"Introduction to Applied Linear Algebra" book, page 79-80, 288-290:

The MNIST (Mixed National Institute of Standards) database of handwritten digits is a data set containing $N = 60000$ grayscale images of size 28×28, which we represent as n-vectors with $n = 28 \times 28 = 784$. Figure 4.6 shows a few examples from the data set. (The data set is available from Yann LeCun at `yann.lecun.com/exdb/mnist`.)

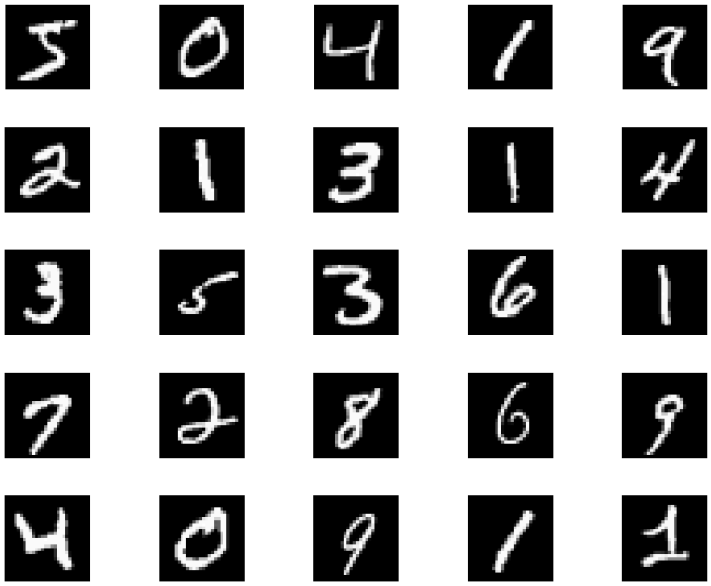


Figure 4.6 25 images of handwritten digits from the MNIST data set. Each image has size 28×28, and can be represented by a 784-vector.

http://yann.lecun.com/exdb/mnist/

THE MNIST DATABASE

of handwritten digits

Yann LeCun, Courant Institute, NYU
Corinna Cortes, Google Labs, New York
Christopher J.C. Burges, Microsoft Research, Redmond

The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.

It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

Four files are available on this site:

```
train-images-idx3-ubyte.gz:  training set images (9912422 bytes)
train-labels-idx1-ubyte.gz:  training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz:   test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz:   test set labels (4542 bytes)
```

14.2 Least squares classifier

Many sophisticated methods have been developed for constructing a Boolean model or classifier from a data set. *Logistic regression* and *support vector machine* are two methods that are widely used, but beyond the scope of this book. Here we discuss a very simple method, based on least squares, that can work quite well, though not as well as the more sophisticated methods.

We first carry out ordinary real-valued least squares fitting of the outcome, ignoring for the moment that the outcome y takes on only the values -1 and $+1$. We choose basis functions $f_1, \ldots, f_p$, and then choose the parameters $\theta_1, \ldots, \theta_p$ so as to minimize the sum squared error

$$(y^{(1)} - \tilde{f}(x^{(1)}))^2 + \cdots + (y^{(N)} - \tilde{f}(x^{(N)}))^2,$$

where $\tilde{f}(x) = \theta_1 f_1(x) + \cdots + \theta_p f_p(x)$. We use the notation $\tilde{f}$, since this function is not our final model $\hat{f}$. The function $\tilde{f}$ is the least squares fit over our data set, and $\tilde{f}(x)$, for a general vector x, is a number.

Our final classifier is then taken to be

$$\hat{f}(x) = \mathbf{sign}(\tilde{f}(x)), \tag{14.1}$$

where $\mathbf{sign}(a) = +1$ for $a \geq 0$ and -1 for $a < 0$. We call this classifier the *least squares classifier*.

The intuition behind the least squares classifier is simple. The value $\tilde{f}(x)$ is a number, which (ideally) is near $+1$ when $y^{(i)} = +1$, and near -1 when $y^{(i)} = -1$. If we are forced to guess one of the two possible outcomes $+1$ or -1, it is natural to choose $\mathbf{sign}(\tilde{f}(x))$. (Indeed, $\mathbf{sign}(\tilde{f}(x))$ is the nearest neighbor of $\tilde{f}(x)$ among the points -1 and $+1$.) Intuition suggests that the number $\tilde{f}(x)$ can be related to our confidence in our guess $\hat{y} = \mathbf{sign}(\tilde{f}(x))$: When $\tilde{f}(x)$ is near 1 we have confidence in our guess $\hat{y} = +1$; when it is small and negative (say, $\tilde{f}(x) = -0.03$), we guess $\hat{y} = -1$, but our confidence in the guess will be low. We won't pursue this idea further in this book, except in multi-class classifiers, which we discuss in §14.3.

The least squares classifier is often used with a regression model, *i.e.*, $\tilde{f}(x) = x^T\beta + v$, in which case the classifier has the form

$$\hat{f}(x) = \mathbf{sign}(x^T\beta + v). \tag{14.2}$$

We can easily interpret the coefficients in this model. For example, if β_7 is negative, it means that the larger the value of x_7 is, the more likely we are to guess $\hat{y} = -1$. If β_4 is the coefficient with the largest magnitude, then we can say that x_4 is the feature that contributes the most to our classification decision.

14.2.2 Handwritten digit classification

We now consider a much larger example, the MNIST data set described in §4.4.1. The (training) data set contains 60000 images of size 28 by 28. (A few samples are shown in figure 4.6.) The number of examples per digit varies between 5421 (for digit five) and 6742 (for digit one). The pixel intensities are scaled to lie between 0 and 1. We remove the pixels that are nonzero in fewer than 600 training examples. The remaining 493 pixels are shown as the white area in figure 14.1. There is also a separate test set containing 10000 images. Here we will consider classifiers to distinguish the digit zero from the other nine digits.

In this first experiment, we use the 493 pixel intensities, plus an additional feature with value 1, as the $n = 494$ features in the least squares classifier (14.1).

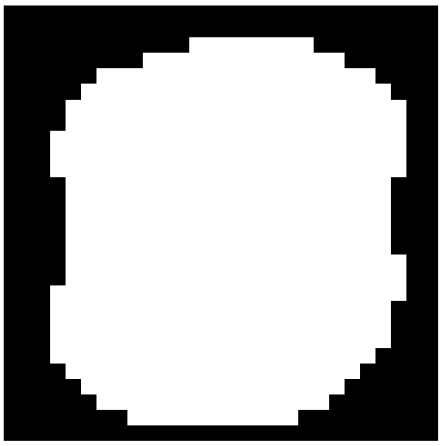


Figure 14.1 Location of the pixels used as features in the handwritten digit classification example.

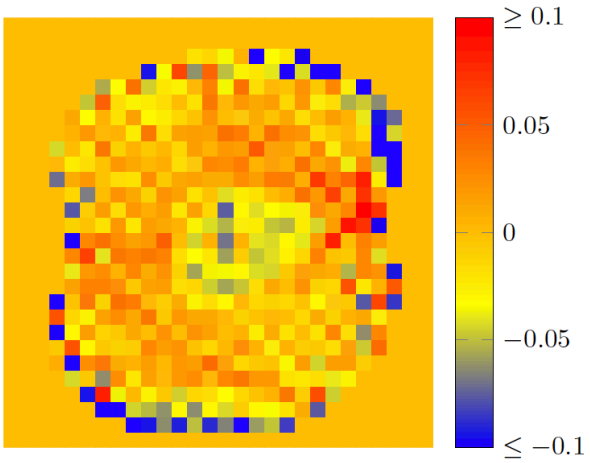


Figure 14.3 The coefficients β_k in the least squares classifier that distinguishes the digit zero from the other nine digits.

Part 1: Find a least-squares binary classifier for handwritten *mnist* digit set, i.e. determine if an image x is a digit k or not digit k. Use label $y_i = 1$ if x_i is digit k and $y_i = -1$ otherwise. Find β, and v , such that $y_i \approx \hat{y}_i = \mathbf{sign}\,(\beta^T x_i + v) = \mathbf{sign}(\tilde{y}_i)$

Note: The real-valued offset v in the linear expression $\beta^T x_i + v$ can be "folded" into vector "β". In order to do that, think of $\beta^T x_i + v = \beta^T x_i + v * 1 = [\beta_1\ \beta_2\ \ldots\ \beta_M\ v] \cdot [x_i^1\ x_i^2\ \ldots\ x_i^M\ 1]^T$, where x_i'th image is written as a 1-D column vector of length $M = 28^2$: $x_i = [x_i^1\ x_i^2\ \ldots\ x_i^M\ 1]^T$.

Count the number of correctly identified hand-written digits using your classifier. You can use either the same "training" set, or "test" set.

Part 2: Modify the LS setting in Part 1 to include the regularization parameter:

$$\sum_{i=1}^{N} (\tilde{y}_i - y_i)^2 + \lambda \|\beta\|^2$$

Display the image corresponding to β for different values of λ. Do you see any change?
Count the number of correctly identified hand-written digits using your classifier. Is there any change of that count as you vary λ? How does that compare with result in Part 1?

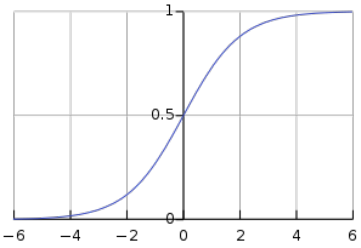
Part 3: Compute 10 largest eigenvalues and corresponding eigenvectors of covariance matrix for the selected digit k of the "train" data set. Display the eigenvectors as 28x28 images.

Part 4: Extend the linear formulation of Least Squares in Part 1 to a non-linear LS:

$$\sum_{i=1}^{N} (\tilde{y}_i - y_i)^2,$$

where $\tilde{y}_i = \beta^T \varphi(x_i) + v$ (instead of $\tilde{y}_i = \beta^T x_i + v$ as in Part 1) and $\varphi(x)$ being a Logistic function:

$$\varphi(x) = \frac{1}{1 + e^{-k(x - x_0)}},$$



Where parameter k represents the rate of steepness of the logistic curve and x_0 corresponds to the midpoint of the sigmoid $\varphi(x)$ (c.f. figure above with $x_0 = 0$ and $k = 1$).
The logistic function $\varphi(x)$ replaces the **sign**(x) used in linear formulation and since the range of $\varphi(x)$ is [0,1] (vs. [-1,1] for **sign**), the labels should be: $y_i = 1$ if x_i is digit k and $y_i = 0$ otherwise (vs. 1 and -1 used in linear case).

You can use ‘least_squares’ function available in scipy.optimize (with ‘lm’ option for Levenberg-Marquardt algorithm) to find parameters β. The rate k of the logistic function needs to be hand-picked though. Experiment with different values for k.

Compare with results in Part 1.

Also, experiment with adding a normalization, as in Part 2, penalizing large norms of β:

$\sum_{i=1}^{N}(\tilde{y}_i - y_i)^2 + \lambda\|\beta\|^2$, and compare with results in Part 2.

In [1]:
```
ls mnist/
```

```
t10k-images-idx3-ubyte.gz   train-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz   train-labels-idx1-ubyte.gz
```

In [2]:
```
import struct
import gzip
from pathlib import Path
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

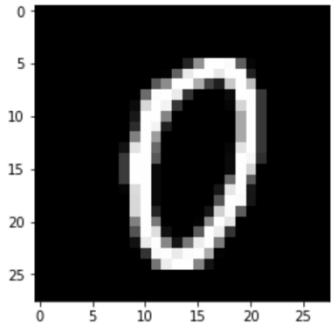
In [3]:
```
# create dictionaries to store the data
train = dict()
test = dict()
```

In [4]:
```
def get_images(filename):
    with gzip.GzipFile(Path('mnist', filename), 'rb') as f:
        magic, size, rows, cols = struct.unpack(">IIII", f.read(16))
        print(magic, size, rows, cols)
        images = np.frombuffer(f.read(), dtype=np.dtype('B'))
    return images.reshape(size, rows,cols)
```

In [5]:
```
train['image'] = get_images('train-images-idx3-ubyte.gz')
test['image'] = get_images('t10k-images-idx3-ubyte.gz')
print(train['image'].shape, test['image'].shape)
```

```
2051 60000 28 28
2051 10000 28 28
(60000, 28, 28) (10000, 28, 28)
```

In [6]:
```
#sanity check that it's loaded in properly
fig, ax = plt.subplots()
_ = ax.imshow(train['image'][1000], cmap='gray')
```



In [7]:
```
def get_labels(filename):
    with gzip.GzipFile(Path('mnist', filename), 'rb') as f:
        magic, num = struct.unpack(">II", f.read(8))
        labels = np.frombuffer(f.read(), dtype=np.dtype('B'))
    return labels
```

In [8]:
```
train['label'] = get_labels('train-labels-idx1-ubyte.gz')
test['label'] = get_labels('t10k-labels-idx1-ubyte.gz')
print(train['label'].shape, test['label'].shape)
```

```
(60000,) (10000,)
```

In [9]:
```
#check that labels are between expected values
np.unique(train['label'])
```

Out[9]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
```

Material for **Part 3:**

https://www.academia.edu/16125627/Neural_and_statistical_classifiers-taxonomy_and_two_case_studies

Neural and Statistical Classifiers—Taxonomy and Two Case Studies

Lasse Holmström, *Member, IEEE*, Petri Koistinen, Jorma Laaksonen,
Student Member, IEEE, and Erkki Oja, *Senior Member, IEEE*

12

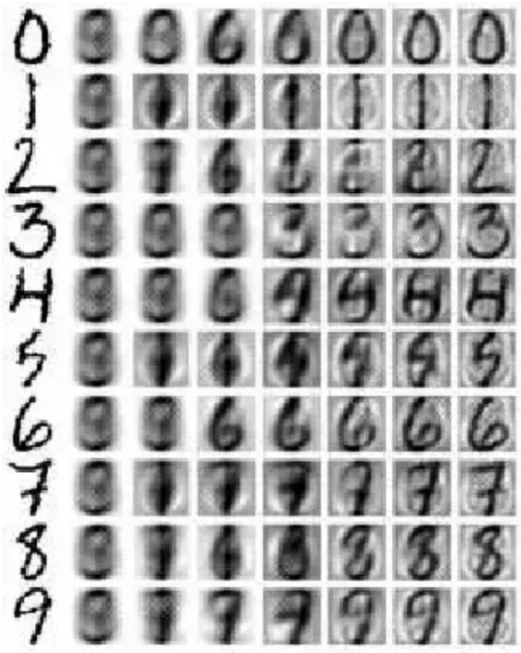


Fig. 2. In the leftmost column, some normalized 32 × 32 digit images. In the second column, mean of the training set and in the remaining columns, reproductions by adding projections in the principal directions up to dimension 1, 2, 5, 16, 32, and 64.

A. Data, Preprocessing, and Feature Extraction

The first case involved handwritten digit data specifically assembled for the present study. To obtain the test material, a set of forms was filled out by randomly chosen Finnish people. The total number of forms was 894 and each contained two handwritten examples of each of the ten digits. The images were scanned in binary form with the resolution of 300 pixels per inch in both directions. In this resolution each digit was printed approximately in a 75 × 100 pixel rectangle. The bounding boxes of the digits were normalized to 32 × 32 pixels and the slant of the images was removed. No normalization of the line width was performed. The preprocessing procedure closely resembles that of [20], [91], and it produced a sample of 17 880 binary vectors, each 1024-dimensional. Some examples are displayed in the leftmost column of Fig. 2. The sample was then divided into two sets of equal size, one for training and one for testing the classifiers.

As pattern vectors we used Karhunen–Loève (KL) transforms [6] of the original preprocessed images derived from the estimated covariance matrix of the training set. This approach is similar to that adopted in [20] and [91]. In general, the magnitude of the covariance matrix eigenvalues can be used to measure the usefulness of the corresponding KL transform components in classification but, for the digit data, the eigenvalues decreased rather steadily in magnitude and it was not possible to decide on any natural reduced pattern vector dimension. On the other hand, higher dimensions imply more computations both in classifier design and application. In our tests we used a maximum pattern dimension of 64. Cross-validation (see below) can then be used to select the best dimension $d \leq 64$ for each classifier separately. The columns of Fig. 2 show how the normalized images can be reproduced from their KL transforms with progressively greater accuracy as the transform dimension is increased.

https://towardsdatascience.com/principal-component-analysis-your-tutorial-and-code-9719d3d3f376

Principal Component Analysis: Your Tutorial and Code

George Seif Follow

Nov 9, 2018 · 5 min read ★

Principal Component Analysis (PCA) is a simple yet powerful technique used for dimensionality reduction. Through it, we can directly decrease the number of feature variables, thereby narrowing down the important features and saving on computations. From a high-level view PCA has three main steps:

(1) Compute the covariance matrix of the data

(2) Compute the eigen values and vectors of this covariance matrix

(3) Use the eigen values and vectors to select only the most important feature vectors and then transform your data onto those vectors for reduced dimensionality!

The entire process is illustrated in the figure above, where our data has been converted from a 3-dimensional space of 1000 points to a 2-dimensional space of 100 points. That's a 10X saving on computation!

(1) Computing the covariance matrix

PCA yields a feature subspace that maximizes the variance along the feature vectors. Therefore, in order to properly measure the variance of those feature vectors, they must be properly balanced. To accomplish this, we first normalise our data to have zero-mean and unit-variance such that each feature will be weighted equally in our calculations. Assuming that our dataset is called *X*:

```
1  from sklearn.preprocessing import StandardScaler
2  X = StandardScaler().fit_transform(X)
```

normalise.py hosted with ❤ by GitHub — view raw

The covariance of two variables measures how "correlated" they are. If the two variables have a positive covariance, then one when variable increases so does the other; with a negative covariance the values of the feature variables will change in opposite directions. The covariance matrix is then just an array where each value specifies the covariance between two feature variables based on the x-y position in the matrix. The formula is:

$$\Sigma = \frac{1}{n-1}\left((\mathbf{X} - \bar{\mathbf{x}})^T\,(\mathbf{X} - \bar{\mathbf{x}})\right)$$

Where the *x* with the line on top is a vector of mean values for each feature of X. Notice that when we multiply a transposed matrix by the original one we end up multiplying each of the features for each data point together! In numpy code it looks like this:

```python
import numpy as np

# Compute the mean of the data
mean_vec = np.mean(X, axis=0)

# Compute the covariance matrix
cov_mat = (X - mean_vec).T.dot((X - mean_vec)) / (X.shape[0]-1)


# OR we can do this with one line of numpy:
cov_mat = np.cov(X.T)
```

compute_cov_mat.py hosted with ❤ by GitHub

(2) Computing Eigen Values and Vectors

The eigen vectors (principal components) of our covariance matrix represent the vector directions of the new feature space and the eigen values represent the magnitudes of those vectors. Since we are looking at our *covariance matrix* the eigen values *quantify* the contributing variance of each vector.

If an eigen vector has a corresponding eigen value of high magnitude it means that our data has high variance along that vector in feature space. Thus, this vector holds a lot information about our data, since any movement along that vector causes large “variance”. On the other hand vectors with small eigen values have low variance and thus our data does not vary greatly when moving along that vector. Since nothing changes when moving along that particular feature vector i.e changing the value of that feature vector does not greatly effect our data, then we can say that this feature isn’t very important and we can afford to remove it.

That’s the whole essence of eigen values and vectors within PCA. Find the vectors that are the most important in representing our data and discard the rest. Computing the eigen vectors and values of our covariance matrix is an easy one-liner in numpy. After that, we’ll sort the eigen vectors in descending order based on their eigen values.

```python
# Compute the eigen values and vectors using numpy
eig_vals, eig_vecs = np.linalg.eig(cov_mat)

# Make a list of (eigenvalue, eigenvector) tuples
eig_pairs = [(np.abs(eig_vals[i]), eig_vecs[:,i]) for i in range(len(eig_vals))]

# Sort the (eigenvalue, eigenvector) tuples from high to low
eig_pairs.sort(key=lambda x: x[0], reverse=True)
```

eigen_pca.py hosted with ❤ by GitHub view raw

(3) Projection onto new vectors

At this point we have a list of eigen vectors sorted in order of "importance" to our dataset based on their eigen values. Now what we want to do is select the most important feature vectors that we need and discard the rest. We can do this in a clever way by looking at the *explained variance percentage* of the vectors. This percentage quantifies how much information (variance) can be attributed to each of the principal components out of the total 100%.

Let's take an example to illustrate. Say we have a dataset which originally has 10 feature vectors. After computing the covariance matrix, we discover that the eigen values are:

[12, 10, 8, 7, 5, 1, 0.1, 0.03, 0.005, 0.0009]

The total sum of this array is = 43.1359. But the first **6 values** represent:

42 / 43.1359 = 99.68% of the total! That means that our first 5 eigen vectors effectively hold 99.68% of the *variance* or *information* about our dataset. We can thus discard the last 4 feature vectors as they only contain 0.32% of the information, a worthy sacrifice for saving on 40% of the computations!

Therefore, we can simply define a threshold upon which we can decide whether to keep or discard each feature vector. In the code below, we simply count the number of feature vectors we would like to keep based on a selected threshold of 97%.

```
# Only keep a certain number of eigen vectors based on
# the "explained variance percentage" which tells us how
# much information (variance) can be attributed to each
# of the principal components

exp_var_percentage = 0.97 # Threshold of 97% explained variance

tot = sum(eig_vals)
var_exp = [(i / tot)*100 for i in sorted(eig_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)

num_vec_to_keep = 0

for index, percentage in enumerate(cum_var_exp):
  if percentage > exp_var_percentage:
    num_vec_to_keep = index + 1
    break
```

vec_proj_pca.py hosted with ❤️ by GitHub

The final step is to actually project our data onto the vectors we decided to keep. We do this by building a *projection matrix*: that's just a fancy word for a matrix we will multiply by to project our data onto the new vectors. To create it, we simply concatenate all of the eigen vectors we decided to keep. Our final step is to simply take the dot product between our original data and our projection matrix.

```python
# Compute the projection matrix based on the top eigen vectors
num_features = X.shape[1]
proj_mat = eig_pairs[0][1].reshape(num_features,1)
for eig_vec_idx in range(1, num_vec_to_keep):
  proj_mat = np.hstack((proj_mat, eig_pairs[eig_vec_idx][1].reshape(num_features,1)))

# Project the data
pca_data = X.dot(proj_mat)
```

vec_proj_pca_2.py hosted with ❤ by GitHub view raw