shape, Vh.shape

plt.subplots(2, 5, figsize=(20, 20))
for i in range(0, 10):
    plt.subplot(5, 2, i+1)
    temp = np.reshape((1000*U[:, i]).astype(int), (28, 28))
    temp[-1, -1] = 0
    plt.imshow(temp)
plt.colorbar()
plt.tight_layout()
plt.show()

idx = np.random.randint(0, train_images.shape[1])
rank = rank # 20 is enough to visually distinguish digits
re_S = np.zeros(len(S))
re_S[:rank] = S[:rank]
re_images = U @ np.diag(re_S) @ Vh
re_im = np.reshape((1000*(re_images[:, idx] + avg_im[:, 0])).astype(int),
    (28, 28))
og_im = np.reshape((1000*train_images[:, idx]).astype(int), (28, 28))
plt.subplots(1, 2)
plt.subplot(121)
plt.title("Original Image")
plt.imshow(og_im)
plt.subplot(122)
plt.title("Reconstructed Image")
plt.imshow(re_im)
plt.tight_layout()

# by the nature of the SVD, the first 3 rows of \Sigma V^T tell you how
    the images are projected onto the princple component
sample_size = 1000
proj_images = np.diag(S[:3]) @ Vh[:3, :sample_size]
print(proj_images.shape)

fig = plt.figure(dpi=120)
ax = fig.gca(projection='3d')
plt.title("1000 digits projected onto PCA space")
for digit in range(10):
    dig_ims = proj_images[:, np.nonzero(train_labels[:sample_size]==digit)]
    ax.scatter(dig_ims[0, :], dig_ims[1, :], dig_ims[2, :],
        label=str(digit))
ax.view_init(elev=15, azim=50)
plt.xlabel("1st principle comp.")
plt.ylabel("2nd principle comp.")
ax.set_zlabel("3rd principle comp.")
plt.legend()
plt.show()

# ########################################
# ###---Linear Discriminant Analysis---###
# ########################################

proj_size = 10
print(cumul_var[proj_size] / cumul_var[-1])
proj_train_ims = np.diag(S[:proj_size]) @ Vh[:proj_size, :]
avg_test = np.mean(test_images, axis=1, keepdims=True)
proj_test_ims = np.transpose(U[:, :proj_size]) @ (test_images - avg_test)
    # project test images onto the U modes found in training step

clf = LDA()

min_err = 1
max_err = 0
argmin_digs = None
argmax_digs = None
for dig1 in range(10):
    for dig2 in range(dig1):
        print(dig1, dig2)
        train_pair_ims = np.transpose(proj_train_ims[:,
            np.nonzero(np.logical_or(train_labels==dig1,
            train_labels==dig2))[0]])
        train_pair_labels =
            train_labels[np.nonzero(np.logical_or(train_labels==dig1,
            train_labels==dig2))[0]]
        clf.fit(train_pair_ims, train_pair_labels)

        test_pair_ims = np.transpose(proj_test_ims[:,
            np.nonzero(np.logical_or(test_labels==dig1,
            test_labels==dig2))[0]])
        test_pair_labels =
            test_labels[np.nonzero(np.logical_or(test_labels==dig1,
            test_labels==dig2))[0]]

        train_fit = clf.predict(train_pair_ims)
        train_err = np.sum((train_fit != train_pair_labels).astype(int)) /
            len(train_fit)
        print("Training Set Error:", train_err)
        predic = clf.predict(test_pair_ims)
        error = np.sum((predic != test_pair_labels).astype(int)) /
            len(predic)
        print("Test Set Error:", error, "\n")
        if min_err > error:
            min_err = error
            argmin_digs = (dig1, dig2)

        if max_err < error:
            max_err = error
            argmax_digs = (dig1, dig2)

print("Easiest to distinguish digits:", *argmin_digs, " with error",
    min_err)
print("Hardest to distinguish digits:", *argmax_digs, " with error",
    max_err)
# 1 0
# Training Set Error: 0.004658507698381366
# Test Set Error: 0.0018912529550827422

# 2 0
# Training Set Error: 0.03391970372864237
# Test Set Error: 0.0268389662027833

# 2 1
# Training Set Error: 0.02874015748031496
# Test Set Error: 0.03461005999077065
# 3 0
# Training Set Error: 0.03625352580056413
# Test Set Error: 0.021105527638190954

# 3 1
# Training Set Error: 0.025168958284782102
# Test Set Error: 0.02191142191142191

# 3 2
# Training Set Error: 0.051038133840681614
# Test Set Error: 0.042605288932419196

# 4 0
# Training Set Error: 0.009944751381215469
# Test Set Error: 0.011722731906218144

# 4 1
# Training Set Error: 0.010251112523839796
# Test Set Error: 0.007085498346717053

# 4 2
# Training Set Error: 0.036016949152542374
# Test Set Error: 0.02830188679245283

# 4 3
# Training Set Error: 0.021715526601520086
# Test Set Error: 0.015060240963855422

# 5 0
# Training Set Error: 0.06170662905500705
# Test Set Error: 0.057692307692307696

# 5 1
# Training Set Error: 0.019649757461152675
# Test Set Error: 0.010853478046373951

# 5 2
# Training Set Error: 0.0466649090429739
# Test Set Error: 0.04573804573804574

# 5 3
# Training Set Error: 0.1109764542936288
# Test Set Error: 0.09989484752891693

# 5 4
# Training Set Error: 0.03436029477048744
# Test Set Error: 0.029882604055496264
# 6 0
# Training Set Error: 0.029389409678236635
# Test Set Error: 0.037667698658410735

# 6 1
# Training Set Error: 0.02022116903633491
# Test Set Error: 0.018633540372670808

# 6 2
# Training Set Error: 0.07805658470865612
# Test Set Error: 0.07035175879396985

# 6 3
# Training Set Error: 0.018258776661963648
# Test Set Error: 0.01676829268292683

# 6 4
# Training Set Error: 0.019302721088435375
# Test Set Error: 0.01907216494845361

# 6 5
# Training Set Error: 0.044977511244377814
# Test Set Error: 0.051351351351351354

# 7 0
# Training Set Error: 0.013619954053167049
# Test Set Error: 0.01195219123505976

# 7 1
# Training Set Error: 0.019835473206734833
# Test Set Error: 0.02912621359223301

# 7 2
# Training Set Error: 0.03313425509285773
# Test Set Error: 0.0383495145631068

# 7 3
# Training Set Error: 0.029444982252339463
# Test Set Error: 0.03434739941118744

# 7 4
# Training Set Error: 0.036672999091434705
# Test Set Error: 0.03482587064676617

# 7 5
# Training Set Error: 0.029779223001882595
# Test Set Error: 0.0265625

# 7 6
# Training Set Error: 0.006320282360666503
# Test Set Error: 0.013595166163141994

# 8 0
# Training Set Error: 0.023016816714795312
# Test Set Error: 0.016888433981576252

# 8 1
# Training Set Error: 0.051377749543397126
# Test Set Error: 0.03698435277382646

# 8 2
# Training Set Error: 0.05385722753831823
# Test Set Error: 0.050348953140578266

# 8 3
# Training Set Error: 0.09380737773326657
# Test Set Error: 0.08669354838709678

# 8 4
# Training Set Error: 0.02274865304028051
# Test Set Error: 0.025051124744376277

# 8 5
# Training Set Error: 0.08339247693399574
# Test Set Error: 0.08092175777063237

# 8 6
# Training Set Error: 0.0225167813747982
# Test Set Error: 0.024327122153209108

# 8 7
# Training Set Error: 0.030868273357543743
# Test Set Error: 0.038461538461538464

# 9 0
# Training Set Error: 0.019036388140161724
# Test Set Error: 0.019105077928607342

# 9 1
# Training Set Error: 0.016625955401465605
# Test Set Error: 0.012126865671641791

# 9 2
# Training Set Error: 0.03334173175443017
# Test Set Error: 0.030867221950024497

# 9 3
# Training Set Error: 0.04461920529801325
# Test Set Error: 0.037642397226349676

# 9 4
# Training Set Error: 0.15511831057586295
# Test Set Error: 0.1592164741336012

# 9 5
# Training Set Error: 0.04722955145118733
# Test Set Error: 0.04418726985796949

# 9 6
# Training Set Error: 0.010027808207634617
# Test Set Error: 0.012709710218607015

# 9 7
# Training Set Error: 0.09644670050761421
# Test Set Error: 0.09081983308787432

# 9 8
# Training Set Error: 0.05406779661016949
# Test Set Error: 0.05849722642460918

# Easiest to distinguish digits: 1 0 with error 0.0018912529550827422
# Hardest to distinguish digits: 9 4 with error 0.1592164741336012


# ### 3 digit classification

dig1 = 2
dig2 = 7
dig3 = 8

train_trip_ims = np.transpose(proj_train_ims[:,
   np.nonzero(np.logical_or.reduce((train_labels==dig1,
   train_labels==dig2, train_labels==dig3)))[0]])
train_trip_labels =
   train_labels[np.nonzero(np.logical_or.reduce((train_labels==dig1,
   train_labels==dig2, train_labels==dig3)))[0]]
clf.fit(train_trip_ims, train_trip_labels)

test_trip_ims = np.transpose(proj_test_ims[:,
   np.nonzero(np.logical_or.reduce((test_labels==dig1, test_labels==dig2,
    test_labels==dig3)))[0]])
test_trip_labels =
    test_labels[np.nonzero(np.logical_or.reduce((test_labels==dig1,
    test_labels==dig2, test_labels==dig3)))[0]]

train_fit = clf.predict(train_trip_ims)
train_err = np.sum((train_fit != train_trip_labels).astype(int)) /
    len(train_fit)
print("Training Set Error:", train_err)
predic = clf.predict(test_trip_ims)
error = np.sum((predic != test_trip_labels).astype(int)) / len(predic)
print("Test Set Error:", error)


# ################################
# ###---LDA on all 10 digits---###
# ################################
clf.fit(np.transpose(proj_train_ims), train_labels)

train_fit = clf.predict(np.transpose(proj_train_ims))
train_err = np.sum((train_fit != train_labels).astype(int)) /
    len(train_fit)
print("Training Set Error:", train_err)
predic = clf.predict(np.transpose(proj_test_ims))
error = np.sum((predic != test_labels).astype(int)) / len(predic)
print("Test Set Error:", error)


# ## Support Vector Machine on all 10 digits

svmclf = svm.SVC(decision_function_shape='ovr') # one versus one
svmclf.fit(np.transpose(proj_train_ims), train_labels)

train_fit = svmclf.predict(np.transpose(proj_train_ims))
train_err = np.sum((train_fit != train_labels).astype(int)) /
    len(train_fit)
print("Training Set Error:", train_err)
predic = svmclf.predict(np.transpose(proj_test_ims))
error = np.sum((predic != test_labels).astype(int)) / len(predic)
print("Test Set Error:", error)

# ## Decision Tree on all 10 digits

treeclf = tree.DecisionTreeClassifier(min_samples_leaf=5) # regularize
treeclf.fit(np.transpose(proj_train_ims), train_labels)

train_fit = treeclf.predict(np.transpose(proj_train_ims))
```

```
train_err = np.sum((train_fit != train_labels).astype(int)) /
    len(train_fit)
print("Training Set Error:", train_err)
predic = treeclf.predict(np.transpose(proj_test_ims))
error = np.sum((predic != test_labels).astype(int)) / len(predic)
print("Test Set Error:", error)


# ## Comparison on the easiest and hardest pairs of digits
# ### Easiest

dig1 = argmin_digs[0]
dig2 = argmin_digs[1]
train_pair_ims = np.transpose(proj_train_ims[:,
    np.nonzero(np.logical_or(train_labels==dig1, train_labels==dig2))[0]])
train_pair_labels =
    train_labels[np.nonzero(np.logical_or(train_labels==dig1,
    train_labels==dig2))[0]]
test_pair_ims = np.transpose(proj_test_ims[:,
    np.nonzero(np.logical_or(test_labels==dig1, test_labels==dig2))[0]])
test_pair_labels = test_labels[np.nonzero(np.logical_or(test_labels==dig1,
    test_labels==dig2))[0]]

# ### SVM Classification

svmclf = svm.SVC()
svmclf.fit(train_pair_ims, train_pair_labels)

train_fit = svmclf.predict(train_pair_ims)
train_err = np.sum((train_fit != train_pair_labels).astype(int)) /
    len(train_fit)
print("Training Set Error:", train_err)
predic = svmclf.predict(test_pair_ims)
error = np.sum((predic != test_pair_labels).astype(int)) / len(predic)
print("Test Set Error:", error, "\n")

# ### Decision Tree Classification

treeclf = tree.DecisionTreeClassifier(min_samples_leaf=5) # regularize
treeclf.fit(train_pair_ims, train_pair_labels)

train_fit = treeclf.predict(train_pair_ims)
train_err = np.sum((train_fit != train_pair_labels).astype(int)) /
    len(train_fit)
print("Training Set Error:", train_err)
predic = treeclf.predict(test_pair_ims)
error = np.sum((predic != test_pair_labels).astype(int)) / len(predic)
```

```
print("Test Set Error:", error, "\n")

# ### Hardest

dig1 = argmax_digs[0]
dig2 = argmax_digs[1]
train_pair_ims = np.transpose(proj_train_ims[:,
    np.nonzero(np.logical_or(train_labels==dig1, train_labels==dig2))[0]])
train_pair_labels =
    train_labels[np.nonzero(np.logical_or(train_labels==dig1,
    train_labels==dig2))[0]]
test_pair_ims = np.transpose(proj_test_ims[:,
    np.nonzero(np.logical_or(test_labels==dig1, test_labels==dig2))[0]])
test_pair_labels = test_labels[np.nonzero(np.logical_or(test_labels==dig1,
    test_labels==dig2))[0]]

# ### SVM Classification

svmclf = svm.SVC()
svmclf.fit(train_pair_ims, train_pair_labels)

train_fit = svmclf.predict(train_pair_ims)
train_err = np.sum((train_fit != train_pair_labels).astype(int)) /
    len(train_fit)
print("Training Set Error:", train_err)
predic = svmclf.predict(test_pair_ims)
error = np.sum((predic != test_pair_labels).astype(int)) / len(predic)
print("Test Set Error:", error, "\n")

# ### Decision Tree Classification

treeclf = tree.DecisionTreeClassifier(min_samples_leaf=5) # regularize
treeclf.fit(train_pair_ims, train_pair_labels)

train_fit = treeclf.predict(train_pair_ims)
train_err = np.sum((train_fit != train_pair_labels).astype(int)) /
    len(train_fit)
print("Training Set Error:", train_err)
predic = treeclf.predict(test_pair_ims)
error = np.sum((predic != test_pair_labels).astype(int)) / len(predic)
print("Test Set Error:", error, "\n")
```

## References

[1] Mallen, Alex Troy. GitHub source code.
`https://github.com/AlexTMallen/AMATH-582/tree/master/HW4`

[2] Mallen, Alex Troy. Dimensionality Reduction of Oscillator Dynamics.
`https://github.com/AlexTMallen/AMATH-582/tree/master/HW3`

[3] Kutz, Jose Nathan. Data Analysis: Clustering and Classification (Lec. 2, part 4)
`https://www.youtube.com/watch?v=hakvxv3XRAs`

[4] Kutz, Jose Nathan. Data Analysis: Clustering and Classification (Lec. 3, part 1)
`https://www.youtube.com/watch?v=YWTzFPZgYCA`

[5] Kutz, Jose Nathan. Data Analysis: Clustering and Classification (Lec. 3, part 2)
`https://www.youtube.com/watch?v=6kjDqumvXG8`

[6] Scikit learn SVM.
`https://scikit-learn.org/stable/modules/svm.html`

[7] Scikit learn Decision Tree.
`https://scikit-learn.org/stable/modules/tree.html`

[8] Scikit learn LDA.
`https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.`

[9] Scikit-learn: Machine Learning in Python. Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E. Journal of Machine Learning Research, 12, 2825–2830, 2011

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.

[11] MNIST Wikipedia. MNIST Database.
`https://en.wikipedia.org/wiki/MNIST_database`