Lecture 22 - Dimensionality Reduction with Principal Component Analysis (PCA)

Principal Component Analysis (PCA)

A very common approach (and one of the simplest approaches) to **dimensionality reduction** is **Principal Component Analysis** (or **PCA**).

- PCA takes data from *sensor coordinates* to *data centric coordinates* using linear transformations.

PCA uses a **linear transformation** to **minimize the redundancy** of the resulting transformed data (by ending up with data that is uncorrelated).

- This means that every transformed dimension is more informative.
- In this approach, the dimensionality of the space is still the same as the original data, but the space of features are now arranged such that they contain the most information.

If we wish to reduce dimensionality of our feature space, we can choose only the features that carry over the most information in the linearly transformed space.

- In other words, PCA will find the underlying **linear manifold** that the data is embedded in.

PCA finds the directions of maximum variance in high-dimensional data and projects it onto a new subspace with equal or fewer dimensions than the original one.

There are a couple of points-of-view on how to find A:

1. Maximum Variance Formulation
2. Minimum-error Formulation

1. PCA as Maximum Variance Formulation

Consider the data X comprised on N data samples in a D-dimensional space, so X is a matrix of size $D \times N$.

The **first step** in PCA is to centralize or demean X. This will is to guarantee that all features will have the same impact and not weight more only because their range of values is much larger (example, age vs income).

- Without loss of generality, let's assume that we subtracted the mean to the input data, X. Now, X has zero mean.

The **second step** is to find the linear transformation A that transforms X to a space where features are:

1. Uncorrelated (preserve all dimensions)
2. reduced (dimensionality reduction)

$$Y = AX$$

where A is a $D \times D$ matrix, X is a $D \times N$ data matrix and therefore Y is also a $D \times N$ transformed data matrix.

The variance of the transformed data Y is given by:

$$\begin{aligned} R_y &= E[YY^T] \\ &= E[AX(AX)^T] \\ &= E[AXX^TA^T] \\ &= AE[XX^T]A^T \\ &= AR_xA^T \end{aligned}$$

Note that we are computing the variance along the dimensions of Y, therefore, R_y is a $D \times D$ matrix. Similarly, R_X represents the covariance of the data X. Covariances matrices are symmetric therefore $R_X = R_X^T$ and $R_Y = R_Y^T$.

Similarly, R_X represents the covariance of the data X.

If we write A in terms of vector elements:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} \vec{a_1}^T \\ \vec{a_2}^T \end{bmatrix}$$

Then,

$$\begin{aligned} R_y &= \begin{bmatrix} a_1^T \\ a_2^T \end{bmatrix} R_X \begin{bmatrix} a_1 & a_2 \end{bmatrix} \\ &= \begin{bmatrix} a_1^T R_X a_1 & a_1^T R_X a_2 \\ a_2^T R_X a_1 & a_2^T R_X a_2 \end{bmatrix} \end{aligned}$$

- If we want to represent the data in a space in which the features are **uncorrelated**, what shape does the covariance matrix have to take?

Diagonal! Why?

- Can we use the eigenvectors of R_X as our linear transformation A?

Consider the case where we are trying to project the data X into a 1-dimensional space, so we are trying to find the direction a_1 where maximal data variance is preserved:

$$\arg_{a_1} \max a_1^T R_X a_1$$

We want this solution to be bounded (considering $a_1 = \infty$ would maximize), so we need to constraint the vector to have norm 1

$$\|a_1\|_2^2 = 1 \iff a_1^T a_1 = 1 \iff a_1^T a_1 - 1 = 0$$

Then, we using Lagrange Optimization:

$$\mathcal{L}(a_1, \lambda_1) = a_1^T R_X a_1 - \lambda_1(a_1^T a_1 - 1)$$

Solving for a_1:

$$\frac{\partial \mathcal{L}}{\partial a_1} = 0 \iff 2R_X a_1 - 2\lambda_1 a_1 = 0 \iff R_X a_1 = \lambda_1 a_1$$

- Does this look familiar?

This is stating that a_1 must be an eigenvector of R_X!

So coming back to the question "Can we use the eigenvectors of X as our linear transformation A?" YES!

- If we left multiply by a_1^T and make use of $a_1^T a_1 = 1$:

$$a_1^T R_X a_1 = \lambda_1$$

So the variance will be maximum when we set the project direction a_1 equal to the eigenvector having the largest eigenvalue λ_1.

- This eigenvector is known as the firt **principal component**.
- As you may anticipate, the linear trasnformation A will the be a matrix whose row entries are **sorted** the eigenvectors (sorted by their correspondent eigenvalue).

$$A = \begin{bmatrix} \vec{a_1}^T \\ \vec{a_2}^T \end{bmatrix}$$

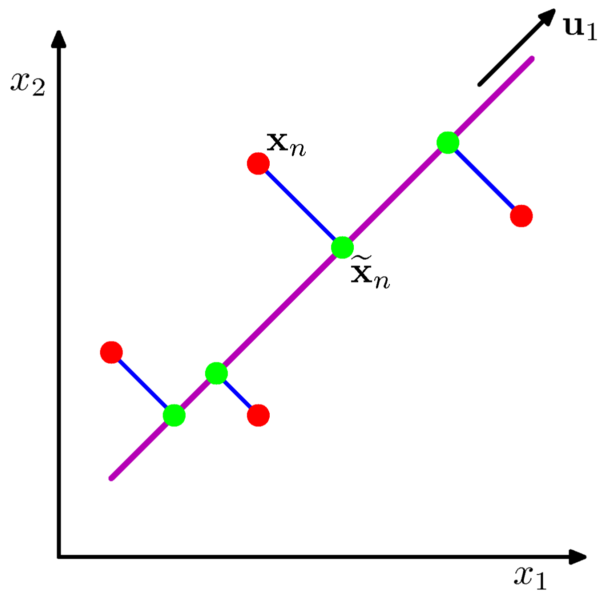
where a_1 and a_2 are eigenvectors of R_X with correspondent eigenvalues λ_1 and λ_2 with $\lambda_1 > \lambda_2$.

2. PCA as Minimum-error Formulation

We can also look at PCA as a minimization of mean squared error.

```
In [1]: from IPython.display import Image
        Image('figures/Figure12.2.png',width=300)
```

Out[1]:



Consider X and a D-dimensional orthogonal basis $\mathbf{u}$:

$$\hat{x} = \sum_{i=1}^{m} y_i a_i$$

where $m < D$.

$$y_j = x^T a_j$$

where $A^T A = I$.

We want to minimize the residual error:

$$\epsilon = x - \hat{x} = \sum_{i=m+1}^{D} y_i a_i$$

- The objective function we will use is the mean square residual:

$$\begin{aligned}
J &= E\left[\|\epsilon\|_2^2\right] \\
&= E\left[\left(\sum_{i=m+1}^{D} y_i a_i\right)\left(\sum_{i=m+1}^{D} y_i a_i\right)\right] \\
&= \sum_{j=m+1}^{D} E[y_j^2], \text{because } a_i^T a_j = 0, \forall i \neq j \text{ and } a_i^T a_j = 1, \forall i = j \\
&= \sum_{j=m+1}^{D} E[(a_j^T \mathbf{x})(\mathbf{x}^T a_j)] \\
&= \sum_{j=m+1}^{D} a_j^T E[\mathbf{x}\mathbf{x}^T] a_j \\
&= \sum_{j=m+1}^{D} a_j^T R_x a_j
\end{aligned}$$

Minimize the error and incorporate Lagrange parameters for $A^T A = I$:

$$\frac{\partial J}{\partial a_j} = 2(R_x a_j - \lambda_j a_j) = 0$$
$$R_x a_j = \lambda_j a_j$$

So, the sum of the error is the sum of the eigenvalues of the unused eigenvectors. So, we want to select the eigenvectors with the m largest eigenvalues.

Steps of PCA

Consider the data X with N data points defined in a D-dimensional space, that is, X is a $D \times N$ matrix.

1. Subtract the mean, $\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$.

2. Compute the covariance matrix R_X (by definition, the covariance already subtracts the data's mean). This matrix is of size $D \times D$.

3. Compute eigenvectors and eigenvalues of the matrix R_X, and store the sorted eigenvectors (e_i) in decreasing eigenvalue (λ_i) order.

4. Build the modal matrix $\mathbf{U} = [\mathbf{e_1} \quad | \quad \mathbf{e_2} \quad | \ldots | \quad \mathbf{e_D}]$, where all the (unit-length) eigenvectors are stacked in columns, sorted by their respective eigenvalues, i.e., $\lambda_1 > \lambda_2 > \cdots > \lambda_D$.

   - For **uncorrelating the data**, preserve all D eigenvectors. Hence $\mathbf{U}$ is a $D \times D$ matrix.
   - For **dimensionality reduction**, keep the top M eigenvectors with the largest eigenvalues. Hence $\mathbf{U}$ is a $D \times M$ matrix.

5. Apply the linear transformation: $\mathbf{y} = \mathbf{U}^T\mathbf{X}$. Here $\mathbf{y}$ is a matrix of size $M \times N$, where $M \leq D$.

Note that the formal definition of covariance already accounts for demeaning the data.

In [1]:

```
import pandas as pd
from scipy import stats
import numpy as np
import numpy.random as npr
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('bmh')
plt.rcParams['axes.grid'] = False

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

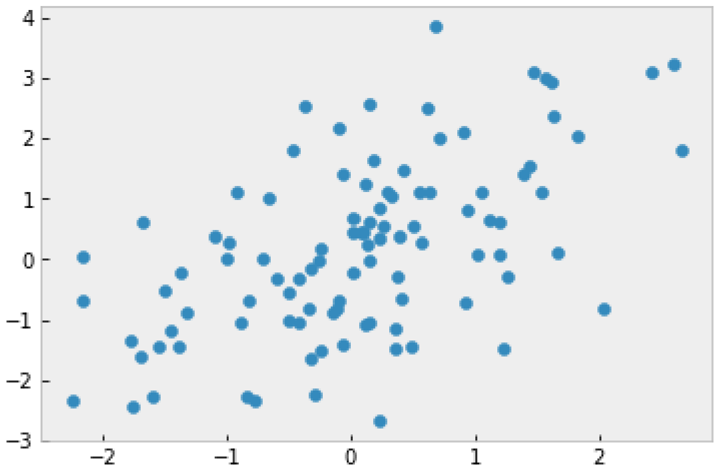
In [2]:
```
# Helper functions

def plotvec(*argv):
    colors=['k','b','r','g','c','m']
    xmin=0
    xmax=-1000000
    ymin=0
    ymax=-1000000
    origin=[0,0]
#     plt.figure()
    for e in enumerate(argv):
        i=e[0]
        arg=e[1]
        plt.quiver(*origin,*arg,angles='xy',scale_units='xy',scale=1,
                   color=colors[i%len(colors)])
        xmin=min(xmin,arg[0])
        xmax=max(xmax,arg[0])
        ymin=min(ymin,arg[1])
        ymax=max(ymax,arg[1])
#     plt.xlim(min(-1, xmin-1), max(1,xmax+1))
#     plt.ylim(min(-1,ymin-1),max(1,ymax+1))

def plot_contours(K,X=None, R=None):
    '''This function plots the contours of a Bivariate Gaussian RV with
    mean [0,0] and covariance K'''

    x = np.linspace(-4, 4, 100)
    y = np.linspace(-4, 4, 100)
    xm, ym = np.meshgrid(x, np.flip(y))
    if X is None:
        X = np.dstack([xm,ym])
    if R is not None:
        X = X@R

    G = stats.multivariate_normal.pdf(X,mean=[0,0],cov=K)

    plt.figure(figsize=(6,6))
    plt.contour(xm,ym,G, extent=[-3,3,-3,3],cmap='viridis');

def makerot(theta):
    '''This function creates a 2x2 rotation
    matrix for a given angle (theta) in degrees'''

    theta=np.radians(theta)

    R = np.array([[np.cos(theta), -np.sin(theta)],
                  [np.sin(theta), np.cos(theta)]])

    return R
```

In [3]:
```
data = stats.multivariate_normal([0,0],[[1,0.8],[0.8,2]]).rvs(size=100)

data.shape
```

Out[3]:
```
(100, 2)
```

In [4]:
```
plt.scatter(data[:,0],data[:,1])
```

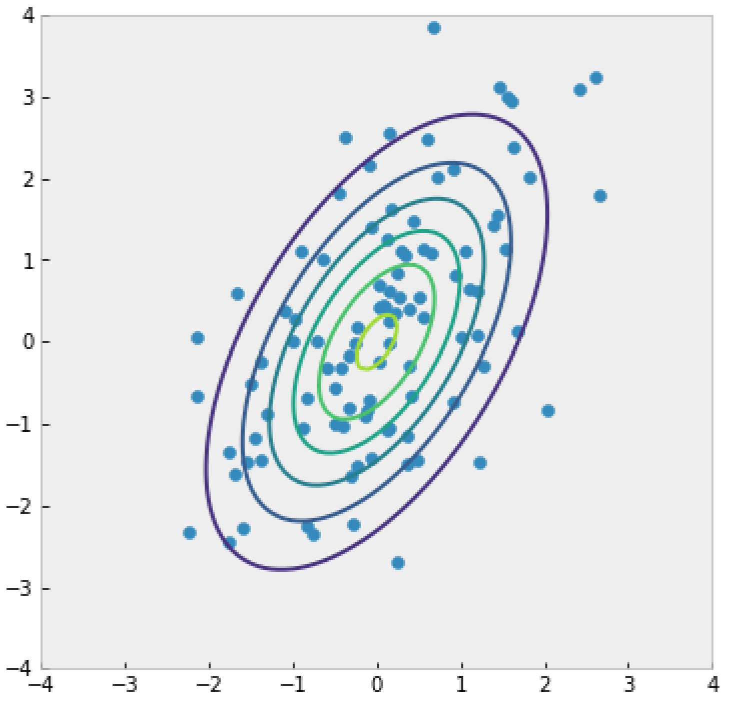
```
Out[4]: <matplotlib.collections.PathCollection at 0x18560468e80>
```



```
In [6]: # Compute covariance matrix

        K = np.cov(data.T)

        # np.cov() it expects the input to be DxN
        K
```

```
Out[6]: array([[1.13986217, 0.87341972],
               [0.87341972, 2.13363429]])
```

```
In [7]: # Plot contours

        plot_contours(K)
        plt.scatter(data[:,0],data[:,1]);
```



```
In [8]: # Eigen-decomposition for Hermitian matrix
        # Hermitian = symmetric and real matrix (all covariances are hermitian matrices)

        L, V = np.linalg.eigh(K)
```

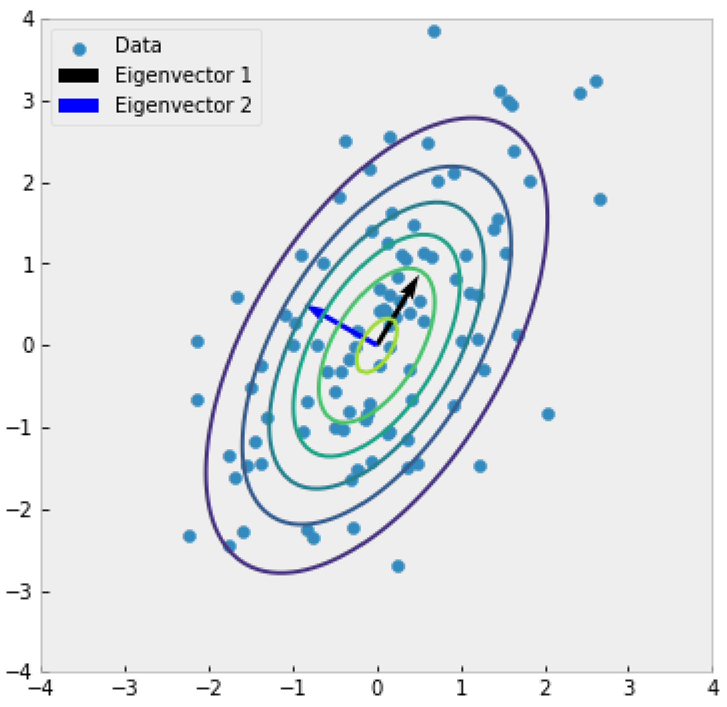
```
L, V
```

Out[8]:
```
(array([0.6318812 , 2.64161527]),
 array([[-0.86443028,  0.50275272],
        [ 0.50275272,  0.86443028]]))
```

In [9]:
```
# Sort eigenvalues and eigenvectors in decreasing order of eigenvalues

L = L[::-1]
V=V[:,::-1]

L, V
```

Out[9]:
```
(array([2.64161527, 0.6318812 ]),
 array([[ 0.50275272, -0.86443028],
        [ 0.86443028,  0.50275272]]))
```

In [10]:
```
plot_contours(K)
plt.scatter(data[:,0],data[:,1])
plotvec(V[:,0], V[:,1])
plt.legend(['Data','Eigenvector 1', 'Eigenvector 2']);
```



In [11]:
```
# Rotate data (uncorrelate)

rotated = V.T@data.T

rotated.shape
```

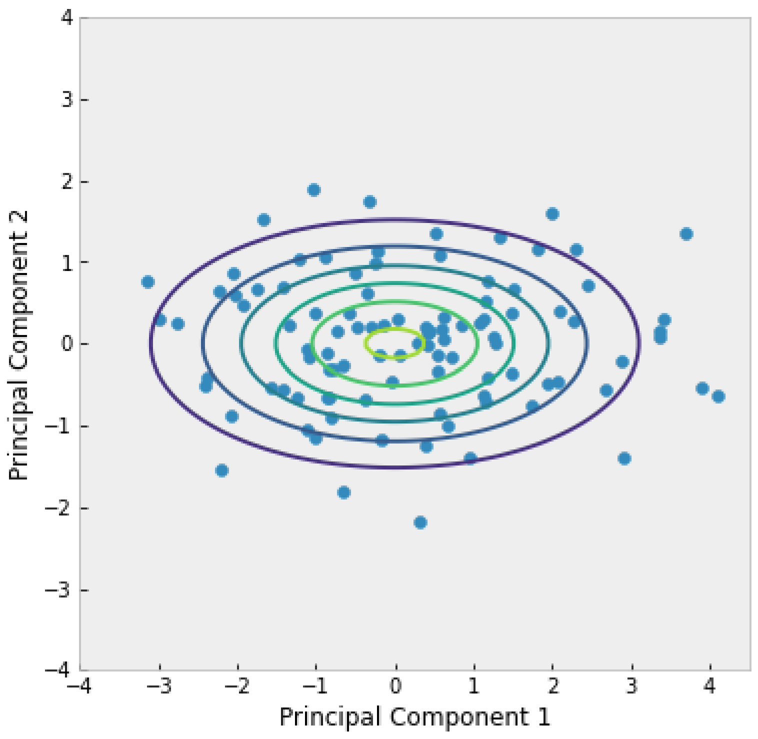
Out[11]:
```
(2, 100)
```

In [12]:
```
# Compute covariance of rotated data

K2 = np.cov(rotated)

np.round(K2,7)
```

```
Out[12]: array([[ 2.6416153, -0.       ],
                [-0.       ,  0.6318812]])
```

The elements in the diagonal correspond to the eigenvalues.

```
In [13]: plot_contours(K2)
         plt.scatter(rotated[0,:],rotated[1,:])
         plt.xlabel('Principal Component 1')
         plt.ylabel('Principal Component 2');
```



```
In [14]: # Angle between standard unit vector [1,0] and 1st eigenvector

         e1=np.array([[1],[0]])

         np.degrees(np.arccos((e1.T@V[:,0])/(np.linalg.norm(e1)*np.linalg.norm(V[:,0]))))

Out[14]: array([59.81771406])
```

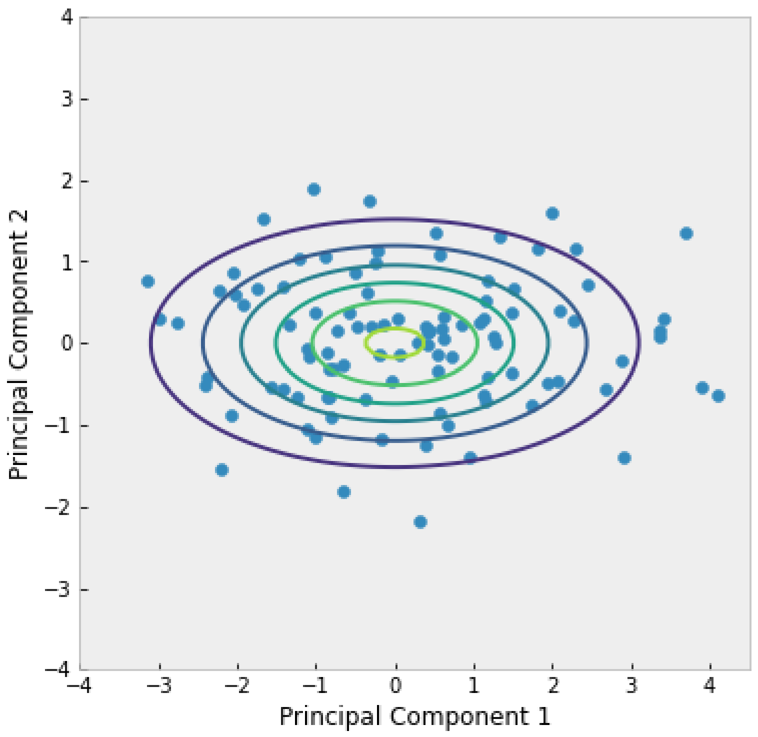
```
In [20]: # Rotate the data by applying rotation matrix

         e1=np.array([[1],[0]])

         angle = np.degrees(np.arccos((e1.T@V[:,0])/(np.linalg.norm(e1)*np.linalg.norm(V[:,0]))
         angle

Out[20]: array([59.81771406])
```

```
In [21]: # Covariance of resulting rotation

         rotated2 = makerot(-angle[0])@data.T
```

```
In [23]: K3 = np.cov(rotated2)

         np.round(K3,7)
```

```
Out[23]: array([[ 2.6416153, -0.        ],
                [-0.        ,  0.6318812]])
```

```
In [24]: plot_contours(K3)
         plt.scatter(rotated2[0,:],rotated2[1,:])
         plt.xlabel('Principal Component 1')
         plt.ylabel('Principal Component 2');
```


Principal Component 2
Principal Component 1