

Principal Component Analysis (PCA)

In this notebook, you shall implement PCA and then apply it (once again) to the MNIST digit dataset.

Learning objective

1. Write code that implements PCA.
2. Write code that implements PCA for high-dimensional datasets

Let's first import the packages we need for this week.

```
In [79]: # PACKAGE: DO NOT EDIT
import numpy as np
import scipy
import scipy.stats
```

```
In [80]: import matplotlib.pyplot as plt
from ipywidgets import interact

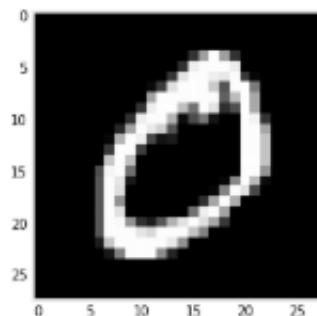
plt.style.use('fivethirtyeight')
%matplotlib inline
```

```
In [81]: from load_data import load_mnist

MNIST = load_mnist('./')
images, labels = MNIST['data'], MNIST['target']
```

Now, let's plot a digit from the dataset:

```
In [82]: plt.figure(figsize=(4,4))
plt.imshow(images[0].reshape(28,28), cmap='gray');
plt.grid(False)
```



PCA

Now we will implement PCA. Before we do that, let's pause for a moment and think about the steps for performing PCA. Assume that we are performing PCA on some dataset \mathcal{X} for M principal components. We then need to perform the following steps, which we break into parts:

1. Data normalization (`normalize`).
2. Find eigenvalues and corresponding eigenvectors for the covariance matrix \mathcal{S} . Sort by the largest eigenvalues and the corresponding eigenvectors (`eig`).
3. Compute the orthogonal projection matrix and use that to project the data onto the subspace spanned by the eigenvectors.



Data normalization `normalize`

You will first implement the data normalization mentioned above.

Before you implement the main steps of PCA, you need to do some data preprocessing.

To preprocess the dataset for PCA, you need to make sure that the dataset has zero mean. Given a dataset \mathbf{X} , subtract the mean vector from each row of the dataset to obtain a zero-mean dataset $\bar{\mathbf{X}}$. In the first part of this notebook, you will implement `normalize` to do that. In practice, sometimes you shall also divide by standard deviation, but this is not necessary for this notebook.

To work with images, it's also a common practice to convert the pixels from unsigned integer 8 (uint8) encoding to a floating point number representation between 0-1. We will do this conversion for you for the MNIST dataset so that you don't have to worry about it.

Data normalization is a common practice. More details can be found in [Data Normalization or Feature Scaling](#).

```
In [83]: # GRADED FUNCTION: DO NOT EDIT THIS LINE
def normalize(X):

    """Normalize the given dataset X to have zero mean.

    Args:
        X: ndarray, dataset of shape (N,D) where D is the dimension of the data,
           and N is the number of datapoints

    Returns:
        (Xbar, mean): tuple of ndarray, Xbar is the normalized dataset
                       with mean 0; mean is the sample mean of the dataset.
    """
    # YOUR CODE HERE
#### Uncomment and modify the code below
N, D = X.shape
mu = np.mean(X, axis=0) # <-- EDIT THIS, compute the mean of X
Xbar = X - mu # <-- EDIT THIS, compute the normalized data Xbar by subtracting mu from each row of X
return Xbar, mu
```

```
In [84]: """Test data normalization"""
from numpy.testing import assert_allclose

X0 = np.array([[0, 0.0],
               [1.0, 1.0],
               [2.0, 2.0]])
X0_normalize, X0_mean = normalize(X0)
# Test that normalized data has zero mean
assert_allclose(np.mean(X0_normalize, 0), np.zeros((2,)))
assert_allclose(X0_mean, np.array([1.0, 1.0]))
assert_allclose(normalize(X0_normalize)[0], X0_normalize)

X0 = np.array([[0, 0.0],
               [1.0, 0.0],
               [2.0, 0.0]])
X0_normalize, X0_mean = normalize(X0)
# Test that normalized data has zero mean and unit variance
assert_allclose(np.mean(X0_normalize, 0), np.zeros((2,)))
assert_allclose(X0_mean, np.array([1.0, 0.0]))
assert_allclose(normalize(X0_normalize)[0], X0_normalize)

# Some hidden tests below
# ...
```

Compute eigenvalues and eigenvectors `eig`

Recall that in order to find the principal components of the data, you must find the eigenvalues of the covariance matrix. In the next function, you shall write code which computes the eigenvalues and eigenvectors for some covariance matrix and then sorts them in decreasing order of the eigenvalues.

You can use the library routines in `np.linalg.*` ([documentation](#)) for this.

```
In [85]: # GRADED FUNCTION: DO NOT EDIT THIS LINE
def eig(S):
    """Compute the eigenvalues and corresponding eigenvectors
    for the covariance matrix S.

    Args:
        S: ndarray, covariance matrix

    Returns:
        (eigvals, eigvecs): ndarray, the eigenvalues and eigenvectors

    Note:
        the eigenvals and eigenvects should be sorted in descending
        order of the eigen values
    """
    # YOUR CODE HERE
    # Uncomment and modify the code below
    # Compute the eigenvalues and eigenvectors
    # Note that you can compute both of these with just a single function call
    eigvals, eigvecs = np.linalg.eig(S)

    # The eigenvalues and eigenvectors need to be sorted in descending order according to the eigenvalues
    # We will use 'np.argsort' to find a permutation of the indices of eigvals that will sort eigvals in ascending order and
    # then find the descending order via [::-1], which reverse the indices
    # (https://docs.scipy.org/doc/numpy/reference/generated/numpy.argsort.html)
    # sort_indices = np.argsort(eigvals)[::-1]
    sort_indices = np.argsort(eigvals)[::-1]
    # Notice that we are sorting the columns (not rows) of eigvecs since the columns represent the eigenvectors.
    return eigvals[sort_indices], eigvecs[:, sort_indices]
```

Some test cases for implementing `eig`.

```
In [86]: def _flip_eigenvectors(B):
    """Flip the eigenvectors.
    """
    signs = np.sign(B[np.argmax(np.abs(B), axis=0), range(B.shape[1])])
    return B * signs

def _normalize_eigenvectors(B):
    # Normalize eigenvectors to have unit length
    # Also flip the direction of the eigenvector based on
    # the first element
    B_normalized = B / np.linalg.norm(B, axis=0)
    for i in range(B.shape[1]):
        if (B_normalized[0, i] < 0):
            B_normalized[:, i] = -B_normalized[:, i]
    return B_normalized

A = np.array([[3, 2], [2, 3]])
expected_eigenvalues = np.array([5., 1.])
expected_eigenvectors = np.array(
    [[ 0.70710678, -0.70710678],
     [ 0.70710678,  0.70710678]])
actual_eigenvalues, actual_eigenvectors = eig(A)
# Check that the eigenvalues match
assert_allclose(actual_eigenvalues, expected_eigenvalues)
# Check that the eigenvectors match
assert_allclose(
    _normalize_eigenvectors(actual_eigenvectors),
    _normalize_eigenvectors(expected_eigenvectors),
```

Compute projection matrix

Next given a orthonormal basis spanned by the eigenvectors, you will compute the projection matrix. The projection matrix P can be computed as follows:

$$P = B(B^T B)^{-1} B^T$$

where B is the basis vectors arranged as the columns of the matrix.

```
In [87]: # GRADED FUNCTION: DO NOT EDIT THIS LINE
def projection_matrix(B):
    """Compute the projection matrix onto the space spanned by 'B'
    Args:
        B: ndarray of dimension (D, M), the basis for the subspace
    Returns:
        P: the projection matrix
    """
    # YOUR CODE HERE
    # Uncomment and modify the code below
    return B @ np.linalg.inv(B.T @ B) @ B.T # <-- EDIT THIS to compute the projection matrix
```

```
In [88]: B = np.array([[1, 0],
                    [1, 1],
                    [1, 2]])

assert_allclose(
    projection_matrix(B),
    np.array([[5, 2, -1],
              [2, 2, 2],
              [-1, 2, 5]]) / 6
)

# Some hidden tests below
```

Compute principal component analysis

Now that you have completed `normalize`, `eig` and `projection_matrix`, you are now ready to combine these functions to do PCA. Please do this function only after you have successfully implemented the previous three functions, as this function relies on them being correct. Even if you have implemented `PCA` correctly, you shall not get the score for this function if the previous three functions that you have implemented are incorrect. If you want, you can submit the notebook and make sure of their correctness.

You can compute the covariance matrix S as follows:

$$S = \frac{\bar{X}^T \bar{X}}{n}$$

where \bar{X} is the normalized dataset and n is the number of datapoints.

```
In [89]: # GRADED FUNCTION: DO NOT EDIT THIS LINE
def PCA(X, num_components):
    """
    Args:
        X: ndarray of size (N, D), where D is the dimension of the data,
           and N is the number of datapoints
        num_components: the number of principal components to use.
    Returns:
        the reconstructed data, the sample mean of the X, principal values
        and principal components
    """
    # YOUR CODE HERE
    N, D = X.shape
    # your solution should take advantage of the functions you have implemented above.
    # Uncomment and modify the code below
    # first perform normalization on the digits so that they have zero mean and unit variance
    X_normalized, mean = normalize(X) # EDIT THIS
    # Then compute the data covariance matrix S
    S = np.cov(X_normalized, rowvar=False, bias=True) # EDIT THIS
```

```

# Next find eigenvalues and corresponding eigenvectors for S
eig_vals, eig_vecs = eig(S)
# Take the top 'num_components' of eig_vals and eig_vecs,
# This will be the corresponding principal values and components
# Remember that the eigenvectors are the columns of the matrix 'eig_vecs'
eig_vals, eig_vecs = eig(S)
# Due to precision errors, the eigenvectors might come out to be complex, so only take their real parts
principal_vals = eig_vals[:num_components]
principal_components = eig_vecs[:, :num_components]
# Due to precision errors, the eigenvectors might come out to be complex, so only take their real parts
principal_components = np.real(principal_components)

# Reconstruct the data by projecting the normalized data on the basis spanned by the principal components
# Remember that the data points in X_normalized are arranged along the rows
# but while projecting, we need them to be arranged along the columns
# Notice that we have subtracted the mean from X so make sure that you add it back
# to the reconstructed data
reconst = np.zeros_like(X_normalized)
# return reconst, mean, principal_vals, principal_components
eig_vals, eig_vecs = eig(S)
P = projection_matrix(principal_components)
reconst = ((P @ X_normalized.T).T) + mean
return reconst, mean, principal_vals, principal_components

```

You can use the code block below to incrementally test whether your function is working correctly. Uncomment the last line of the PCA function and run the code block to check if you have made a mistake.

```

In [90]: X = np.array([[3, 6, 7],
                  [8, 9, 8],
                  [1, 5, 2]])

reconst, mean, principal_vals, principal_components = PCA(X, 1)

print('Cheacking mean...')
mean_exp = np.array([4, 20 / 3, 3])
np.testing.assert_allclose(mean, mean_exp, rtol=1e-5)
print("Mean is computed correctly!")

print('Checking principal values...')
principal_vals_exp = np.array([15.39677773])
np.testing.assert_allclose(principal_vals, principal_vals_exp, rtol=1e-5)
print('Principal Values are computed correctly!')

print('Checking principal components...')
principal_components_exp = np.array([[-0.68811066],
                                     [-0.40362611],
                                     [ 0.60298398]])
np.testing.assert_allclose(principal_components, principal_components_exp, rtol=1e-5)
print("Principal components are computed correctly!")

print('Checking reconstructed data...')
reconst_exp = np.array([[ 1.68166528,  5.30679755,  5.03153182],
                      [ 7.7868029 ,  8.8878974 , -0.31833472],
                      [ 2.53153182,  5.80538505,  4.2868029 ]])
np.testing.assert_allclose(reconst, reconst_exp, rtol=1e-5)
print("Reconstructed data is computed correctly!")

```

```

Cheacking mean...
Mean is computed correctly!
Checking principal values...
Principal Values are computed correctly!
Checking principal components...
Principal components are computed correctly!
Checking reconstructed data...
Reconstructed data is computed correctly!

```

```
In [91]: def draw_vector(v0, v1, ax=None, label=None):
    """Draw a vector from v0 to v1."""
    ax = ax or plt.gca()
    arrowprops=dict(arrowstyle='->',
                    linewidth=2,
                    shrinkA=0, shrinkB=0,
                    color='k')
    ax.annotate('', v1, v0, arrowprops=arrowprops, label=label)
```

Some test cases that check the implementation of PCA

```
In [92]: D = 2
N = 10
# Generate a dataset X from a 2D Gaussian distribution
mvn = scipy.stats.multivariate_normal(
    mean=np.ones(D, dtype=np.float64),
    cov=np.array([[1, 0.8], [0.8, 1]]), dtype=np.float64)
)

X = mvn.rvs((N,), random_state=np.random.RandomState(0))
reconst, m, pv, pc = PCA(X, 1)
# Check the shape returned by the PCA implementation matches the specification.
assert reconst.shape == X.shape
assert m.shape == (D, )
assert pv.shape == (1, )
assert pc.shape == (D, 1)

# Check that PCA with num_components == D gives identical reconstruction
reconst, m, pv, pc = PCA(X, D)
assert reconst.shape == X.shape
assert m.shape == (D, )
assert pv.shape == (2, )
assert pc.shape == (D, 2)
assert_allclose(reconst, X)
```

In []:

Visualize PCA

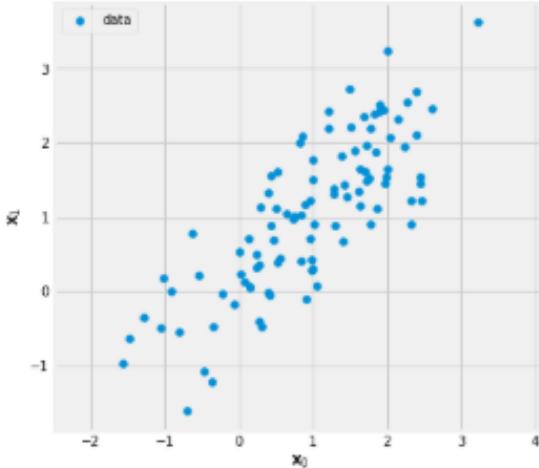
We will first visualize what PCA does on a 2D toy dataset. You can use the visualization below to get better intuition about what PCA does and use it to double check your code above.

```
In [93]: # This code generates two dimensional points with mean (1, 1) and a covariance of 0.8 between the x and y coordinates
# Feel free to change these values to generate different distributions
mvn = scipy.stats.multivariate_normal(
    mean=np.array([1, 1]),
    cov=np.array([[1, 0.8], [0.8, 1]]))

# X contains 100 points generated according to the above criteria
X = mvn.rvs((100,), random_state=np.random.RandomState(0))

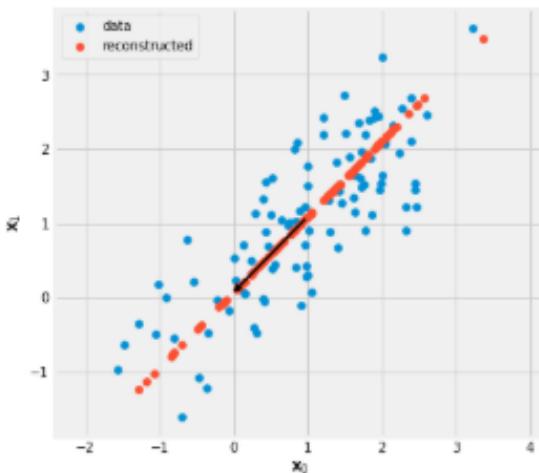
num_components = 1
X_reconst, mean, principal_values, principal_components = PCA(X, num_components)

# We shall plot the points that we have generated below
# You do not need to modify this code
fig, ax = plt.subplots(figsize=(6, 6))
ax.scatter(X[:, 0], X[:, 1], label='data')
plt.axis('equal')
plt.legend()
ax.set(xlabel='\mathbf{x}_0', ylabel='\mathbf{x}_1')
plt.show()
```



We shall now perform PCA on this data and plot the reconstructed points and the direction of the principal component. Can you guess the axis where the reconstructed points should lie?

```
In [94]: # The code below plots the original points, the reconstructed points and the principal component vector
# You do not need to modify it
fig, ax = plt.subplots(figsize=(6, 6))
ax.scatter(X[:, 0], X[:, 1], label='data')
for (principal_variance, principal_component) in (zip(principal_values, principal_components.T)):
    draw_vector(
        mean, mean + np.sqrt(principal_variance) * principal_component,
        ax=ax)
ax.scatter(X_reconst[:, 0], X_reconst[:, 1], label='reconstructed')
plt.axis('equal')
plt.legend()
ax.set(xlabel='$\mathbf{x}_0$', ylabel='$\mathbf{x}_1$')
plt.show()
```



We can also compare our PCA implementation with the implementation in scikit-learn (a popular machine learning library in Python that includes implementation of PCA) to see if we get similar results. Note that even if your implementation is correct, the answer your code gives and the answer scikit-learn gives may not be absolutely identical because floating point operations are never exact and always truncated. That is why we allow for a margin of error in the comparison.

```
In [95]: random = np.random.RandomState(0)
X = random.randn(10, 5)

from sklearn.decomposition import PCA as SKPCA

for num_component in range(1, 4):
    # We can compute a standard solution given by scikit-Learn's implementation of PCA
    pca = SKPCA(n_components=num_component, svd_solver="full")
    sklearn_reconst = pca.inverse_transform(pca.fit_transform(X))
    reconst, _, _, _ = PCA(X, num_component)
    # The difference in the result should be very small (<10^-20)
    print(
        "difference in reconstruction for num_components = {}: {}".format(
            num_component, np.square(reconst - sklearn_reconst).sum()
        )
    )
    np.testing.assert_allclose(reconst, sklearn_reconst)

difference in reconstruction for num_components = 1: 2.643993664852573e-28
difference in reconstruction for num_components = 2: 4.477357638322725e-28
difference in reconstruction for num_components = 3: 7.867635675115741e-29
```

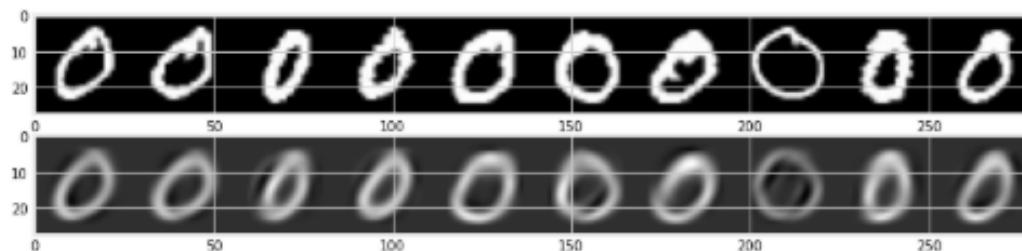
PCA for MNIST digits

Once you have implemented PCA correctly, it's time to apply to the MNIST dataset. Each image in the dataset has a size of 28*28 pixels and thus has a dimensionality of 784. First, we will do some preprocessing of the data to get it into a good shape.

```
In [96]: ## Some preprocessing of the data
NUM_DATAPOINTS = 1000
X = (images.reshape(-1, 28 * 28)[:NUM_DATAPOINTS]) / 255.
```

Next, we shall perform PCA on the images and display the original images and the reconstructed images below the original ones. We are currently using 10 principal components, but we encourage you to change this and see how the reconstructed images differ for different number of components as well. In particular, how good is the reconstruction on using just one or two components? Also, are all the 784 components necessary to get a reconstruction which is pretty much the same as the original image?

```
In [97]: reconst, _, _, _ = PCA(X, num_components=10)
num_images_to_show = 10
reconst_images = np.reshape(reconst[:num_images_to_show], (-1, 28, 28))
fig, ax = plt.subplots(2, 1, figsize=(num_images_to_show * 3, 3))
ax[0].imshow(np.concatenate(np.reshape(X[:num_images_to_show], (-1, 28, 28)), -1), cmap="gray")
ax[1].imshow(np.concatenate(reconst_images, -1), cmap="gray");
```



The greater number of principal components we use, the smaller will our reconstruction error be. Now, let's answer the following question:

How many principal components do we need in order to reach a Mean Squared Error (MSE) of less than 10.0 for our dataset?

Remember that MSE is a measure of the loss of data during reconstruction. Basically it is the mean of the squared distances of the reconstructed data points from the original datapoints.

We have provided a function in the next cell which computes the mean squared error (MSE), which will be useful for answering the question above.

```
In [54]: def mse(predict, actual):
    """Helper function for computing the mean squared error (MSE)"""
    return np.square(predict - actual).sum(axis=1).mean()
```

```
In [58]: loss = []
reconstructions = []
# iterate over different number of principal components, and compute the MSE
for num_component in range(1, 100, 5):
    reconst, _, _ = PCA(X, num_component)
    error = mse(reconst, X)
    reconstructions.append(reconst)
    # print('n = {}, reconstruction_error = {:.2f}'.format(num_component, error))
    loss.append((num_component, error))

reconstructions = np.asarray(reconstructions)
reconstructions = reconstructions
loss = np.asarray(loss)
```

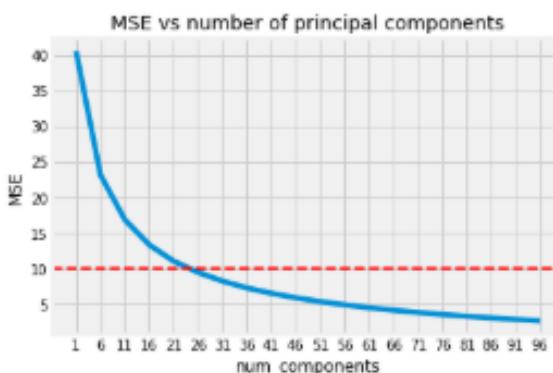
```
In [59]: import pandas as pd
# create a table showing the number of principal components and MSE
pd.DataFrame(loss, columns=['no. of components', 'mse']).head()
```

```
Out[59]:
```

	no. of components	mse
0	1.0	40.618882
1	6.0	23.198508
2	11.0	16.899711
3	16.0	13.367106
4	21.0	11.072143

We can also put these numbers into perspective by plotting them.

```
In [60]: fig, ax = plt.subplots()
ax.plot(loss[:,0], loss[:,1]);
ax.axhline(10, linestyle='--', color='r', linewidth=2)
ax.xaxis.set_ticks(np.arange(1, 100, 5));
ax.set(xlabel='num_components', ylabel='MSE', title='MSE vs number of principal components');
```



But numbers don't tell us everything! Just what does it mean qualitatively for the loss to decrease from around 45.0 to less than 10.0?

Let's find out! In the next cell, we draw the the leftmost image is the original digit. Then we show the reconstruction of the image on the right, in descending number of principal components used.