EE 610, ST: ML Fundamentals

Unsupervised Learning

Dr. W. David Pan

Dept. of ECE

UAH

Topics

- Cluster analysis
- K-means algorithm
- K-medoids method
- Singular Value Decomposition
- Principal Component Analysis
- Implementations

Clustering and Dimensionality Reduction

- Unsupervised learning is a conceptually different problem to supervised learning. Unsupervised learning is useful when you want to explore your data but don't yet have a specific goal or are not sure what information the data contains.
  - we cannot hope to perform regression -- we do not know the outputs for any data points, so we cannot guess what the function is.
  - The aim of classification is to identify similarities between inputs that belong to the same class. However, there is not any information about the correct classes.
- What if an algorithm can exploit similarities between inputs in order to cluster inputs that are similar together? This might perform classification automatically.
- So unsupervised learning can be used to find clusters of similar inputs in the data, by discovering the similarities automatically.
- It's also a good way to reduce the dimensions of the data using unsupervised learning.

Cluster Analysis

- In cluster analysis, data is partitioned into groups based on some measure of similarity or shared characteristic.
- Clusters are formed so that objects in the same cluster are very similar and objects in different clusters are very distinct.
- The k-Means algorithm is a well-known method for cluster analysis.
  - We partition data into k number of mutually exclusive clusters.
  - How well a point fits into a cluster is determined by the distance from that point to the cluster's center.

k-Means Clustering Method

- A distance measure
  - In order to measure distances between points, we need to define distances. While the Euclidean distance is often used, there are other alternatives (e.g., city block, cosine, etc.)
- The center of each cluster
  - Once we have a distance measure, we can compute the central point of a set of data points.
  - The mean average (centroid) is used as the central point when Euclidean distance is used, since doing so is equivalent to minimizing the Euclidean distance (which is the sum-of-squares error) from each data point in each cluster to its center.

Algorithm

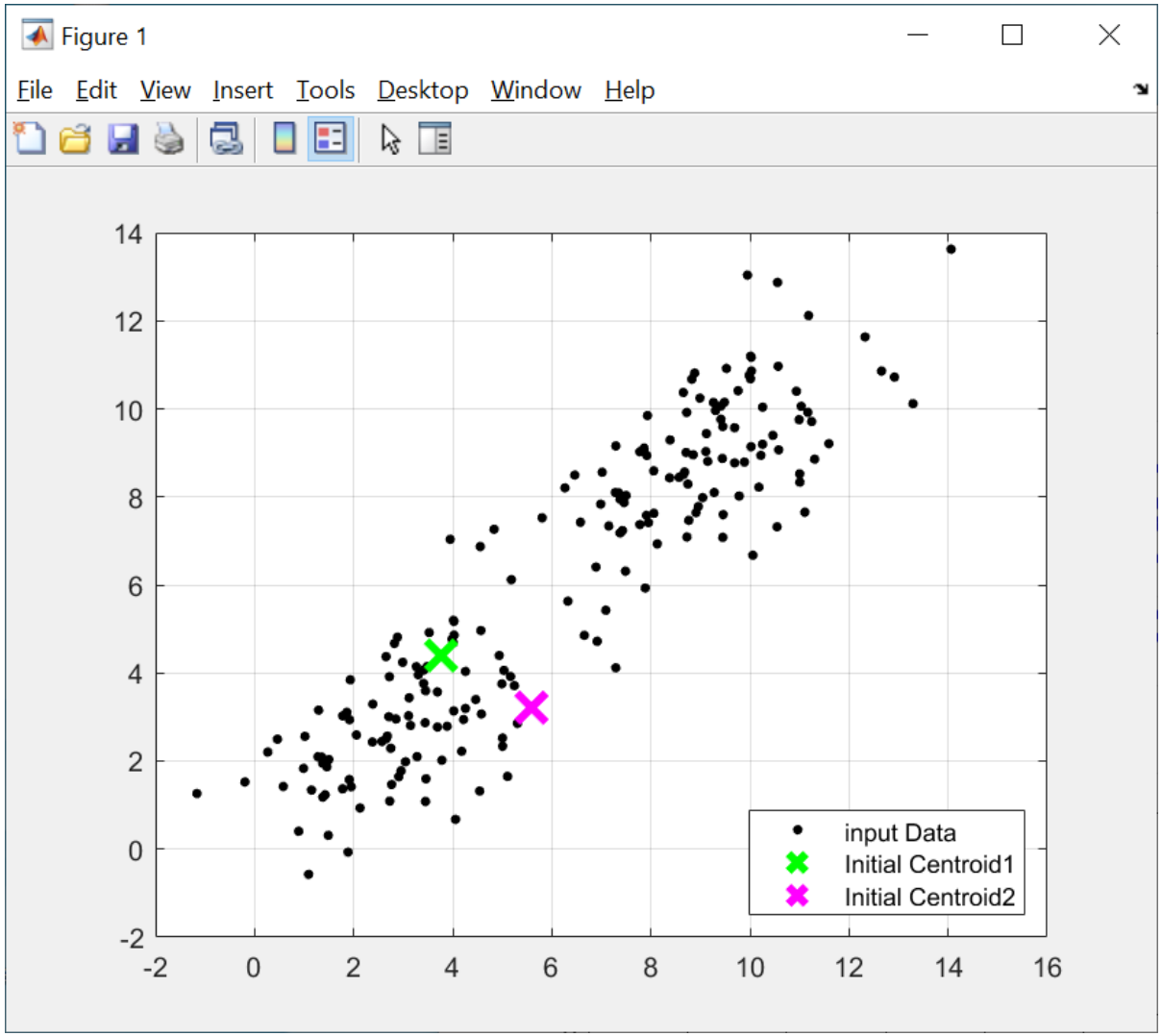
- Initialization
  - Choose a value for k.
  - Choose k random positions in the input space.
  - Assign the cluster centers $\boldsymbol{\mu}_j$ to those positions.
- Learning
  - For each data point $\mathbf{x}_i$:
    - Compute the distance to each cluster center
    - Assign the data point to the nearest cluster center with distance $d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j)$.
  - For each cluster center:
    - Move the position of the center to the mean of the data points in that cluster: $\boldsymbol{\mu}_j = \frac{1}{N_j}\sum_{i=1}^{N_j} \mathbf{x}_i$, where N_j is the number of points in cluster j.
  - Repeat the above steps until the cluster centers stop moving.
- Usage
  - For each test data point:
    - Compute the distance to each cluster center.
    - Assign the data point to the nearest cluster center with distance $d_i = \min_j d(\mathbf{x}_i, \boldsymbol{\mu}_j)$.

```
% 'two_means.m'

N = 100;

m1 = [3, 3]';   % Mean vector
cov1 = [2 1; 1 2]; % Cov matrix
rng default
r1 = mvnrnd(m1,cov1,N);

data_C1 = zeros(N, 2);
data_C1 = r1;

m2 = [9, 9]';
cov2 = [2,1; 1,2];
rng default
r2 = mvnrnd(m2,cov2,N);

data_C2 = zeros(N, 2);
data_C2 = r2;

X = vertcat (data_C1, data_C2);

G1 = zeros(2*N, 2);
G2 = zeros(2*N, 2);

n_iter = 1;
```

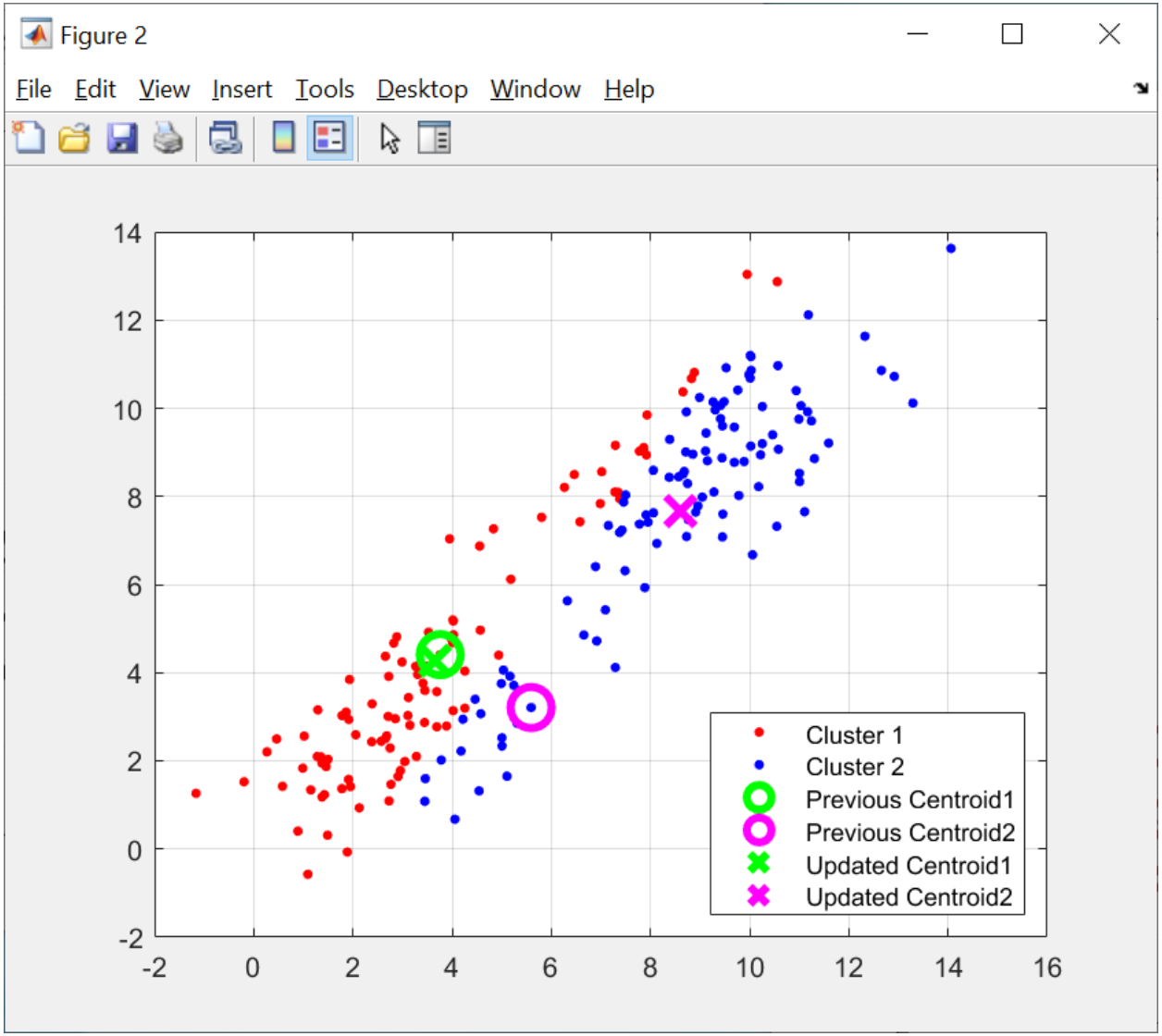
```
while 1
    n_iter = n_iter + 1;

    N1 = 0;
    N2 = 0;
    for i = 1: 2*N
        d1 = pdist2(X(i,:),mu1);
        d2 = pdist2(X(i,:),mu2);
        if (d1 <= d2)
            N1 = N1 + 1;
            G1(N1,:) = X(i,:);
        else
            N2 = N2 + 1;
            G2(N2,:) = X(i,:);
        end
    end
    mu1_prev = mu1;
    mu2_prev = mu2;

    G1_new = G1(1:N1,:);
    G2_new = G2(1:N2,:);

    mu1 = mean(G1_new)
    mu2 = mean(G2_new)
    if (mu1 == mu1_prev & mu2 == mu2_prev)
        break;
    end
end
```

Initial Centroids

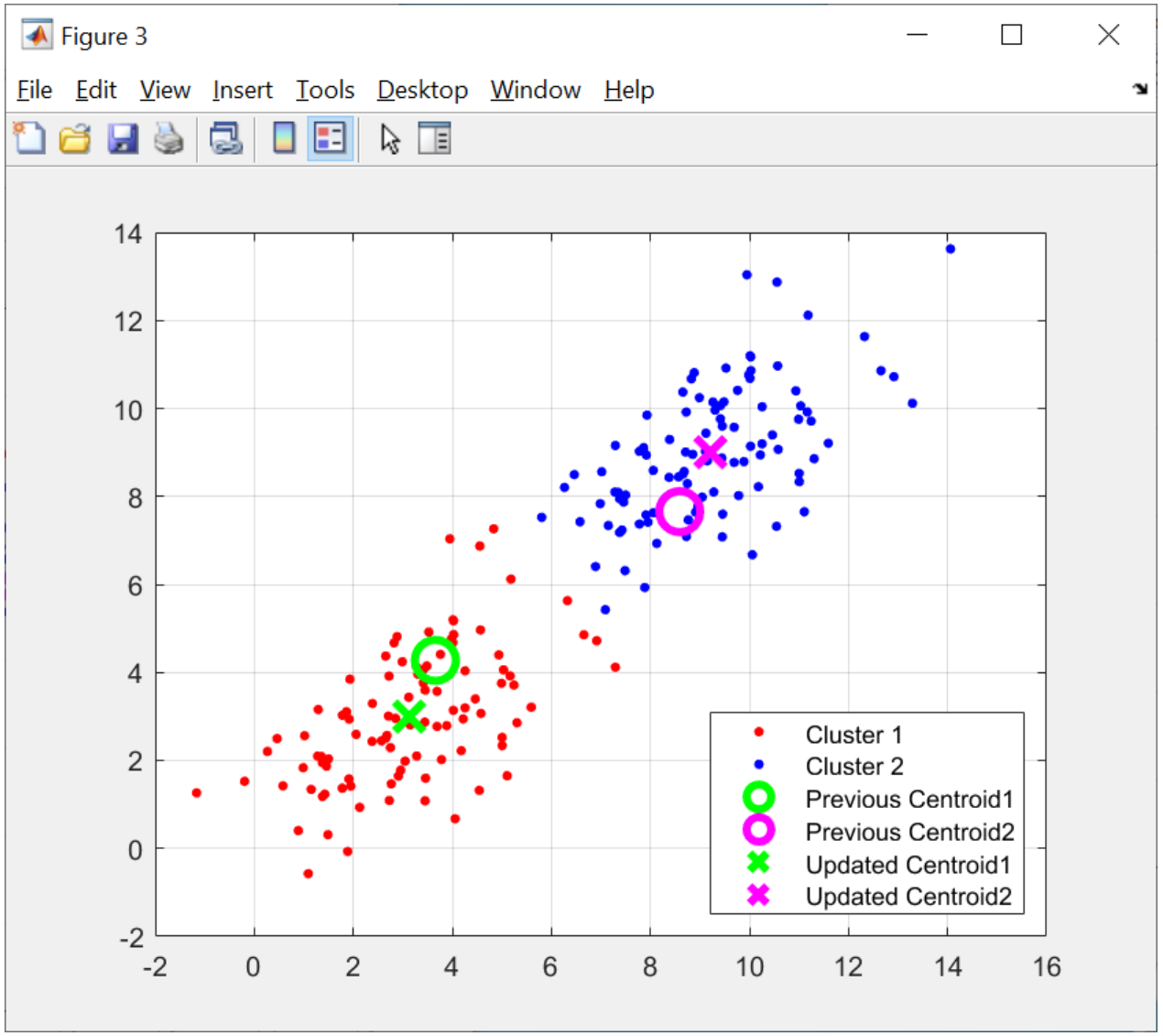
```
% Initial centroids
mu1 = X(1,:)
mu2 = X(2,:)
```



After 1st Iteration

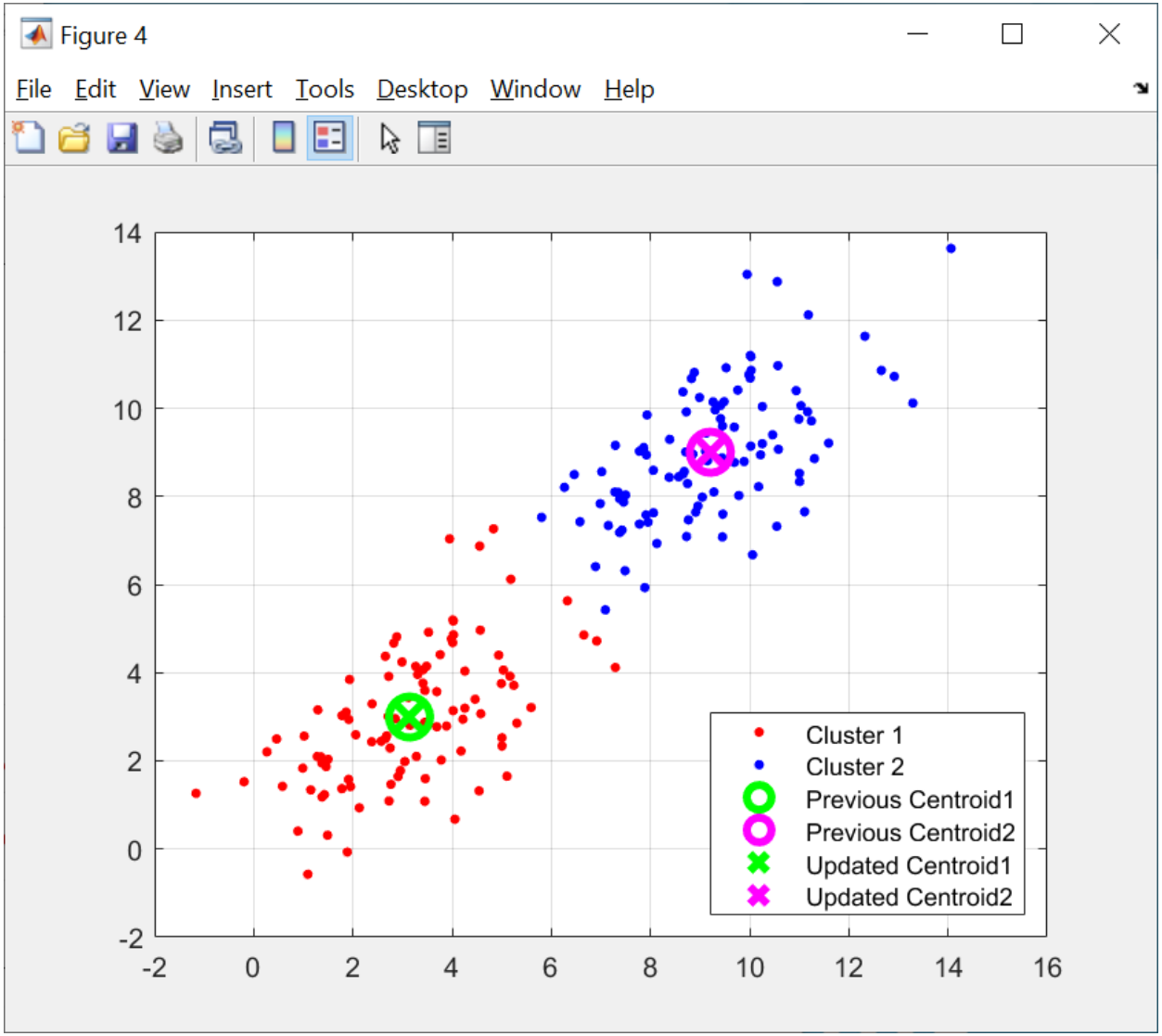


After 2nd Iteration



Convergence after 3rd Iteration

mu1 =
3.1418 2.9944
mu2 =
9.2063 9.0017

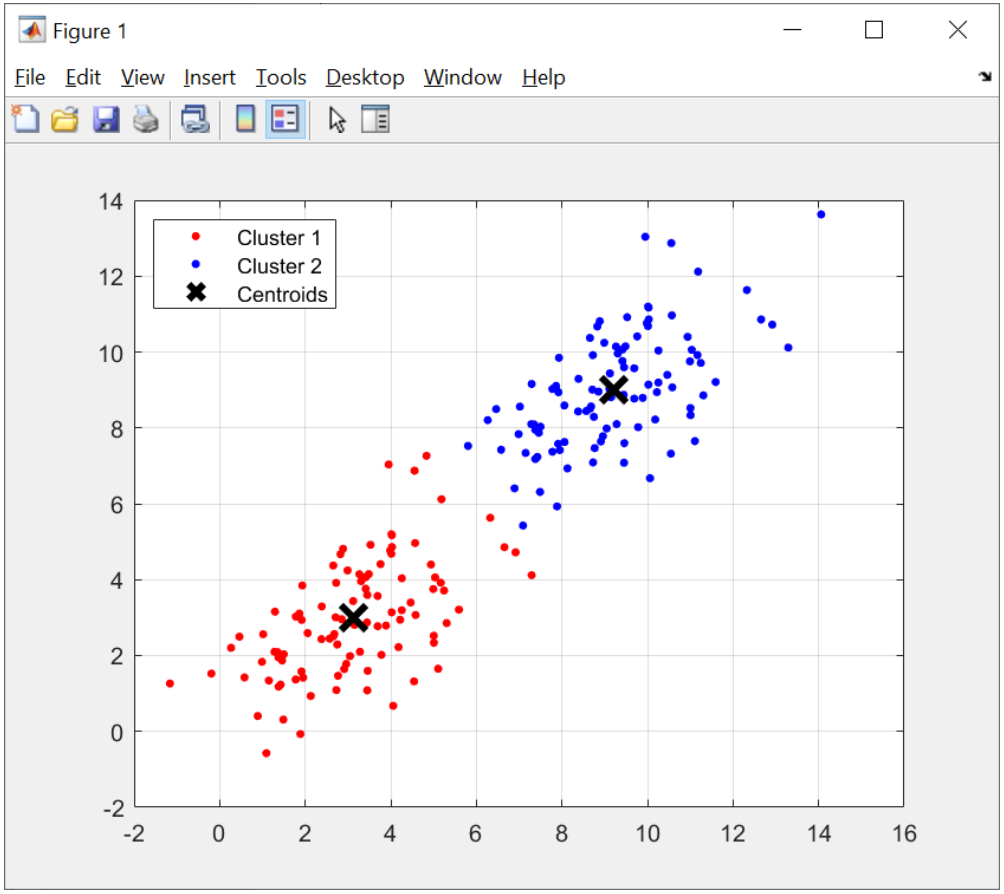


Training and Testing with **kmeans**

'kmeans_demo.m'

```
[idx,C] = kmeans(X,2);

% Display the clusters
plot(X(idx==1,1),X(idx==1,2),'r.',
'MarkerSize',12)
hold on
plot(X(idx==2,1),X(idx==2,2),'b.',
'MarkerSize',12)

% Display the centroids
plot(C(:,1),C(:,2),'kx',...
'MarkerSize',15,'LineWidth',3)
legend('Cluster 1','Cluster
2','Centroids',...'Location','NW')

% Testing
```

Xtest = [6 4];
[dist, **cluster_index**] = **pdist2**
(**C**,Xtest,'euclidean','Smallest',1);
>> pdist2(Xtest, C(1,:))
>> pdist2(Xtest, C(2,:))



>> mean(data_C1)
ans =
3.1741 2.9980
>> mean(data_C2)
ans =
9.1741 8.9980

>> C
C =
3.1418 2.9944
9.2063 9.0017

Distance Measure

Distance Metric	Description	Formula
`'sqeuclidean'`	Squared Euclidean distance (default). Each centroid is the mean of the points in that cluster.	$d(x,c) = (x-c)(x-c)'$

```
[idx,C,sd] = kmeans(X,2);
>> sd
sd =
  480.7163
  475.0740

>> d1 = pdist2(X(idx==1,:), C(1,:)).^2;
   d2 = pdist2(X(idx==2,:), C(2,:)).^2;
>> sum(d1)
ans =
  480.7163
>> sum(d2)
ans =
  475.0740
```

sklearn

Matlab:

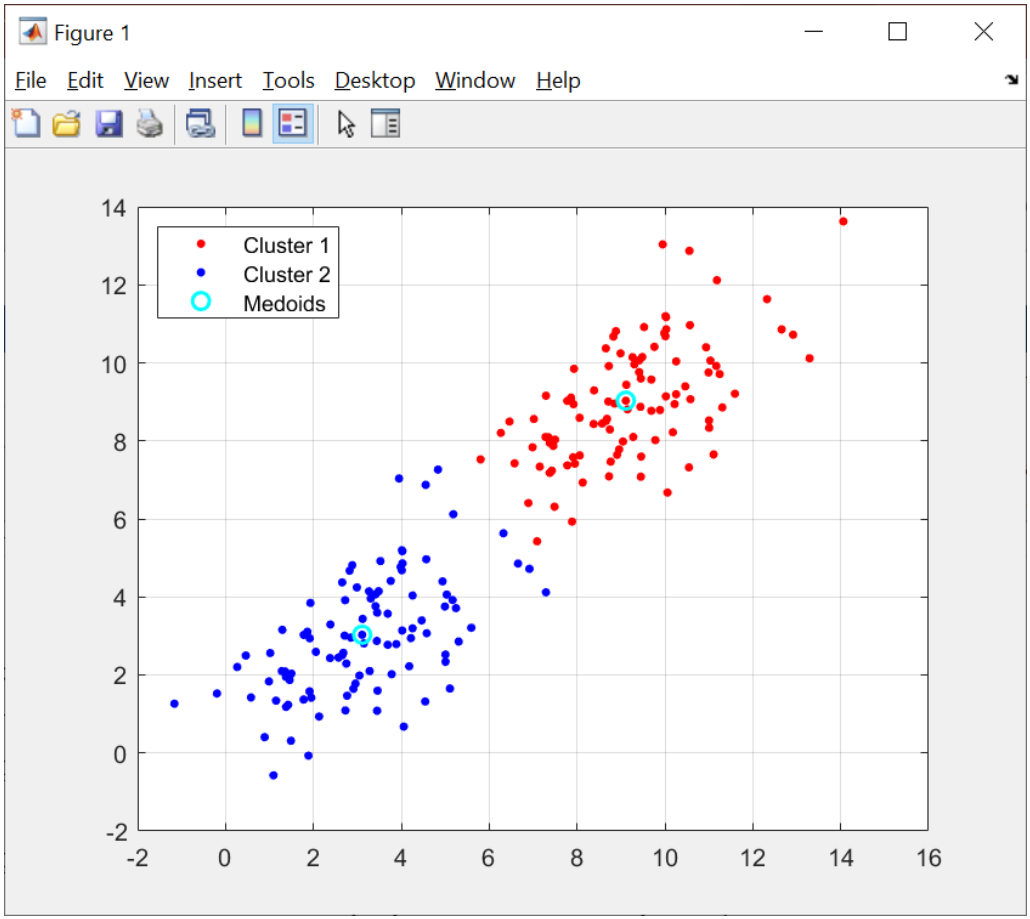
```
writematrix (X, 'kmeans.csv');
```

```
import numpy as np
infile = r"C:\...\kmeans.csv"
dataset = np.loadtxt(infile, delimiter=',')
X = dataset[:, 0:2]

from sklearn.cluster import KMeans

cluster = KMeans(n_clusters=2, random_state=0).fit(X)

cluster.cluster_centers_
:array([[9.20631359, 9.00165323],
        [3.14182431, 2.99438324]])

xtest = [[6,4]]
cluster.predict(xtest)
:array([1])

xtest = [[6,8]]
cluster.predict(xtest)
:array([0])
```

K-medoids Method

- k-medoids clustering is a partitioning method commonly used in domains that require robustness to outlier data, or ones for which the mean does not have a clear definition.
- It is similar to k-means, and the goal k-methods is to divide a set of measurements or observations into k subsets or clusters so that the subsets minimize the sum of distances between a measurement and a center of the measurement's cluster.
  - In the k-means algorithm, the center of the subset is the mean of measurements in the subset, often called a centroid.
  - In the k-medoids algorithm, the center of the subset is a member of the subset, called a medoid.
- The k-medoids algorithm returns medoids which are the actual data points in the data set. This allows us to use the algorithm in situations where the mean of the data does not exist within the data set.
- Thus the main difference between k-medoids and k-means is that the centroids returned by k-means may not be within the data set.

```
[idx,C] = kmedoids(X,2);

% Display the clusters
plot(X(idx==1,1),X(idx==1,2),'r.
','MarkerSize',12)
hold on
plot(X(idx==2,1),X(idx==2,2),'b.
','MarkerSize',12)

plot(C(:,1),C(:,2),'co',...
 'MarkerSize',7,'LineWidth',1.5)
grid

legend('Cluster 1','Cluster
2','Medoids',...
       'Location','NW');
```



Medoids

```
>> C
C =
    9.1094    9.0302
    3.1094    3.0302
```

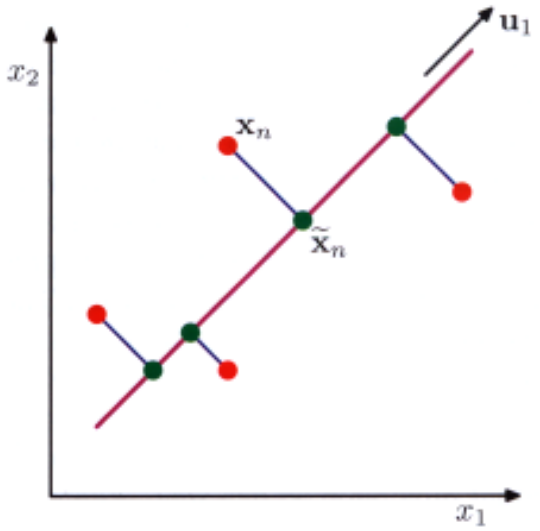
```
>> find(X(:,1)==C(1,1) & X(:,2) == C(1,2))
ans =
   152
>> X(152,:)
ans =
    9.1094    9.0302
```

```
>> find(X(:,1)==C(2,1) & X(:,2) ==
C(2,2))
ans =
    52
>> X(52,:)
ans =
    3.1094    3.0302
```

PCA

- Principal Component Analysis, or PCA, can help us get a good understanding of the variance in the multivariate data.
- Suppose input data have m features, which are random variables $X_1, X_2, \ldots X_m$.
- While the features are all correlated to varying degrees, PCA changes the variables into $Y_1, Y_2, \ldots Y_m$, which are linear combinations of the Xs.
- The first several **principal** components Y_k (sometimes called "**latent factors**") "explains" a big chunk of the variance of the original variables, with the remaining components becoming somewhat meaningless.

- In this two-feature example, PCA seeks a space of lower dimensionality, known as the principal subspace, as denoted by the magenta line, such that the orthogonal projection of the data points (red dots) onto this subspace maximizes the variance of the projected points (green dots).



- PCA is widely used for applications such as dimensionality reduction (from m down to k), lossy data compression, feature extraction, and data visualization.

The idea of PCA

- In the PCA algorithm, we first center the data by subtracting off the mean.
- Next, we choose the direction with the largest variation and places an axis in that direction, along which we project the data.
- We then look at the variation that remains, and finds another axis that is orthogonal to the first and covers as much of the remaining variation as possible.
- Repeat the above procedure until we run out of possible axes.
- All the variation is along the axes chosen, since the covariance matrix becomes diagonal after data projection — each new variable is uncorrelated with other variables.
- Some of the axes that are found last have very little variation, and so they can be removed without affecting much the variability in the data.
- Thus PCA can be used for lossy data compression, dimensionality reduction and feature selection for supervised learning.

Theory

- Suppose the centralized input data $\mathbf{X}$ is a $N \times m$ matrix. That is, the input consists of N samples, with each sample is a point in a m-dimensional space (or with m components).
- Let the covariance matrix of $\mathbf{X}$ be $\mathbf{C} = E[\mathbf{X}^{\mathrm{T}}\mathbf{X}]$, where $\mathbf{C}$ is a $m \times m$ matrix. It follows $\mathbf{C}^{\mathrm{T}} = \mathbf{C}$.
- Since $\mathbf{C}$ is a symmetric matrix, any two of its eigenvectors, $\mathbf{u}_1$ and $\mathbf{u}_2$, corresponding to distinct eigenvalues λ_1 and λ_2 are orthogonal:

$$\mathbf{C}\mathbf{u}_1 = \lambda_1\mathbf{u}_1$$

$$(\mathbf{C}\mathbf{u}_1)^{\mathrm{T}} = (\lambda_1\mathbf{u}_1)^{\mathrm{T}} \longrightarrow \mathbf{u}_1^T\mathbf{C}^T = \lambda_1\mathbf{u}_1^{\mathrm{T}}$$

$$\mathbf{C}\mathbf{u}_2 = \lambda_2\mathbf{u}_2$$

$$\mathbf{u}_1^T\mathbf{C} = \lambda_1\mathbf{u}_1^{\mathrm{T}} \longrightarrow \mathbf{u}_1^{\mathrm{T}}(\mathbf{C}\mathbf{u}_2) = \lambda_1\mathbf{u}_1^{\mathrm{T}}\mathbf{u}_2$$

$$\mathbf{u}_1^{\mathrm{T}}\lambda_2\mathbf{u}_2 = \lambda_1\mathbf{u}_1^{\mathrm{T}}\mathbf{u}_2$$

$$\mathbf{u}_1^{\mathrm{T}}\mathbf{u}_2 = 0$$

Variance of Projections

- $\mathbf{CV} = \mathbf{VD}$, where $\mathbf{V}$ is the eigenvector matrix consisting of m normalized eigenvectors, $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m]$, $\mathbf{v}_i^{\mathrm{T}}\mathbf{v}_i = 1$.
- $\mathbf{D}$ is a diagonal matrix consisting of the corresponding m eigenvalues λ_i, such that $\mathbf{C}\mathbf{v}_i = \lambda_i \mathbf{v}_i$.
- If we project the data $\mathbf{X}$ to the axis given by the eigenvector $\mathbf{v}_i$, then we generate the $N \times 1$ "scores" vector $\mathbf{Y} = \mathbf{X}\mathbf{v}_i$.
- The variance of the projection is the corresponding eigenvalue:

$$E\left[\mathbf{Y}_{\mathrm{i}}^{\mathrm{T}}\mathbf{Y}_{\mathrm{i}}\right] = E[(\mathbf{X}\mathbf{v}_i)^{\mathrm{T}}\mathbf{X}\mathbf{v}_i] = E[\mathbf{v}_i{}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{v}_i] = \mathbf{v}_i{}^{\mathrm{T}}E[\mathbf{X}^{\mathrm{T}}\mathbf{X}]\mathbf{v}_i$$

$$= \mathbf{v}_i{}^{\mathrm{T}}\mathbf{C}\mathbf{v}_i = \mathbf{v}_i{}^{\mathrm{T}}\lambda_i\mathbf{v}_i = \lambda_i\mathbf{v}_i{}^{\mathrm{T}}\mathbf{v}_i = \lambda_i$$

- The correlation of the projections using different eigenvectors ($i \neq j$):

$$E\left[\mathbf{Y}_{\mathrm{i}}^{\mathrm{T}}\mathbf{Y}_{\mathrm{j}}\right] = E\left[(\mathbf{X}\mathbf{v}_i)^{\mathrm{T}}\mathbf{X}\mathbf{v}_j\right] = E\left[\mathbf{v}_i{}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\mathbf{X}\mathbf{v}_j\right] = \mathbf{v}_i{}^{\mathrm{T}}E[\mathbf{X}^{\mathrm{T}}\mathbf{X}]\mathbf{v}_j$$

$$= \mathbf{v}_i{}^{\mathrm{T}}\mathbf{C}\mathbf{v}_j = \mathbf{v}_i{}^{\mathrm{T}}\lambda_i\mathbf{v}_j = \lambda_i\mathbf{v}_i{}^{\mathrm{T}}\mathbf{v}_j = 0$$

- We can choose the k principal components to go along with the k eigenvectors, corresponding to k largest eigenvalues, after sorting the eigenvalues in an descending order.

Singular Value Decomposition

- The eigenvalues and eigenvectors can also be determined by the SVD method.
- In linear algebra, the singular value decomposition (SVD) is a factorization of a matrix.
- It decompose a $N \times m$ matrix $\mathbf{X}$ with an orthonormal eigenbasis.
- $\mathbf{X} = \mathbf{ASB}^{\mathrm{T}}$, where $\mathbf{A}$ is an $N \times N$ orthogonal matrix, that is $\mathbf{AA}^{\mathbf{T}} = \mathbf{A}^{\mathbf{T}}\mathbf{A} = \mathbf{I}$, $\mathbf{S}$ is an $N \times m$ rectangular diagonal matrix with m non-negative real numbers on the diagonal, and $\mathbf{B}$ is an $m \times m$ orthogonal matrix.
- In Compact SVD, $\mathbf{X} = \mathbf{ASB}^{\mathrm{T}}$, where $\mathbf{A}$ is an $N \times r$ matrix, $\mathbf{S}$ is an $r \times r$ diagonal matrix with r non-zero singular values, and $\mathbf{B}$ is an $m \times r$ matrix. Both a and are semi-orthogonal matrices, that is, $\mathbf{A}^{\mathbf{T}}\mathbf{A} = \mathbf{I}_{\mathrm{r}}$, and $\mathbf{B}^{\mathbf{T}}\mathbf{B} = \mathbf{I}_{\mathrm{r}}$.

Singular Values and Eigenvalues

- $\mathbf{X} = \mathbf{A}\mathbf{S}\mathbf{B}^{\mathrm{T}}$, thus

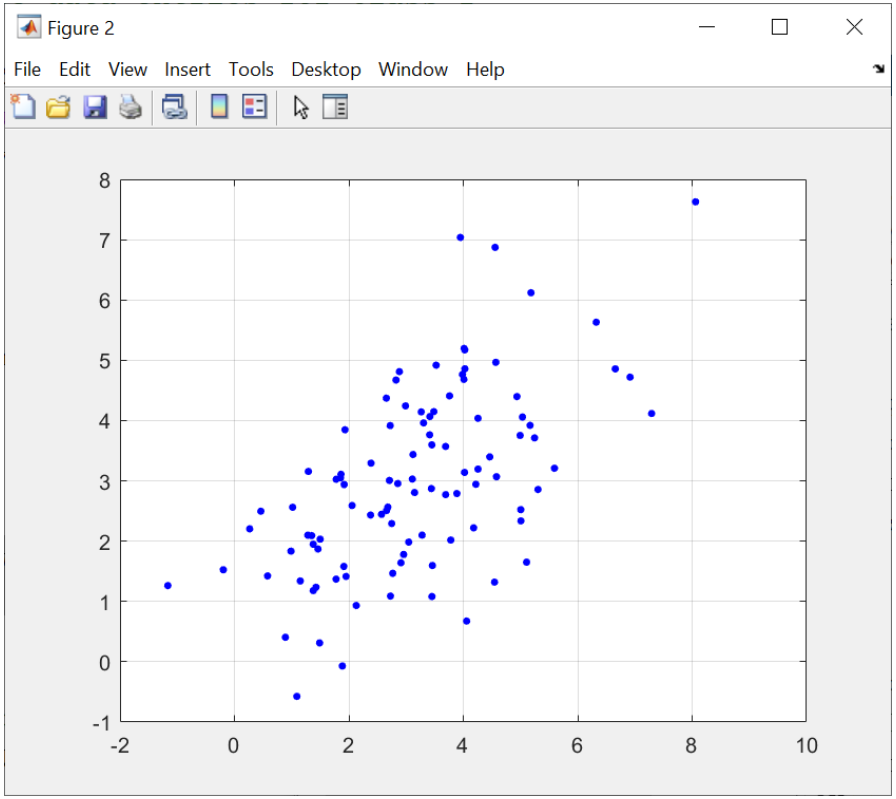
$$\mathbf{X}^{\mathrm{T}}\mathbf{X} = \left(\mathbf{A}\mathbf{S}\mathbf{B}^{\mathrm{T}}\right)^{\mathrm{T}}\mathbf{A}\mathbf{S}\mathbf{B}^{\mathrm{T}} = \mathbf{B}\mathbf{S}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{S}\mathbf{B}^{\mathrm{T}}$$

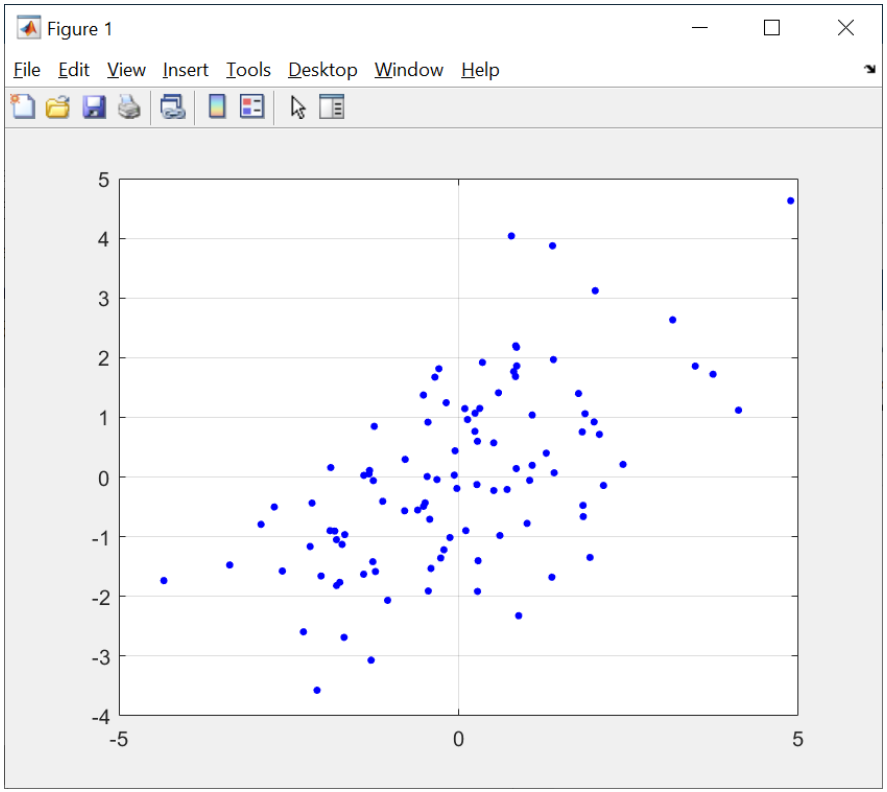
- Since $\mathbf{A}^{\mathrm{T}}\mathbf{A} = \mathbf{I}$, $\mathbf{X}^{\mathrm{T}}\mathbf{X} = \mathbf{B}\mathbf{S}^{\mathrm{T}}\mathbf{S}\mathbf{B}^{\mathrm{T}} = \mathbf{B}\boldsymbol{\Sigma}\mathbf{B}^{\mathrm{T}}$, where $\boldsymbol{\Sigma}$ is a $m \times m$ diagonal matrix consisting the square of the singular values at the diagonal of $\mathbf{S}$.
- Since $\mathbf{B}^{\mathrm{T}}\mathbf{B} = \mathbf{I}$, $(\mathbf{X}^{\mathrm{T}}\mathbf{X})\mathbf{B} = \mathbf{B}\boldsymbol{\Sigma}\mathbf{B}^{\mathrm{T}}\mathbf{B} = \mathbf{B}\boldsymbol{\Sigma}$, thus
- $\mathbf{B}$ is the eigenvector matrix of $(\mathbf{X}^{\mathrm{T}}\mathbf{X})$, and $\boldsymbol{\Sigma}$ is the diagonal matrix of eigenvectors of $(\mathbf{X}^{\mathrm{T}}\mathbf{X})$.

PCA Algorithm

- Write N data points $\boldsymbol{x}_i = (\boldsymbol{x}_{i1}, \boldsymbol{x}_{i2}, \ldots, \boldsymbol{x}_{im})$ as row vectors.
- Put these vectors into a matrix $\mathbf{X}_{\mathrm{raw}}$ (which will have size $N \times m$)
- Center the data by subtracting off the mean of each column, putting it into matrix $\mathbf{X}$.
- Compute the covariance matrix $\mathbf{C} = cov(\mathbf{X})$.
- Compute the eigenvalues and eigenvectors of $\mathbf{C}$, so $\mathbf{CV} = \mathbf{VD}$, where $\mathbf{V}$ holds the eigenvectors of $\mathbf{C}$, and $\mathbf{D}$ is the $m \times m$ diagonal eigenvalue matrix.
- Sort the columns of $\mathbf{D}$ into order of decreasing eigenvalues, and apply the same order to the columns of $\mathbf{V}$.
- Choose first k principal components and project the data onto the corresponding eigenvectors.

Centering the Raw Input Data





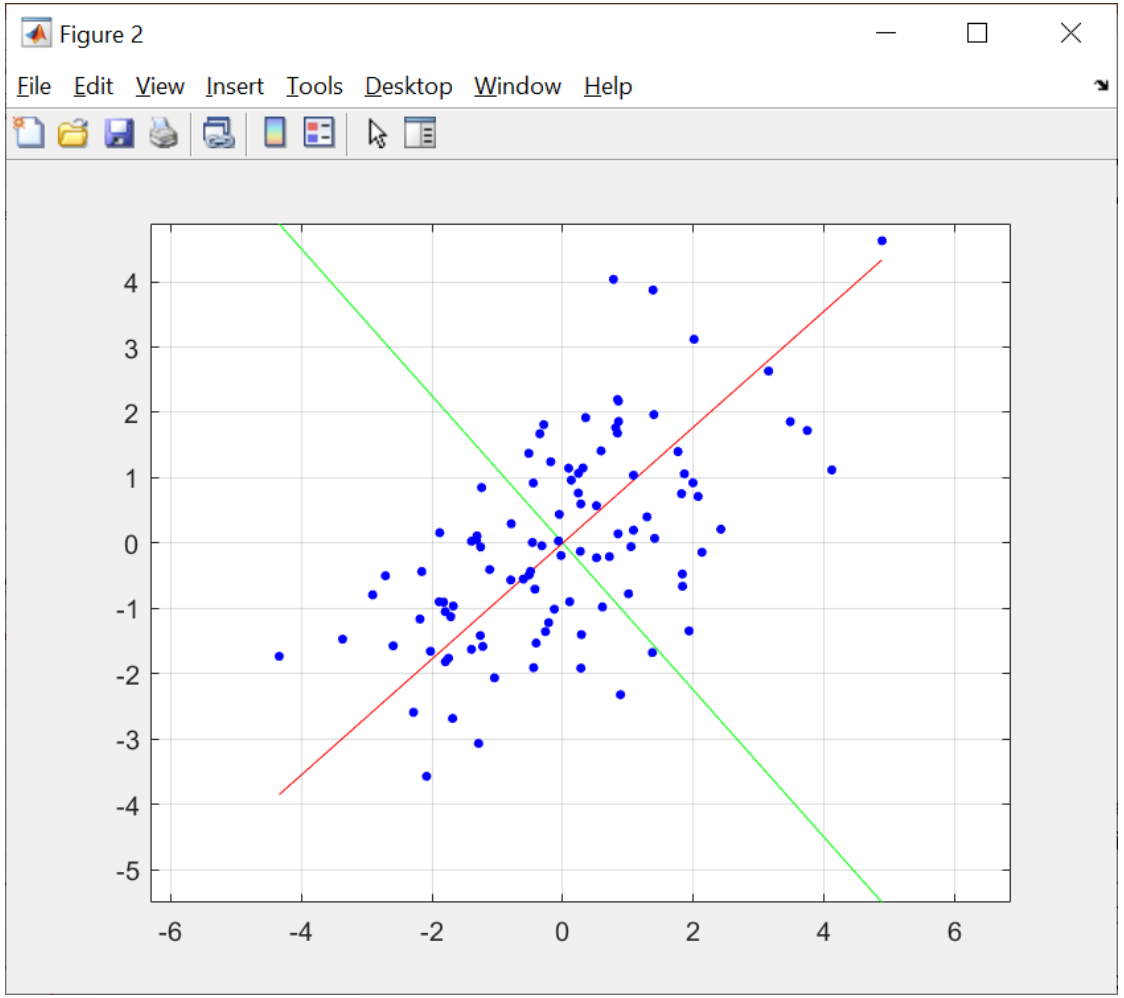

```
N = 100;
m1 = [3, 3]';
cov1 = [2 1; 1 2];
rng default
Xraw = mvnrnd(m1,cov1,N);
```

```
% Center the data
mu = mean(Xraw);
X = Xraw - mu;
mean(X)
```

Unit eigenvectors are perpendicular

```
% Empirical sample
covariance matrix
(X'*X)/(N-1)
% Same as using cov()
C = cov(X)

% Find out the eigenvalues
and eigenvectors
[V,D] = eig(C);
C*V
V*D

V1 = V(:,1); norm(V1)
V2 = V(:,2); norm(V2)
V1'*V2
```

```
% Display the eigenvectors for data projection
hor = min(X(:,1)): 0.01: max(X(:,1));
ver = V1(2)/V1(1)*hor;
plot(hor, ver, 'g');

hold on;
hor = min(X(:,1)): 0.01: max(X(:,1));
ver = V2(2)/V2(1)*hor;
plot(hor, ver, 'r');
```



D =

1.0084	0
0	4.0373

V =

0.6639	-0.7478
-0.7478	-0.6639

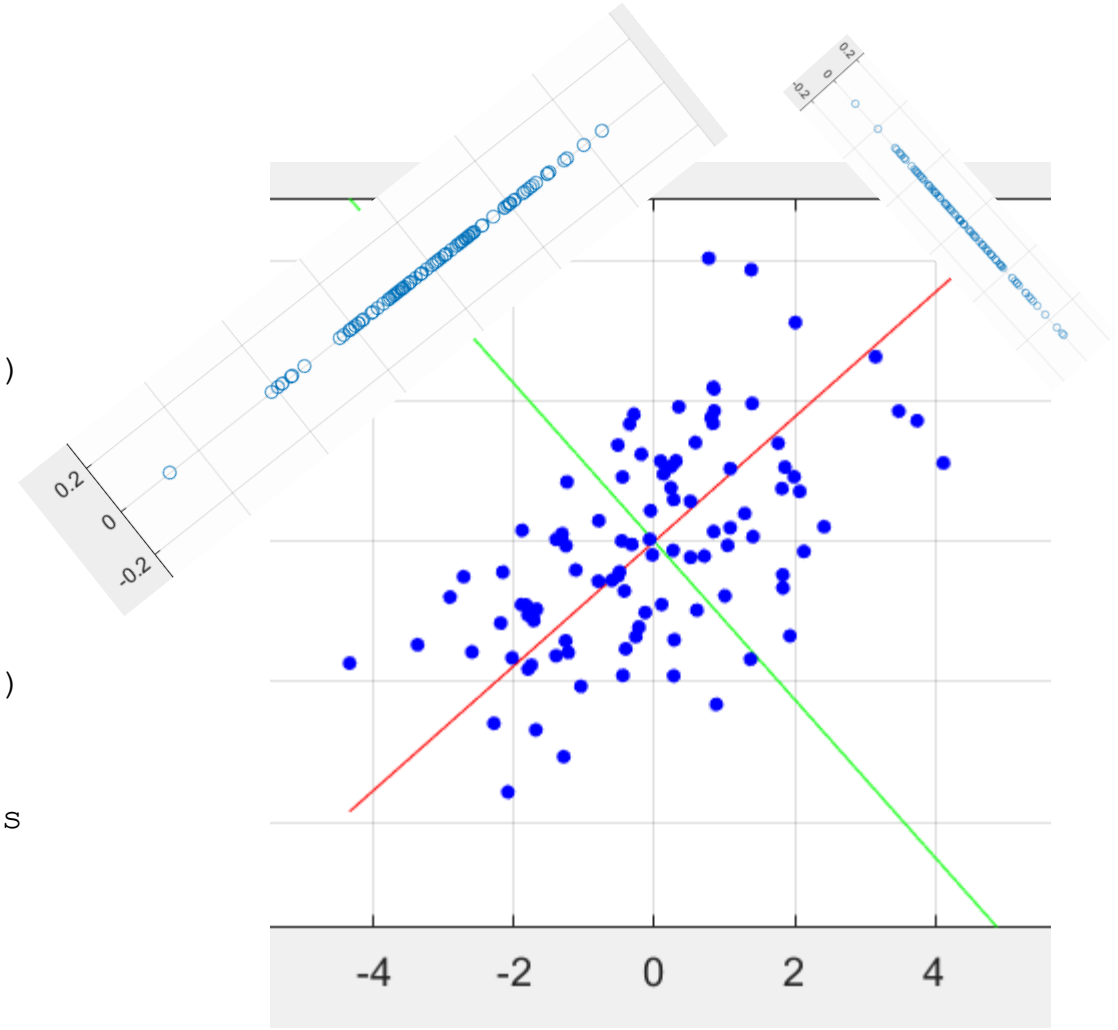
SVD

```
[A,S,B]= svd(X/sqrt(N-1),'econ');
>> whos A
  Name        Size            Bytes  Class     Attributes
  A         100x2              1600  double

S =
    2.0093         0
         0    1.0042

>> S.^2
ans =
    4.0373         0
         0    1.0084

>> B
B =
   -0.7478   -0.6639
   -0.6639    0.7478
```

```
[V,D] = eig(C);

D =
    1.0084         0
         0    4.0373

V =
    0.6639   -0.7478
   -0.7478   -0.6639
```

```
D =
    1.0084         0
         0    4.0373
```

```
% Variance after projection
Y1 = X*V1;
var(Y1)
ans =
    1.0084
figure;  scatter(Y1,zeros(100,1))

Y2 = X*V2;
var(Y2)
>> var(Y2)
ans =
    4.0373
figure;  scatter(Y2,zeros(100,1))
grid

% Correlation between projections
>> Y1'*Y2
ans =
  -9.9476e-14
```



The **pca** function in Matlab

```
>> [coeff,score,latent] = pca(Xraw);
coeff =
    0.7478   -0.6639
    0.6639    0.7478
```

```
V =
    0.6639   -0.7478
   -0.7478   -0.6639
```

- Each column of coeff contains coefficients (indicating projection direction vectors) for one principal component, and the columns are in descending order of component variance.
- Coefficients are sometimes called "**loadings**", which are the coefficients (or weights) of the linear combination (weighted average) of the original variables from which the principal components (PCs) are constructed.

```
>> whos score
  Name        Size            Bytes  Class     Attributes
  score       100x2            1600  double
```

```
>> cov(score)
ans =
    4.0373    0.0000
    0.0000    1.0084
```

- Scores are the value of projection of the input data onto a eigenvector (or the results of the linear combination of the original variables).

```
>> latent
latent =
    4.0373
    1.0084
```

Latent: Principal component variances, that is the eigenvalues of the covariance matrix, returned as a column vector.

Loadings and Scores

```
>> X(1,:)
ans =
    0.5863    1.4114
```

```
>> coeff
coeff =
    0.7478   -0.6639
    0.6639    0.7478
```

```
>> inv(coeff)
ans =
    0.7478    0.6639
   -0.6639    0.7478
```

```
>> X(1,:)*coeff(:,1)
ans =
    1.3755
```

```
>> X(1,:)*coeff(:,2)
ans =
    0.6663
```

```
>> score(1,:)
ans =
    1.3755    0.6663
```

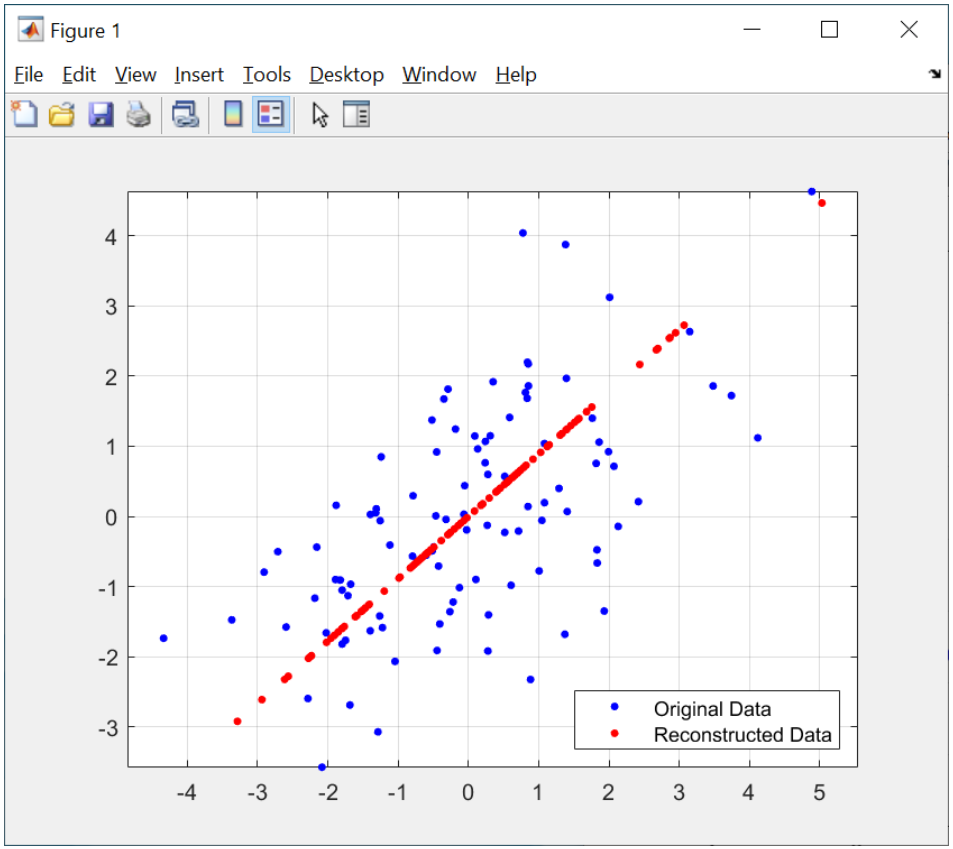
```
>> coeff'*coeff
ans =
    1.0000   -0.0000
   -0.0000    1.0000
```

Reconstruction of X(1,:) using coeff

```
>> score(1,:)*coeff‘    // inv(coeff) = transpose(coeff)
ans =
    0.5863    1.4114
```

Reconstruction using the Principal Component

```
score_truncated = score;
score_truncated(:,2) = 0;
X_rec2 = score_truncated * coeff';
figure;
plot(X(:,1),X(:,2),'b.','MarkerSize',12)
hold on;
plot(X_rec2(:,1),X_rec2(:,2),'r.','Marker
Size',12)
legend('Original Data', 'Reconstructed
Data','Location','SE');
grid
axis equal
```




```
>> diff = (X_rec2 - X);
diff_sq = diff(:,1).^2 + diff(:,2).^2;
% Average distortion (mean square error)
sum(diff_sq)/(N-1)
ans =
   1.0084 (same as the variance of the 2nd
component dropped)
```

```
>> latent
latent =
   4.0373
   1.0084
```

sklearn

```
import numpy as np
infile = r"C:\...\pca.csv"
dataset = np.loadtxt(infile, delimiter=',')
X = dataset[:, 0:2]

from sklearn.decomposition import PCA

pca = PCA(n_components=2)
# sklearn automatically centers the input raw data
pca.fit(X)

# Eigenvectors (loadings)
print(pca.components_)

# Eigvenvalues (latent)
print(pca.explained_variance_)

# Scores
Y = pca.transform(X)

#axis = 0, along the columb; ddof = 1 for dividing by (N-1);
np.var(Y, axis = 0, ddof=1)
```

```
# Reconstruction by keeping only the 1st principal component
# setting the 2nd component in Y to zero
Y_trunc = Y
Y_trunc[:,1] = 0

X_rec = pca.inverse_transform(Y_trunc)

# Centered (instead of the raw) input to compare with the reconstructed data
X_center = X - np.mean(X)

# Mean square error
diff = X_rec - X_center
diff_sq = diff[:,0]**2 + diff[:,1]**2
np.sum(diff_sq)/(np.size(diff_sq)-1)
```