Section 13 Dimension Reduction: Principal Component Analysis

Starting from this lecture, we are goting to talk about **unsupervised machine learning**. The fundamental difference with supervised learning is that in unsupervised learning problems, there is no label (response) y to be predicted. All we have is the data matrix $X \in \mathbb{R}^{n\times p}$, and the general task is to explore the "pattern" of data.

Important Remark: For simplicity, below we assume that all variables (features) of X has mean zero. For arbitrary X, we can pre-process it by substracting the mean of each variable for the corresponding column.

One classical type of problem in unsupervised learning is **Dimension Reduction** (another type of problem is clustering).

(unrigorous) Mathematical Description: Given high-dimensional data observation $\mathbf{x} \in \mathbb{R}^{1\times p}$ (imagine this row matrix $\mathbf{x}$ as one sample), find a "reasonable" projection function

$$\mathbf{t} = \mathbf{h}(\mathbf{x}) : \mathbb{R}^{1\times p} \to \mathbb{R}^{1\times k}, k << p$$

that "preserves" the high-dimensional information.

- A naive solution is to randomly pick k components of $\mathbf{x}$-- of course this is a huge waste of information.
- Another simple yet reasonable assumption is that $\mathbf{h}$ is linear transformation -- of course, the linear coefficients should depend on the "structure" of dataset. In other words, the "new coordinates" are the linear combination of "old coordinates".

  $$\mathbf{t} = \mathbf{h}(\mathbf{x}) = \mathbf{x}\mathbf{V}_k,$$
  $$\mathbf{V}_k \in \mathbb{R}^{p\times k}.$$

  In data matrix form (n samples), we have

  $$\mathbf{T}_k = \mathbf{X}\mathbf{V}_k \in \mathbb{R}^{n\times k}$$

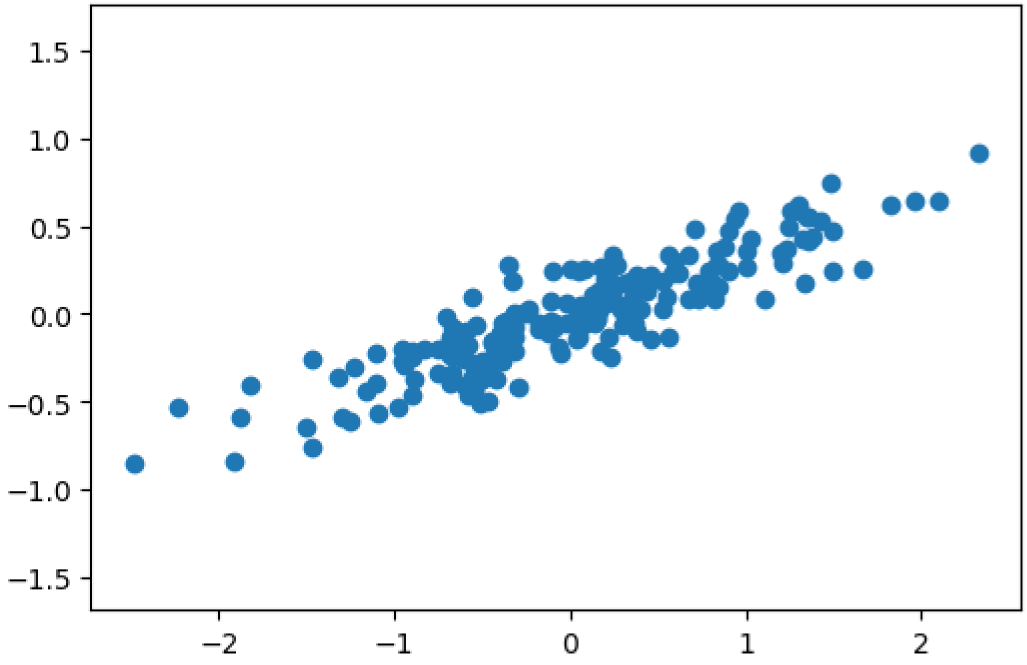
Principal Component Analysis (PCA) is one typical linear dimension reduction method. Write $\mathbf{V}_k = [\mathbf{v}_1\mathbf{v}_2\cdots\mathbf{v}_k]$, then the column vectors $\mathbf{v}_j\,(1\leq j\leq k)$ are called the first k **Principal Components (PCs)** of the dataset, and T_k is called the score matrix -- each row represents the k scores of one sample in k PCs -- they are the representation of the sample in $\mathbb{R}^k$ space.

Now the central question becomes: how to find the PCs based on the dataset?

In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
rng = np.random.RandomState(1)
X = np.matmul(rng.rand(2, 2), rng.randn(2, 200)).T
fig = plt.figure(dpi=100)
```

```
plt.scatter(X[:, 0], X[:, 1])
plt.axis('equal');
```



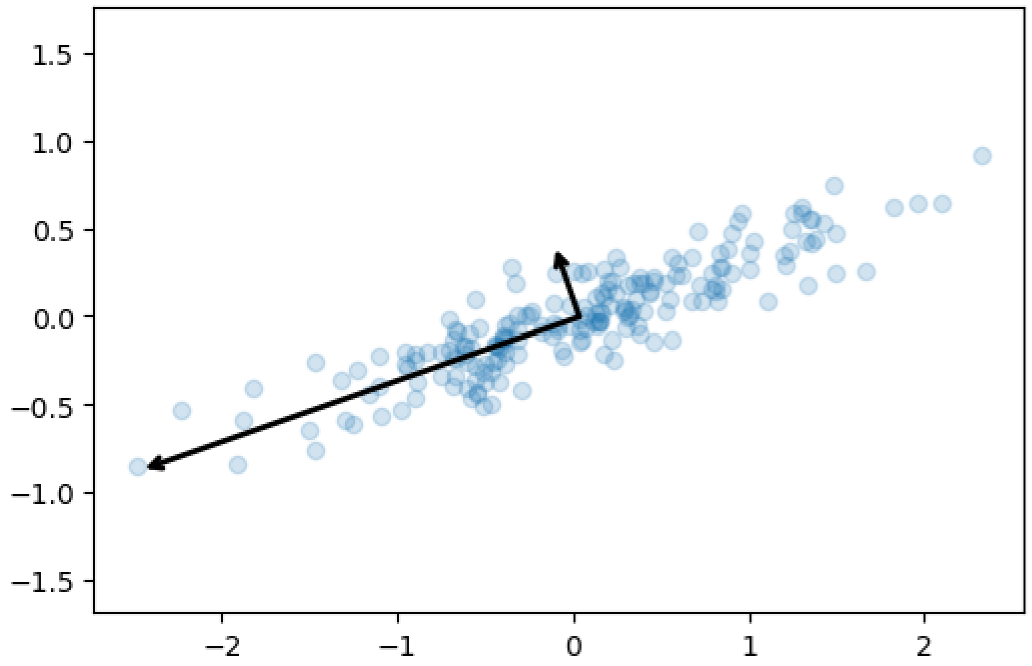
The data is in 2D space. If I "force" you to believe that the data can be further reduced to one-dimension , you may (reluctantly) admit that data is generated along one line (the long axis of "ellipse" sketched by the data), and the other short axis merely corresponds "noise".

In [12]:

```
from sklearn.decomposition import PCA
# run pca from sklearn
pca = PCA(n_components=2)
pca.fit(X)

def draw_vector(v0, v1, ax=None):
    ax = ax or plt.gca() #plt.gca(): Get current axes
    arrowprops=dict(arrowstyle='->',
                    linewidth=2,
                    shrinkA=0, shrinkB=0)
    ax.annotate('', v1, v0, arrowprops=arrowprops)

# plot data
fig = plt.figure(dpi=100)
plt.scatter(X[:, 0], X[:, 1], alpha=0.2)
for length, vector in zip(pca.explained_variance_, pca.components_):
    v = vector * 3 * np.sqrt(length) #the vectors in pca.components_ are unit vectors.
    draw_vector(pca.mean_, pca.mean_ + v) #pca.mean_ allows us to center our axes bette
plt.axis('equal');
```



Therefore we may imagine the major "axis" of ellipse are the principal components of the data, and the reduced coordinate are the projections on such direction.

How do we determine the direction of "ellipse axis"? A staightfoward way is from **covariance matrix** of the data (to fully understand this, you need some knowledge about quadratic forms in linear algebra and multivariate Gaussian distribution in probability, although it's optional for the basic requirements).

PCA from Covariance Matrix

- **Step 0**: Center the data matrix, making it column mean zero.
- **Step 1**: Calculate the covariance matrix

  $$\Sigma = \frac{1}{n-1} X^\top X \in \mathbb{R}^{p\times p}.$$

  The element Σ_{ij} denotes the covariance between variable (feature) i and j in the data.
- **Step 2**: Eigen-decomposition of symmetrix covariance matrix Σ,

  $$\Sigma = V\Lambda V^\top,$$

  where $V \in \mathbb{R}^{p\times p}$ is orthogonal matrix whose columns are unit eigen-vectors and Λ is the diagonal matrix of eigen-values. We further arrange the λ_j in descending orders.
- **Step 3**: Principal Components are just the first k columns of V, denoting as V_k. Indeed, they are the eigen-vectors corresponding to the top k eigen-values.
- **Step 4**: Compute the score matrix

  $$T_k = XV_k.$$

Then each row of T_k is the coordinate of the sample in $\mathbb{R}^k$ space.

(Optional) Remark: The covariance matrix of T_k is $\frac{1}{n-1}T_k^\top T_k = V_k^\top \Sigma V_k = \Lambda_k$, which is defined as the (1:k,1:k) submatrix of Λ. This means that different PCs are independent from each other, and λ_j is indeed the variance of j-th PC.

In [13]:
```
import numpy as np

class myPCA():
    '"write your document strings here"'

    def __init__(self, n_components = 2):
        '"write your document strings here"'
        self.n_c = n_components


    def fit(self,X):
        '"write your document strings here"'
        cov_mat = np.cov(X.T) # covariance matrix, the input matrix to this function do
        eig_val, eig_vec = np.linalg.eigh(cov_mat) #eigen-values and orthogonal eigen-v
        eig_val = np.flip(eig_val) # reverse the order --descending
        eig_vec = np.flip(eig_vec,axis=1) # reverse the order
        self.eig_values = eig_val[:self.n_c] # select the top eigen-vals
        self.principle_components = eig_vec[:,:self.n_c] # select the top eigen-vecs
        self.variance_ratio = self.eig_values/eig_val.sum() # variance explained by eac

    def transform(self,X):
        '"write your document strings here"'
        return np.matmul(X-X.mean(axis = 0),self.principle_components) #project the dat
```

In [14]:
```
from sklearn.datasets import load_digits
X,y = load_digits(return_X_y = True)
```

In [15]:
```
X.shape
```

Out[15]:
```
(1797, 64)
```

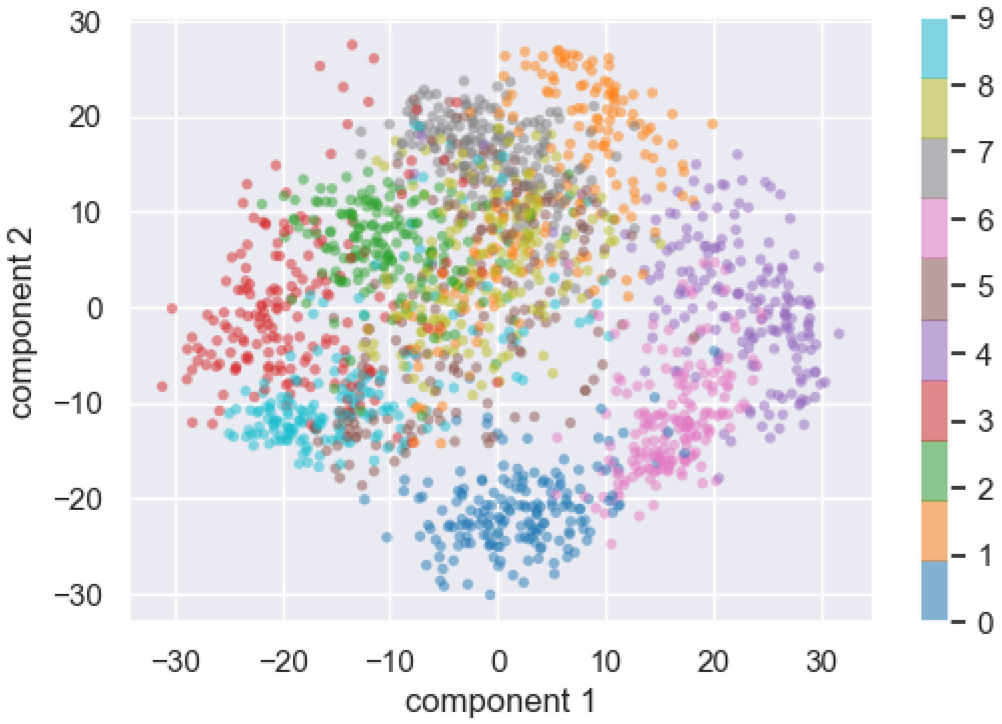
In [16]:
```
pca = myPCA(n_components = 15)
pca.fit(X)
X_pca = pca.transform(X)
```

In [5]:
```
X_pca.shape
```

Out[5]:
```
(1797, 15)
```

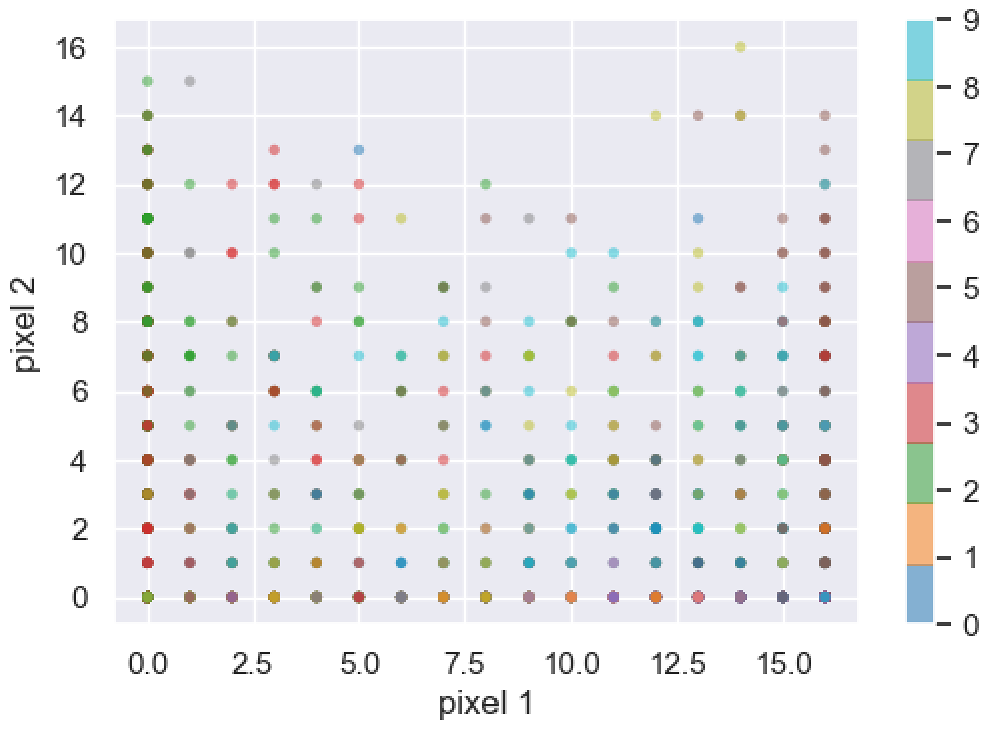
We have successfully reduced our number of components from 64 pixels to 15 features. Since graphing in 15-dimensional space is still too hard, let's just plot the first two features `X_pca[:,0]` and `X_pca[:,1]` and color these points using `y` from when we loaded our digits. This coloring will help us visualize how well our algorithm clustered samples from the same class.

In [26]:
```
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
figure = plt.figure(dpi=100)
plt.scatter(X_pca[:, 0], X_pca[:, 1],c=y, s=15, edgecolor='none', alpha=0.5,cmap=plt.cm
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.colorbar();
```



But wait, couldn't we have just picked any two pixels at random from our original image, and plotted those? Let's compare how well samples of the same class are clustered in this case.

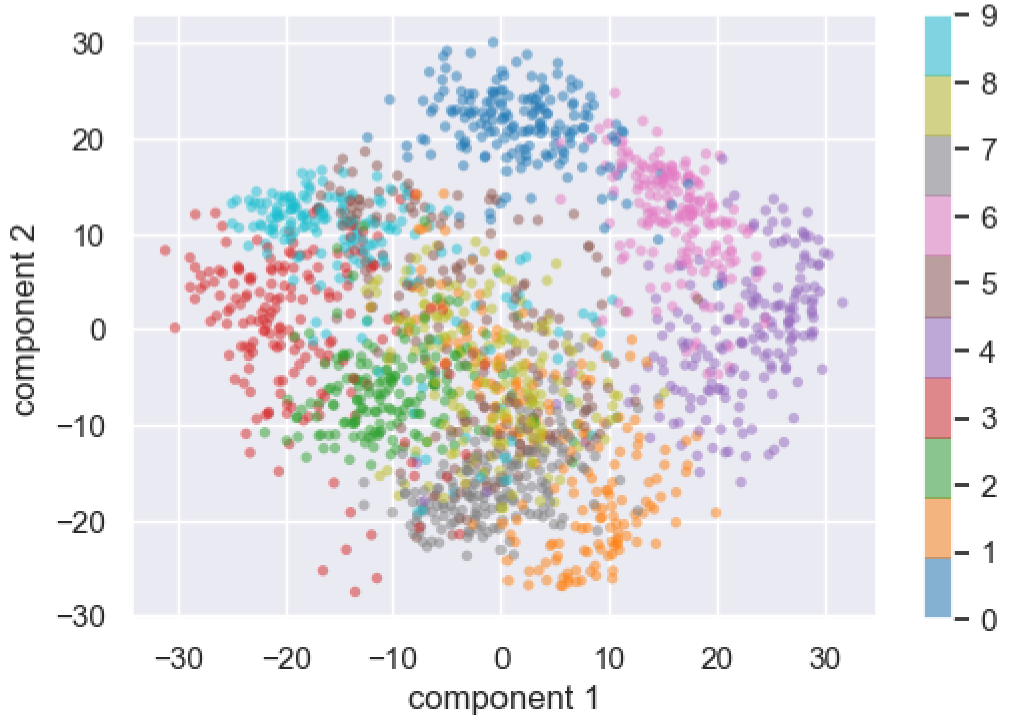
In [8]:
```
from numpy import random
figure = plt.figure(dpi=100)
rand_ind = random.randint(64, size=(2)) # pick up two random pixels from original data
plt.scatter(X[:, rand_ind[0]], X[:, rand_ind[1]],c=y, s=15, edgecolor='none', alpha=0.5
plt.xlabel('pixel 1')
plt.ylabel('pixel 2')
plt.colorbar();
```



Now lets take our original data `X` with 64 pixels, reduce it to 2 components, and compare this to our "2-out-of-15-components" plot.

In [27]:
```
from sklearn.decomposition import PCA
pca_sklearn = PCA(n_components=2)
projected = pca_sklearn.fit_transform(X)

figure = plt.figure(dpi=100)
plt.scatter(projected[:, 0], projected[:, 1],c=y, s=15, edgecolor='none', alpha=0.5,cma
plt.xlabel('component 1')
plt.ylabel('component 2')
plt.colorbar();
```



In [28]:
```
print(pca.principle_components[:,1]) # principle components (directions), in our code
print(pca_sklearn.components_[1,:]) # in sklearn, note the shapes are different
```

```
[ 0.00000000e+00  1.01064569e-02  4.90849204e-02  9.43337493e-03
  5.36015636e-02  1.17755318e-01  6.21281792e-02  7.93574578e-03
  1.63216259e-04  2.10167064e-02 -6.03485687e-02  5.33769554e-03
  9.19769205e-02  5.19210493e-02  5.89354684e-02  3.33283413e-03
  4.22872096e-05 -3.62458505e-02 -1.98257337e-01  4.86386550e-02
  2.25574894e-01  4.50541862e-03 -2.67696727e-02  2.08735745e-04
  5.66233953e-05 -7.71235121e-02 -1.88447107e-01  1.37952518e-01
  2.61042779e-01 -4.98350596e-02 -6.51113775e-02 -4.03200346e-05
  0.00000000e+00 -8.81559918e-02 -8.71737595e-02  2.70860181e-01
  2.85291800e-01 -1.66461582e-01 -1.27860543e-01  0.00000000e+00
 -2.89440157e-04 -5.08304859e-02 -1.30274463e-01  2.68906468e-01
  3.01575537e-01 -2.40259064e-01 -2.17555551e-01 -1.32726068e-03
 -2.86742937e-04 -1.05548282e-02 -1.53370694e-01  1.19535173e-01
  9.72508046e-02 -2.85869538e-01 -1.48776446e-01 -5.42290907e-04
  3.34028085e-05  1.00791167e-02  7.02724074e-02 -1.71108112e-02
 -1.94296399e-01 -1.76697117e-01 -1.94547053e-02  6.69693895e-03]
[-6.63932261e-17 -1.01064569e-02 -4.90849276e-02 -9.43337296e-03
 -5.36015501e-02 -1.17755323e-01 -6.21281839e-02 -7.93574585e-03
 -1.63216181e-04 -2.10166996e-02  6.03485706e-02 -5.33768441e-03
 -9.19768866e-02 -5.19210469e-02 -5.89354734e-02 -3.33283343e-03
 -4.22871216e-05  3.62458633e-02  1.98257329e-01 -4.86386642e-02
 -2.25574890e-01 -4.50542173e-03  2.67696758e-02 -2.08734925e-04
 -5.66233570e-05  7.71235193e-02  1.88447106e-01 -1.37952526e-01
 -2.61042798e-01  4.98350550e-02  6.51113886e-02  4.03200585e-05
 -0.00000000e+00  8.81559891e-02  8.71737697e-02 -2.70860158e-01
 -2.85291806e-01  1.66461575e-01  1.27860550e-01 -0.00000000e+00
  2.89439977e-04  5.08304725e-02  1.30274445e-01 -2.68906469e-01
 -3.01575551e-01  2.40259055e-01  2.17555556e-01  1.32726067e-03
  2.86742832e-04  1.05548208e-02  1.53370671e-01 -1.19535171e-01
 -9.72507995e-02  2.85869550e-01  1.48776445e-01  5.42290283e-04
 -3.34028158e-05 -1.00791175e-02 -7.02724156e-02  1.71108116e-02
  1.94296406e-01  1.76697116e-01  1.94546991e-02 -6.69694109e-03]
```

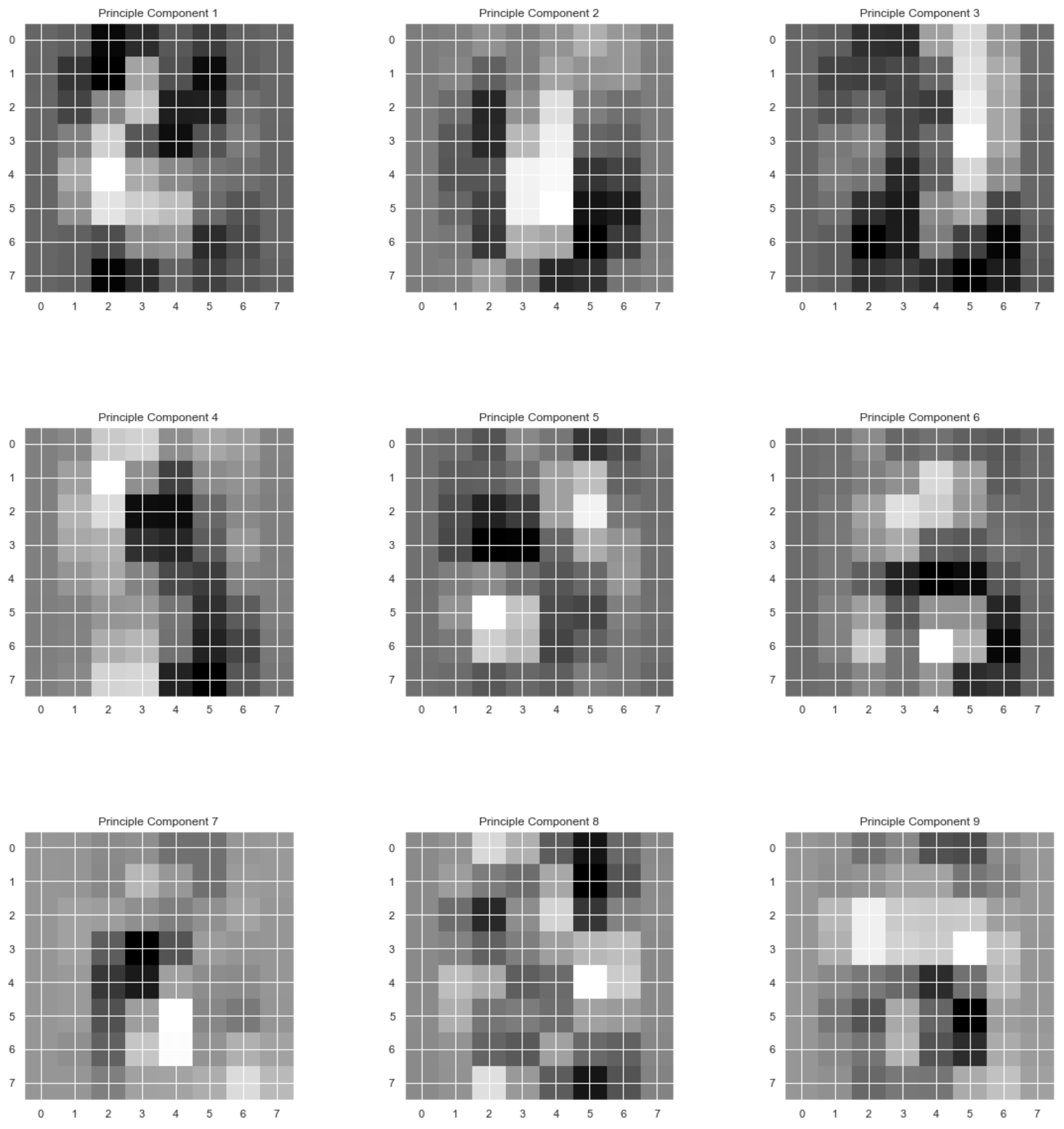
In [29]:
```
print(pca.variance_ratio) # ratio of variance explained by first few PCs
print(pca_sklearn.explained_variance_ratio_)
```

```
[0.14890594 0.13618771 0.11794594 0.08409979 0.05782415 0.0491691
 0.04315987 0.03661373 0.03353248 0.03078806 0.02372341 0.02272697
 0.01821863 0.01773855 0.01467101]
[0.14890594 0.13618771]
```

Visualize the principle components

In [12]:

```
fig, axs = plt.subplots(3,3, figsize=(20, 20))
fig.subplots_adjust(hspace = .5, wspace=.1)
axs = axs.ravel()
for i in range(9):
    axs[i].imshow(pca.principle_components[:,i].reshape(8,8),cmap=plt.cm.gray)
    axs[i].set_title('Principle Component '+str(i+1))
```



Decomposition of the original data with PC:

$$X = XVV^T = TV^T$$

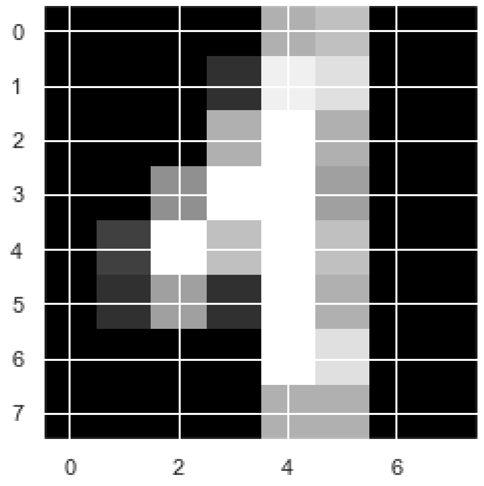
Take the $i-th$ row, we have

$$x_i = \sum_{j=1}^{p} t_{ij} v_j^T$$

In [35]:

```
i = 200
plt.imshow(X[i,:].reshape(8,8),cmap=plt.cm.gray)
print(X_pca[i,:],y[i])
```

```
[  7.2077319    9.86510963  13.93678278 -25.0953412   -1.68236039
   4.21409169  -7.09187188  -0.73268062  -6.63871124   6.32758422
  -2.58379089  12.75161224  -1.97174185   3.84649193  -1.68604951] 1
```



Another Algorithm: Singular Value Decomposition (SVD)

The PCA in scikit-learn is realized with SVD of data matrix, which sometimes can be more stable numerically than covariance matrix-based approach.

Let $X \in \mathbb{R}^{n\times p}$ be the centered data matrix (each feature is of mean zero).

> Singular Value Decomposition (SVD): any real matrix can be decomposed into the following form:
>
> $$X = USV^\top$$

For specific detail on these matrices:

1. $S \in \mathbb{R}^{n\times p}$ is a diagonal matrix whose diagonal entries are non-negative and in decreasing order. The elements along the diagonal are positive square roots of the eigenvalues of $X^\top X$.

2. $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{p\times p}$ are orthogonal matrices (i.e. columns of V are orthogonal, meaning $V^\top V = VV^\top = I$. For U we also have $U^\top U = UU^\top = I$).

3. The columns of $V = [\mathbf{v}_1 \ \mathbf{v}_2 \ \cdots \ \mathbf{v}_p]$ are are known as the right singular vectors. They are the eigenvectors of $X^\top X$. More importantly for us, they are indeed the **principal components**

(important directions).

4. The columns of $U = [\mathbf{u}_1 \ \mathbf{u}_2 \ \cdots \ \mathbf{u}_n]$ are known as the left singular vectors. They are the eigenvectors of $XX^\top$.

Relation with covariance-based approach: Since X is centered, we have covariance matrix

$$\Sigma = \frac{1}{n-1} X^\top X = \frac{1}{n-1} V S^\top S V^\top,$$

Note that $S^\top S$ is a diagonal matrix (containing eigenvalues of $X^\top X$), therefore V is the eigen-vector matrix of Σ -- i.e. principle components. The first k score (projection) matrix $T_k \in \mathbb{R}^{N \times k}$ of the data on first k-PCs are then calculated as $T_k = XV_k = U_k S_k$.

Reference Reading Suggestions

- ISL: Chapter 10
- ESL: Chapter 14.5
- PML: Chapter 20.1