Chapter 12 Unsupervised Learning

– Math 313 Statistics for Data Science

Guangliang Chen

Associate Professor

cheng@hope.edu

Hope College, Fall 2023

Presentation Overview

1. 12.1 The challenge of unsupervised learning
2. 12.2 Principal component analysis (PCA)
3. 12.3 Missing values and matrix completion
4. 12.4 Clustering methods
   - *k*-means clustering
   - Spectral clustering

The challenge of unsupervised learning

Recall that in supervised learning, we are given a set of features $X_1, X_2, \ldots, X_p$, as well as a response Y, that are measured on n observations. The goal is to predict Y using $\vec{X} = (X_1, X_2, \ldots, X_p)$.

In unsupervised learning, we only have a set of features $\vec{X}$ measured on n observations. The goal is to learn relationships among the features and/or discover patterns in the data.

Since there is no response, unsupervised learning is more challenging than supervised learning.

Two very common tasks in unsupervised learning:

- **Data reduction and/or visualization**: Principal component analysis (PCA)
- **Clustering** (finding groups in the data): k-means, spectral clustering

Principal component analysis (PCA)

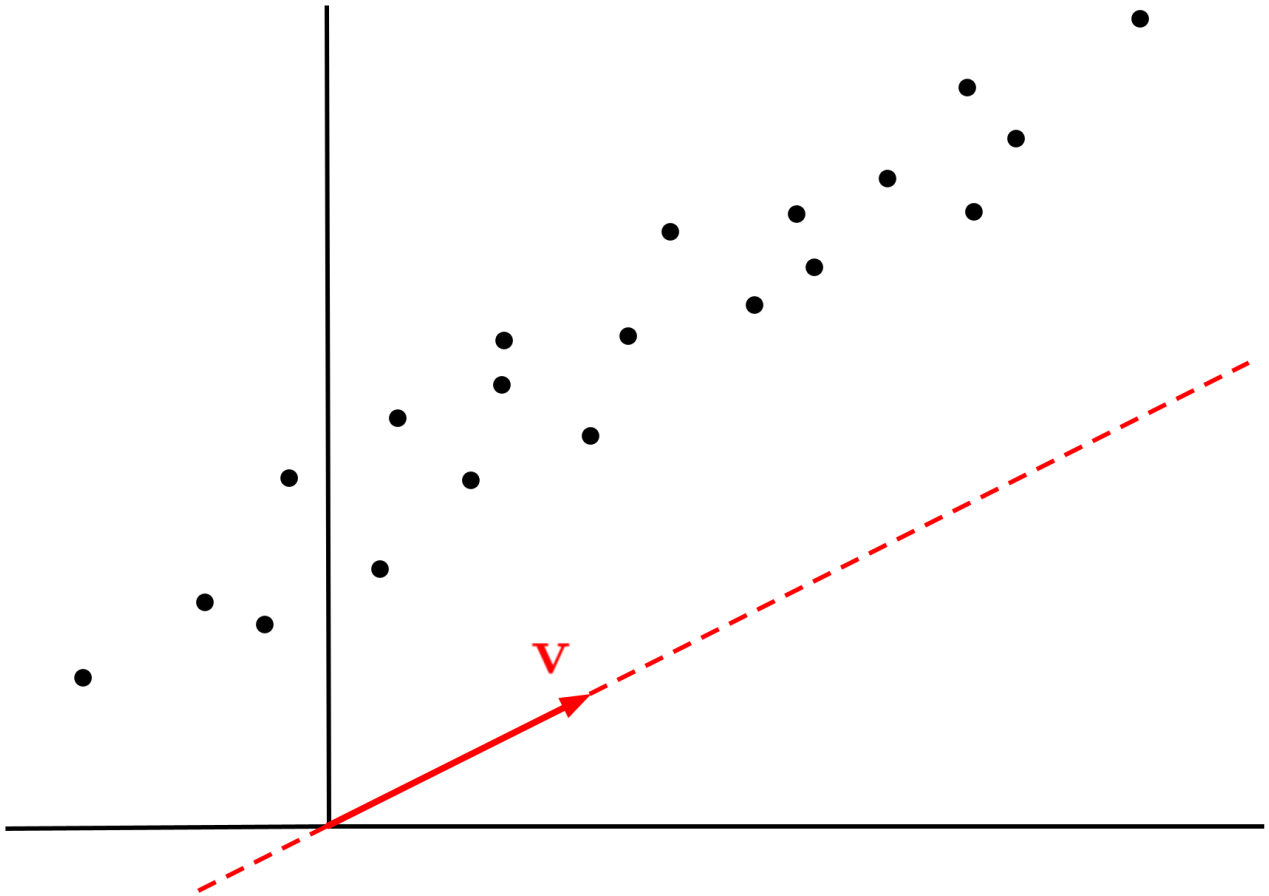
Given a set of features $\vec{X} = (X_1, X_2, \ldots, X_p)$ measured on n observations, i.e.,

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1p} \\ x_{21} & x_{22} & \ldots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{np} \end{bmatrix} \in \mathbb{R}^{n \times p}$$

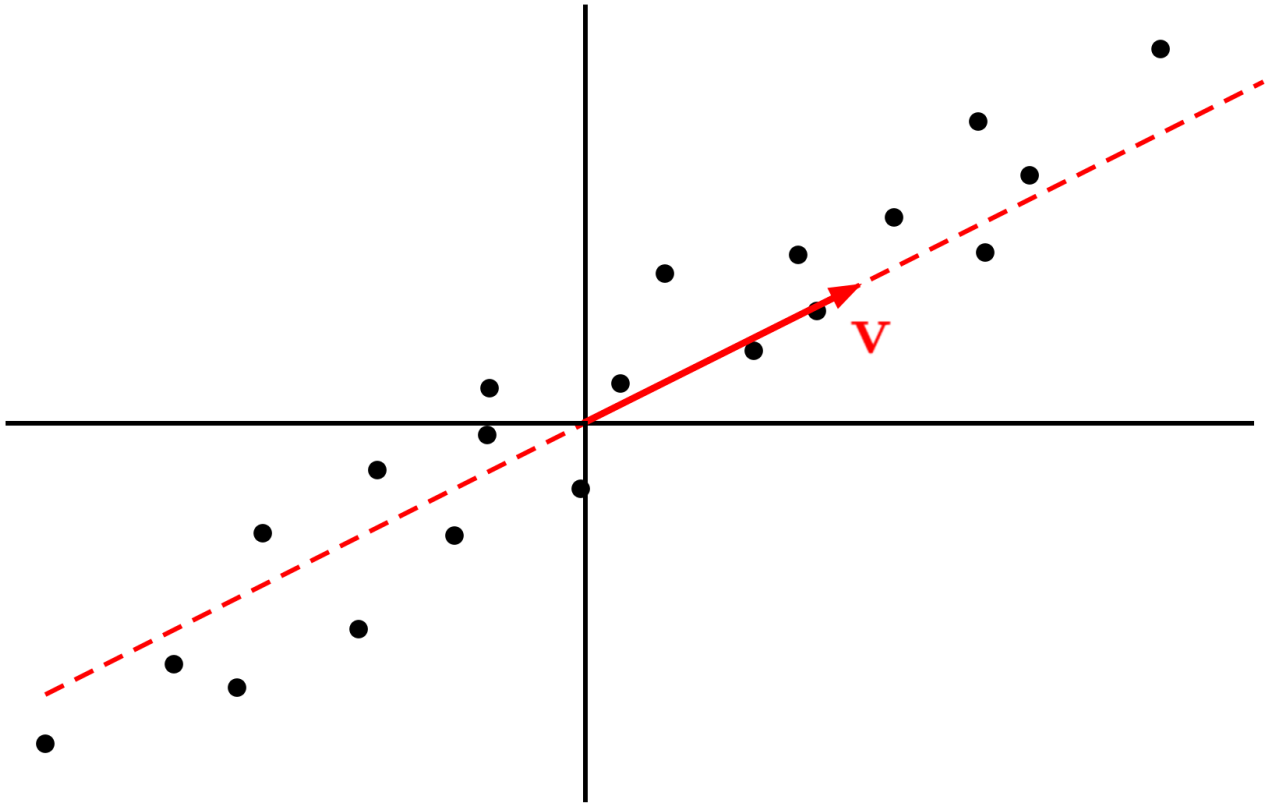
PCA tries to find several orthogonal directions for projecting the data onto them such that the variance of the projected data is as large as possible.

The 1-dimensional PCA problem

Given a data set $\mathbf{X} \in \mathbb{R}^{n \times p}$, we would like to find a unit vector $\mathbf{v} = (v_1, \ldots, v_p)^T \in \mathbb{R}^p$ such that when projecting the data onto the direction, the variance of the projections is the largest possible.



Since parallel lines work the same in terms of projection, we move the coordinate axes to the centroid of the data set, i.e., $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} \vec{x}_i$.



Mathematically, this corresponds to centering the data:

$$\widetilde{x}_i = \vec{x}_i - \bar{x}, \quad 1 \leq i \leq n$$

Note that

- The centroid of the data set, $\bar{x}$, is the row mean of the data matrix

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vdots \\ \vec{x}_n \end{bmatrix}$$

- When subtracting the centroid from each row, we ensure that each column (feature) sums to zero:

$$\widetilde{\mathbf{X}} = \begin{bmatrix} \vec{x}_1 - \bar{x} \\ \vec{x}_2 - \bar{x} \\ \vdots \\ \vec{x}_n - \bar{x} \end{bmatrix} = \begin{bmatrix} \widetilde{x}_1 \\ \widetilde{x}_2 \\ \vdots \\ \widetilde{x}_n \end{bmatrix} = \begin{bmatrix} \widetilde{\mathbf{x}}_1 & \widetilde{\mathbf{x}}_2 & \cdots & \widetilde{\mathbf{x}}_p \end{bmatrix}$$

For any candidate direction $\mathbf{v}$, how do we project the (centered) data set onto it?

- For a single point $\widetilde{x}_i$, we use the dot product:

$$z_i = \widetilde{x}_i \cdot \mathbf{v}.$$

- Collectively, for all points, we need the matrix-vector multiplication

$$\mathbf{z} = \widetilde{\mathbf{X}}\mathbf{v}.$$

That is,

$$\underbrace{\begin{bmatrix} \widetilde{x}_{11} & \widetilde{x}_{12} & \dots & \widetilde{x}_{1p} \\ \widetilde{x}_{21} & \widetilde{x}_{22} & \dots & \widetilde{x}_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \widetilde{x}_{n1} & \widetilde{x}_{n2} & \dots & \widetilde{x}_{np} \end{bmatrix}}_{\widetilde{\mathbf{X}}} \underbrace{\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_p \end{bmatrix}}_{\mathbf{v}} = \underbrace{\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}}_{\mathbf{z}}$$

So now the p-dimensional data set $\mathbf{X}$ has been reduced to a one-dimensional data set $\mathbf{z}$. How do we quantify its variance?

Since the data set has been centered, the mean of the projections $\mathbf{z} = (z_1, z_2, \ldots, z_n)^T \in \mathbb{R}^p$ is also zero.

As a result, the variance of the 1D projections are simply

$$\frac{1}{n-1}\sum_{i=1}^{n} z_i^2$$

The term $\sum_{i=1}^{n} z_i^2$ is often called the scatter of the 1D projections (it differs from variance only by the coefficient $\frac{1}{n-1}$).

Think about this process:

- We start by picking a unit vector **v** arbitrarily
- We then center the data and project them onto **v** in order to obtain a set of 1D coordinates in **z**, with amount of scatter $\sum_{i=1}^{n} z_i^2$.
- As **v** changes, **z** changes and the amount of scatter changes.

What is the optimal direction **v** that leads to the largest amount of scatter in the projection space?

To answer this question, we set up the following optimization problem:

$$\max_{\mathbf{v}:\|\mathbf{v}\|=1} \sum_{i=1}^{n} z_i^2 \qquad \text{where} \quad z_i = \widetilde{x}_i \cdot \mathbf{v}$$

It turns out that the problem can be solved by performing a *singular value decomposition (SVD)* of the centered data matrix $\widetilde{\mathbf{X}}$, with the optimal $\mathbf{v}$ given by the first right singular vector of $\widetilde{\mathbf{X}}$, denoted by $\mathbf{v}_1$:

$$\widetilde{\mathbf{X}} \xrightarrow{\text{svd}} \mathbf{v}_1$$

We won't go into detail on matrix SVD but simply use Python to do the job for us.

In the book,

- the centered data matrix is still denoted $\mathbf{X}$ (pay attention to it);
- the notation for the optimal direction is ϕ_1 instead of $\mathbf{v}_1$;
- the resulting projections $\mathbf{z}_1 = \widetilde{\mathbf{X}}\mathbf{v}_1$ are called the **first principal component** of the data;
- the components of $\mathbf{v}_1$ are called the **loadings** of the first principal component $\mathbf{z}_1$, because they are used to linearly combine the components of each $\widetilde{x}_i$;
- the vector $\mathbf{v}_1$ is called the first **principal component loadings vector**, but other people may call it the first **principal axis** or **direction** of the data.

An example

Example

Consider the following toy data set ($n = 4, p = 2$):

$$\mathbf{X} = \begin{bmatrix} 0 & 2 \\ 2 & 0 \\ 1 & 3 \\ 3 & 1 \end{bmatrix} \quad \overset{\text{centroid}}{\Longrightarrow} \quad \bar{x} = (1.5, 1.5)^T.$$

The centered data set is

$$\widetilde{\mathbf{X}} = \begin{bmatrix} 0 & 2 \\ 2 & 0 \\ 1 & 3 \\ 3 & 1 \end{bmatrix} - \begin{bmatrix} 1.5 & 1.5 \\ 1.5 & 1.5 \\ 1.5 & 1.5 \\ 1.5 & 1.5 \end{bmatrix} = \begin{bmatrix} -1.5 & 0.5 \\ 0.5 & -1.5 \\ -0.5 & 1.5 \\ 1.5 & -0.5 \end{bmatrix}$$

Using Python, we obtain the first principal axis of the data as follows:

$$\mathbf{v}_1 = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} \end{bmatrix}$$

The first principal component of the data is thus

$$\mathbf{z} = \widetilde{\mathbf{X}}\mathbf{v}_1 = \begin{bmatrix} -1.5 & 0.5 \\ 0.5 & -1.5 \\ -0.5 & 1.5 \\ 1.5 & -0.5 \end{bmatrix} \cdot \frac{\sqrt{2}}{2} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \begin{bmatrix} -\sqrt{2} \\ \sqrt{2} \\ -\sqrt{2} \\ \sqrt{2} \end{bmatrix}$$

In [ ]:
```
import numpy as np
X = np.array([[0,2],[2,0],[1,3],[3,1]])
print(X)
```

In [ ]:
```
from matplotlib.pyplot import subplots
fig, ax = subplots(figsize=(6, 6));
ax.plot(X[:,0], X[:,1], 'o');
```

In [ ]:
```
centroid = X.mean(axis=0)
print(centroid)
```

In [ ]:
```
X_centered = X - centroid
print(X_centered)
```

In [ ]:
```
u,s,vt = np.linalg.svd(X_centered, full_matrices=False)
```

In [ ]:
```
vt # only need vt
```

In [ ]:
```
v1 = vt[0,:]
print(v1)
```

In [ ]:
```
pc1 = np.dot(X_centered, v1)
print(pc1)
```

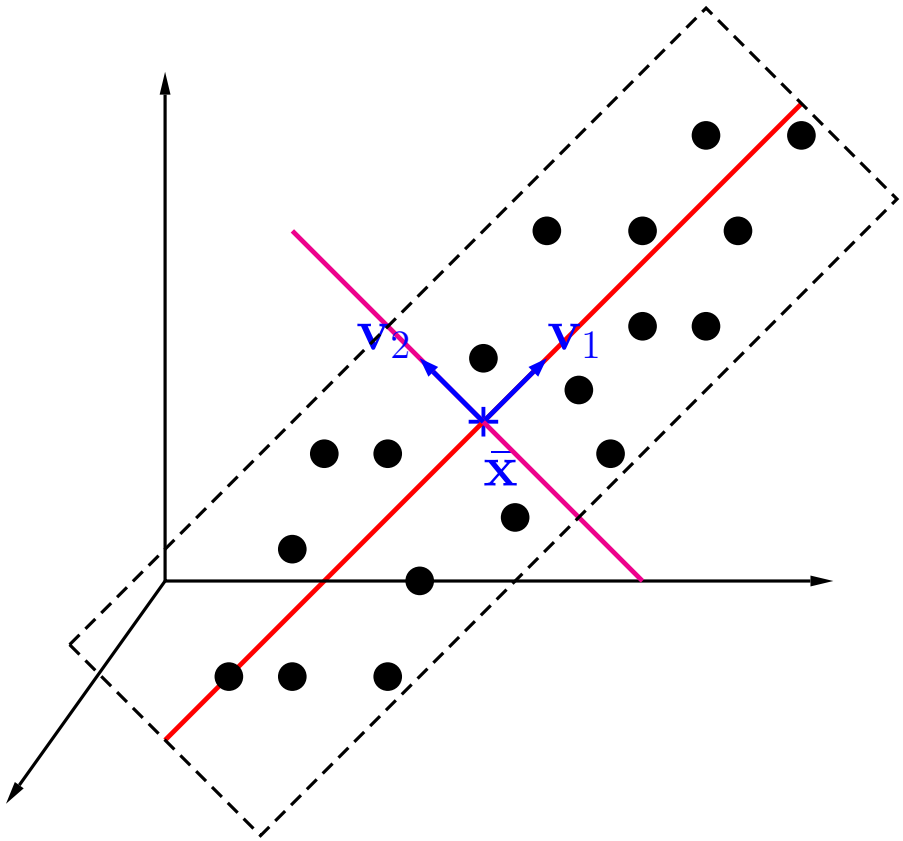
Alternatively, you can use the PCA function directly

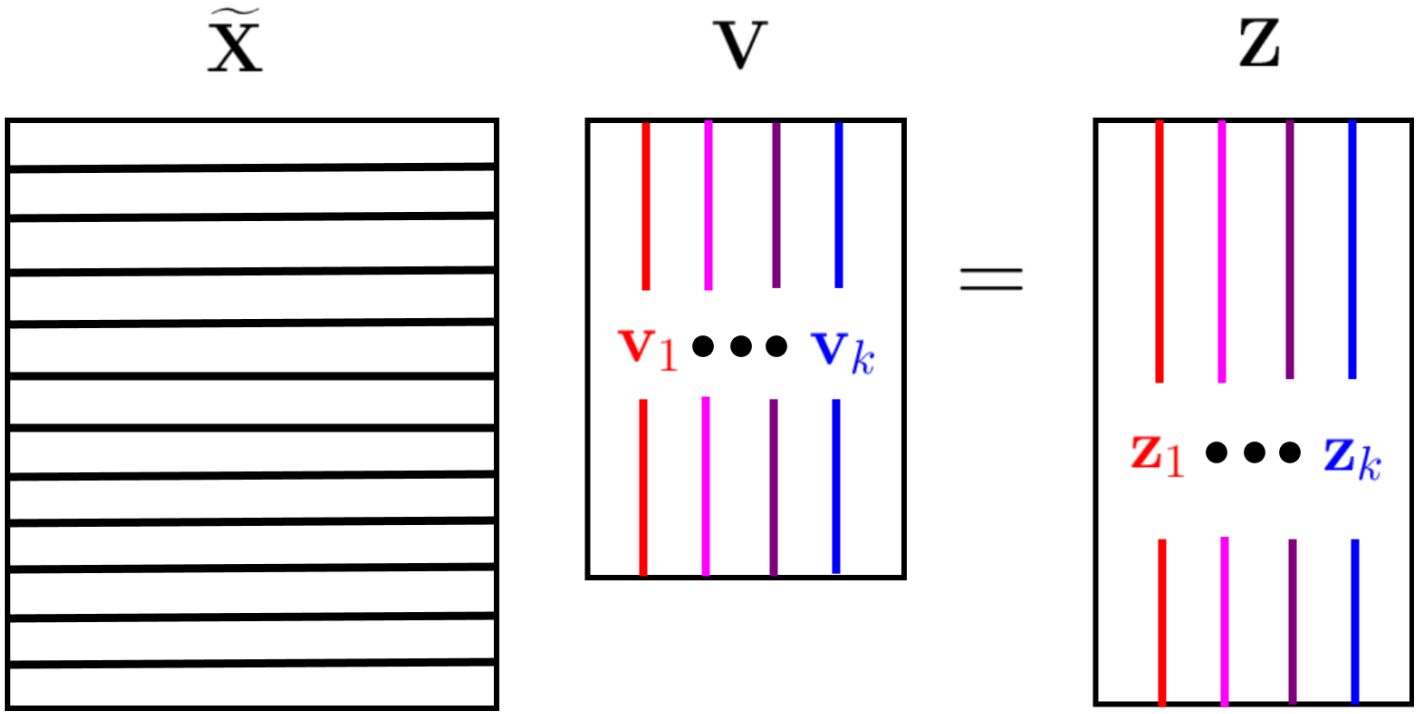
```
In [1]: import numpy as np
        X = np.array([[0,2],[2,0],[1,3],[3,1]])

In [2]: from sklearn.decomposition import PCA
        pca = PCA(n_components=1)

In [3]: pca.fit_transform(X)   # PCA with principal component output
Out[3]: array([[ 1.41421356],
               [-1.41421356],
               [ 1.41421356],
               [-1.41421356]])

In [4]: pca.mean_              # centroid
Out[4]: array([1.5, 1.5])

In [5]: pca.components_        # loadings (maximum-variance direction)
Out[5]: array([[-0.70710678,  0.70710678]])
```

k-dimensional PCA

It turns out that we can use more right singular vectors $\mathbf{v}_2, \ldots, \mathbf{v}_k$ of the centered data matrix (along with $\mathbf{v}_1$) to form a k-dimensional maximum-variance projection plane:

- The singular vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ are all unit vectors and mutually orthogonal to each other;
- Individually, $\mathbf{v}_1$ is better than $\mathbf{v}_2$, which is further better than $\mathbf{v}_3$, so on and so forth;
- The different principal components are uncorrelated with each other.



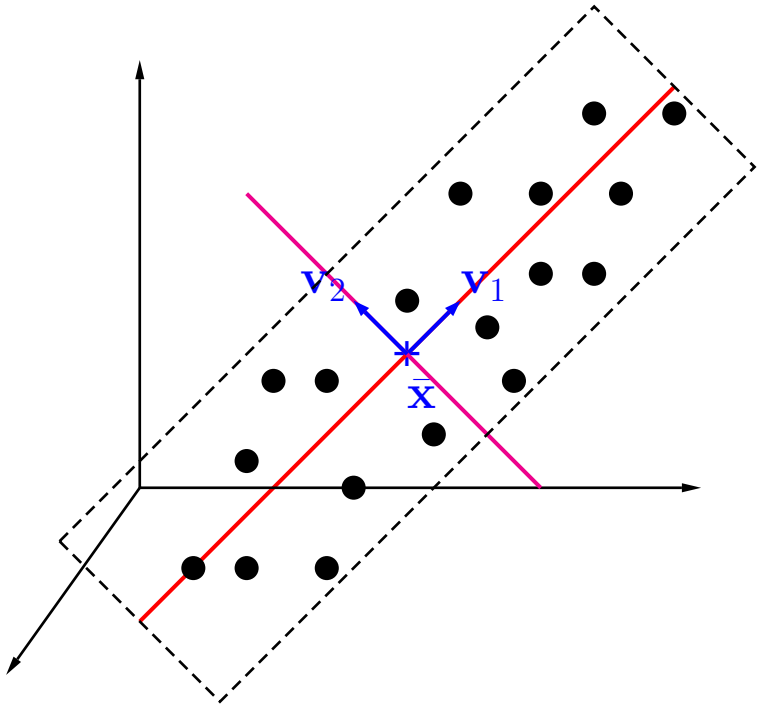


Note that each row of $\mathbf{Z}$ corresponds to a row of $\widetilde{\mathbf{X}}$ (data point); it provides the low dimensional representation of the point.

Other interpretations of PCA

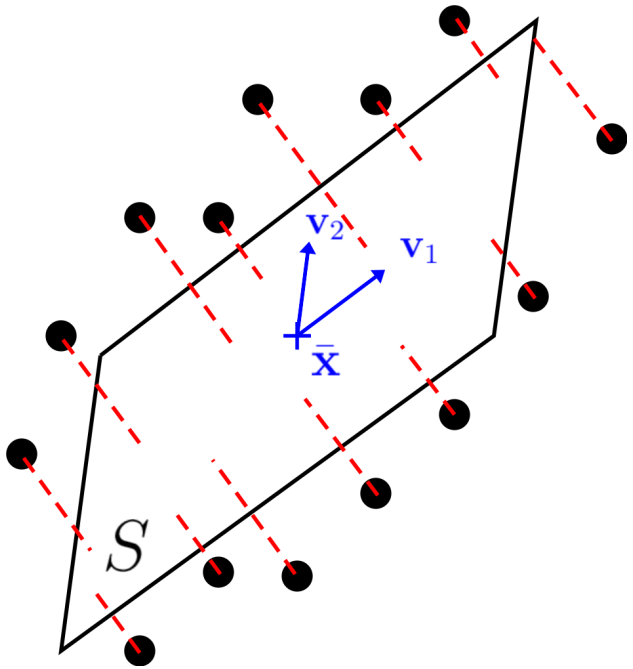
1. PCA is a change of coordinate system by using the maximum-variance directions of the data!

- The new origin is set at the centroid of the data set, $\bar{\mathbf{x}}$;
- The new coordinate axes are set along the principal axes of the data, i.e., $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$;
- The new coordinates of the data are the principal components.



This allows us to visualize the data by plotting the top few principal components.

2. The PCA plane minimizes the orthogonal fitting errors among all planes of the same dimension.

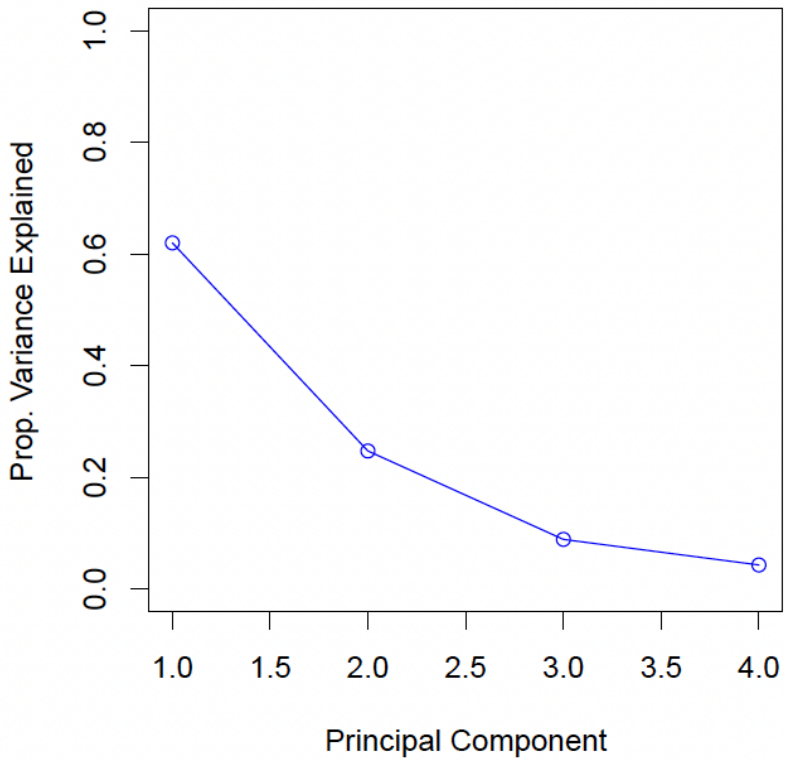


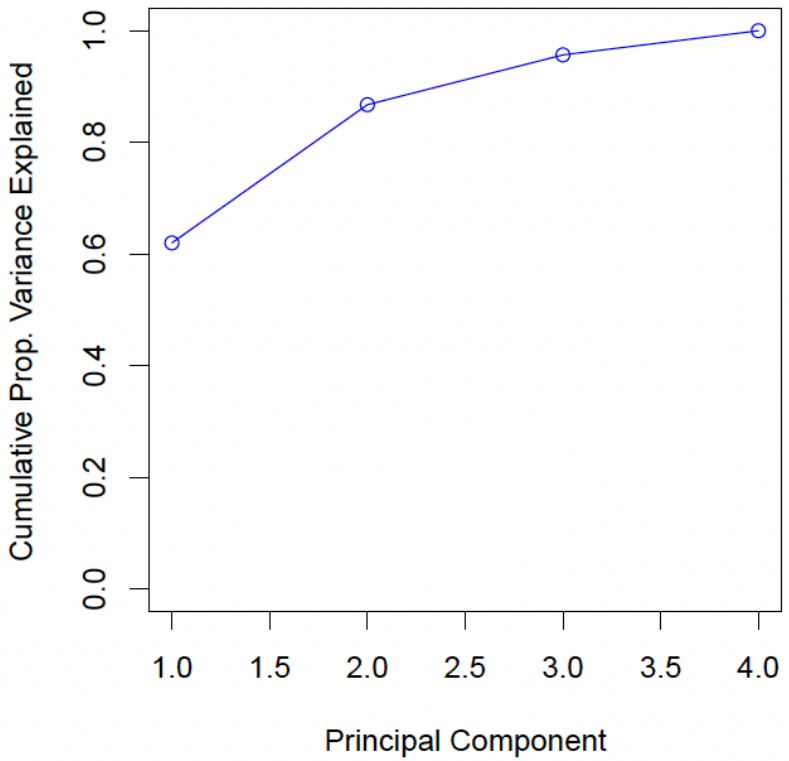
3. PCA provides an approximate decomposition of the centered data matrix.

$$\widetilde{\mathbf{X}} \approx \underbrace{\mathbf{Z}}_{\text{coeffs}} \cdot \underbrace{\mathbf{V}^T}_{\text{basis}}$$

Numerical issues

- Remember to standardize the variables in practical settings where features tend to have very different magnitudes.
- Choose the number of principal components k by **proportion of variance explained**: Compute the amount of scatter each principal component has and examine the cumulative amount of scatter relative to the overall amount of scatter in the data.





Lab - part 1

Section 12.5.1 Principal Components Analysis

(Remember to include the scripts before the section as well)

Applications of PCA

- **Data visualization**: project the data using top few principal axes and plot the data for examining
- **Dimensionality reduction** (and noise suppression) to serve other machine learning tasks such as regression, classification, and clustering
- **Imputing missing values** in a data set through matrix completion (next)

Missing values

In some contexts, the data set given to us is missing values in some locations.

This can happen due to various reasons, e.g.,

- data loss/corruption
- intentional masking of information
- inability to collect all the information

An example of the last scenario is the Netfix movie ratings data[1], where a very high percentage of the ratings are missing.

[1] https://en.wikipedia.org/wiki/Netflix_Prize

	Jerry Maguire	Oceans	Road to Perdition	A Fortunate Man	Catch Me If You Can	Driving Miss Daisy	The Two Popes	The Laundromat	Code 8	The Social Network	...
Customer 1	•	•	•	•	4	•	•	•	•	•	...
Customer 2	•	•	3	•	•	•	3	•	•	3	...
Customer 3	•	2	•	4	•	•	•	•	2	•	...
Customer 4	3	•	•	•	•	•	•	•	•	•	...
Customer 5	5	1	•	•	4	•	•	•	•	•	...
Customer 6	•	•	•	•	•	2	4	•	•	•	...
Customer 7	•	•	5	•	•	•	•	3	•	•	...
Customer 8	•	•	•	•	•	•	•	•	•	•	...
Customer 9	3	•	•	•	5	•	•	1	•	•	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

TABLE 12.2. *Excerpt of the Netflix movie rating data. The movies are rated from 1 (worst) to 5 (best). The symbol • represents a missing value: a movie that was not rated by the corresponding customer.*

Imputing missing values

Warning: Missing values are not always recoverable unless missed randomly.

Simple techniques to handle missing values:

- **Remove** the instances that contain missing observations $\longrightarrow$ can be a significant waste of information
- **Replace** the missing values of each feature by the mean/median (continuous) or most frequent level (categorical) of the column
- **Estimate** the missing values based on the most similar rows through averaging (continuous) / majority vote (categorical)

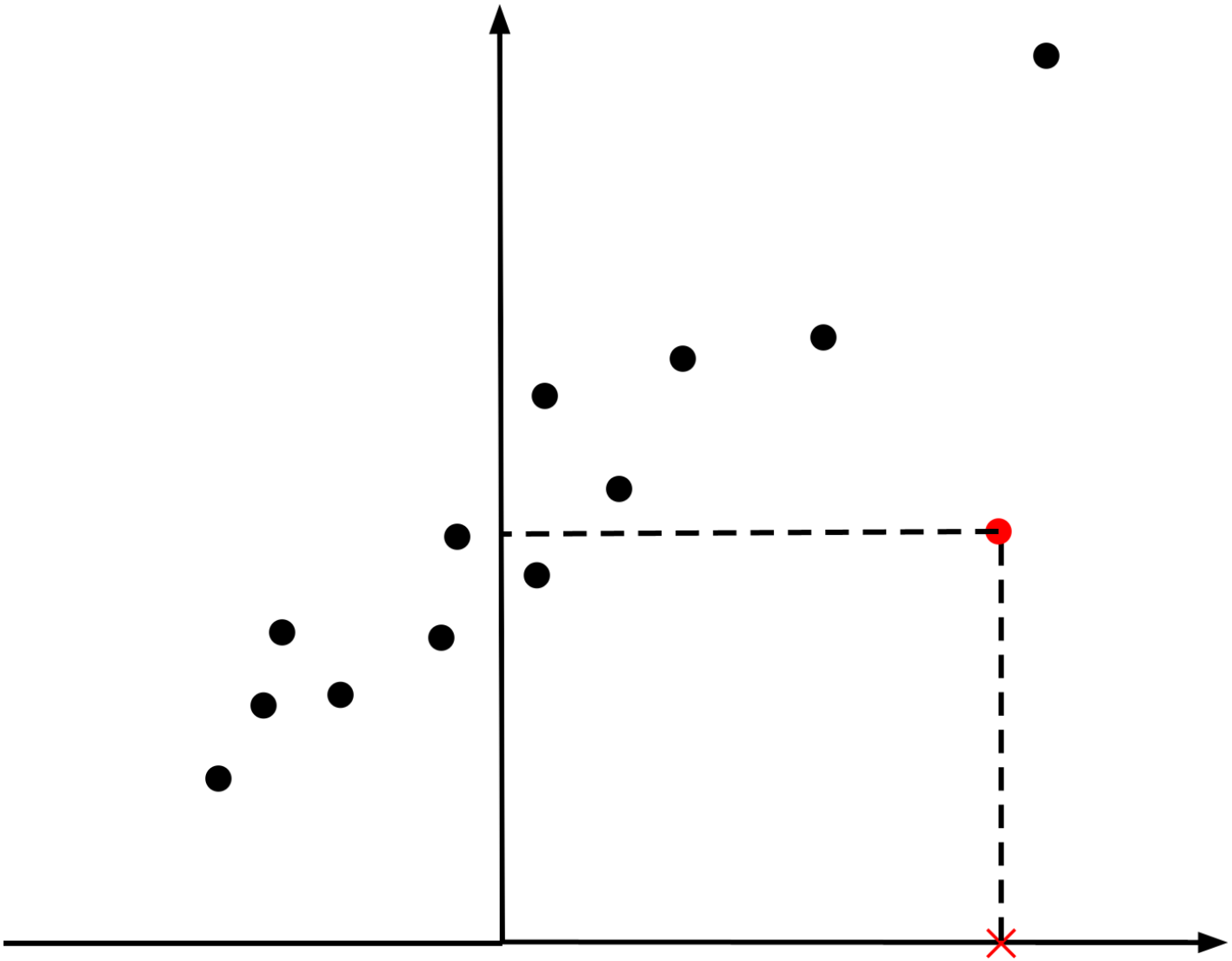
Here, we show how PCA can be used to impute the missing values, through a process known as **matrix completion**.

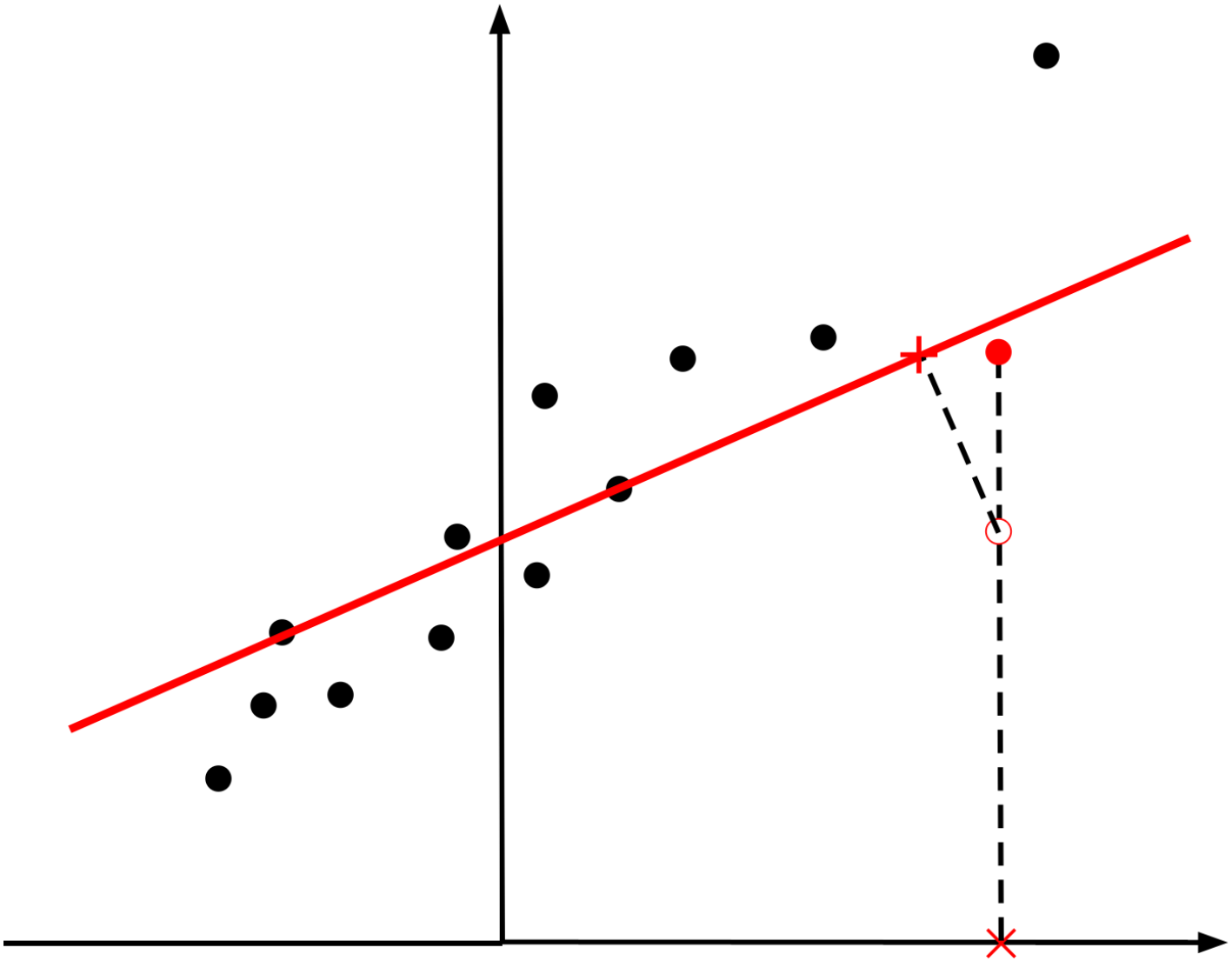
Matrix completion through PCA

For this idea to work, besides the random missingness assumption, we suppose that the full data set lies around a k-dimensional plane, where k is known.

Accordingly, we will tune the missing values (while keeping the observed values fixed) toward a k-dimensional plane:

1. We will start by filling the missing values by very simple methods such as column mean.
2. We will then perform k-dimensional PCA on the complete data and use the PCA plane to fill in the missing values (the observed values are not changed).
3. We iterate the above process until convergence.





An algorithm

Input: $\mathbf{X} \in \mathbb{R}^{n \times p}$ (with missing entries), PCA dimension k
Output: Imputed data matrix $\mathbf{Y} \in \mathbb{R}^{n \times p}$

1: Initialization:

- Let $\mathcal{M}$ the index set of missing entries in $\mathbf{X}$
- Set counter $j = 0$
- Fill in the missing values of each column in $\mathbf{X}$ by column mean; let the correspondingly completed matrix be $\mathbf{X}_0$ (initial attempt).

2: Repeat

- Perform k-dimensional PCA with $\mathbf{X}_j$ to obtain $\widetilde{\mathbf{X}}_j \approx \mathbf{Z}\mathbf{V}^T$
- Let $\mathbf{X}(\mathcal{M}) = \mathbf{Z}\mathbf{V}^T(\mathcal{M}) + \bar{\mathbf{x}}_j$ (and keep the entries of $\mathbf{X}$ elsewhere); denote the new matrix by $\mathbf{X}_{j+1}$
- Update counter by $j \leftarrow j + 1$

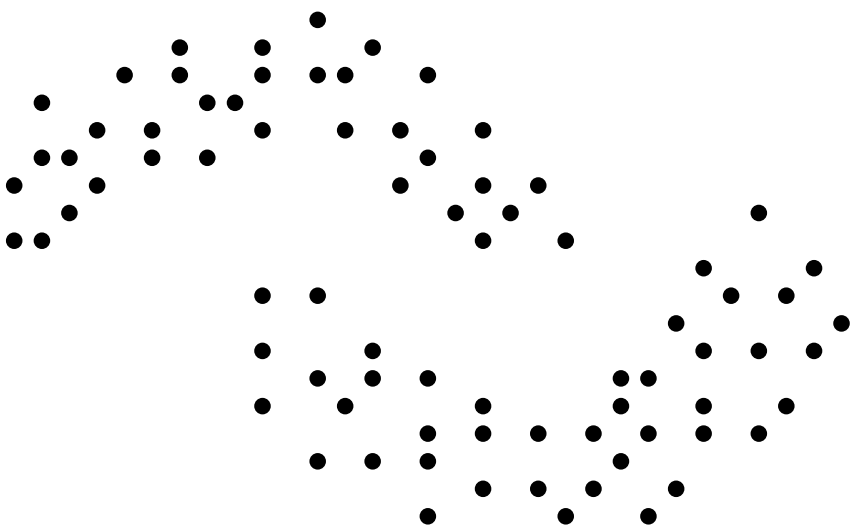
Until the process has converged.

3: Return $\mathbf{Y} = \mathbf{X}_j$

Section 12.5.2 Matrix Completion

Introduction to Clustering

Assume a data set $\mathbf{X} \in \mathbb{R}^{n \times d}$. We would like to divide the data set into k disjoint subsets (called clusters) such that within every cluster points are "similar" (at least to their near neighbors), and between clusters points are "dissimilar".



Remark. Clustering is an unsupervised machine learning task, often called learning without a teacher.

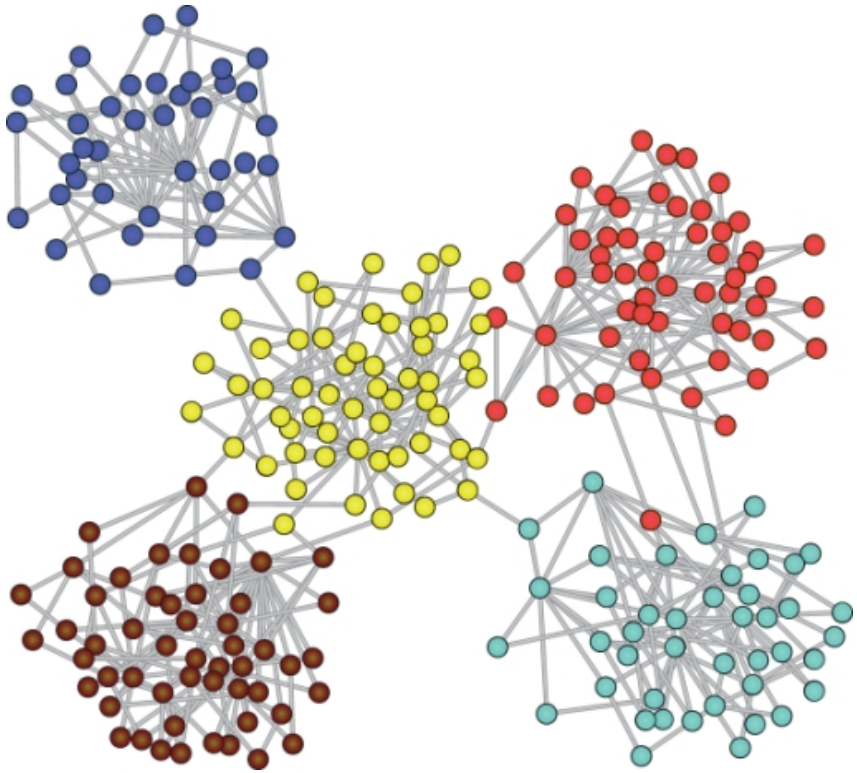
In contrast, classification is supervised (learning with a teacher).

Applications of clustering

Clustering is the process of discovering groups in the data, a very common data mining task.

Examples of its application include

- Document grouping
- Customer segmentation
- Social network partitioning
- Image segmentation



Some necessary components of clustering

- Objects and their attributes (i.e., data matrix)
- Number of clusters
- Similarity or dissimilarity measure
- Algorithmic implementation
- Evaluation criterion (e.g., objective function)
- Interpretation of results

Clustering (by machines) is hard!

Questions to be answered *before* clustering:

- Which similarity measure is proper?
- How many clusters should we find?
- What objective function / model to use?

Afterwards:

- What kind of pattern of the data set do the clusters reveal?

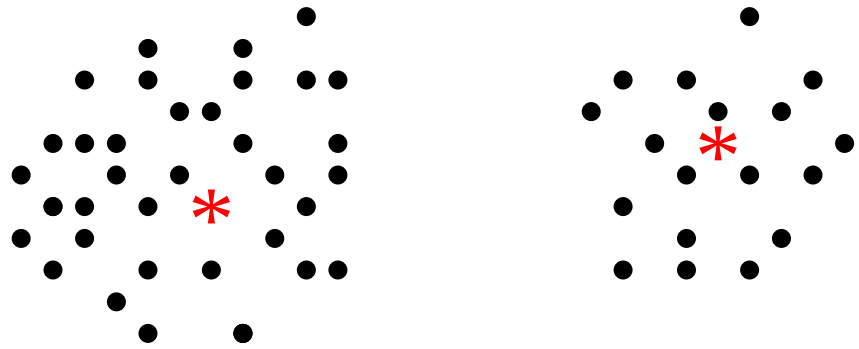
Clustering methods

- Hierarchical clustering
- Centroid-based clustering (e.g., **k-means**)
- Distribution-based clustering (e.g., mixture of Gaussians)
- Density-based clustering
- **Spectral clustering**

k-means clustering

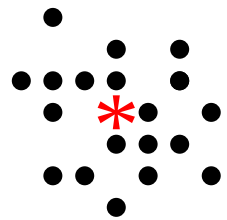
Problem. Given n data points in $\mathbb{R}^d$, $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, and a positive integer k, find a partition of X into k disjoint clusters $C_1, \ldots, C_k$ such that the total within-class scatter is the smallest possible:

$$\min_{C_1,\ldots,C_k} \sum_{j=1}^{k} \underbrace{\sum_{\mathbf{x}\in C_j} \|\mathbf{x} - \boldsymbol{\mu}_j\|^2}_{\text{scatter of } C_j}$$



where $\boldsymbol{\mu}_j$ is the centroid of C_j:

$$\boldsymbol{\mu}_j = \frac{1}{n_j} \sum_{\mathbf{x}\in C_j} \mathbf{x}$$



Observation. The original problem is combinatorial in nature (the naive approach of checking every possible partition is too costly).

Technique. We rewrite the above problem as follows:

$$\min_{\{C_j\},\{\mathbf{a}_j\}} \sum_{j=1}^{k} \sum_{\mathbf{x}\in C_j} \|\mathbf{x} - \mathbf{a}_j\|^2$$

where the $\mathbf{a}_j$'s have been added as new variables (relaxed reference points).

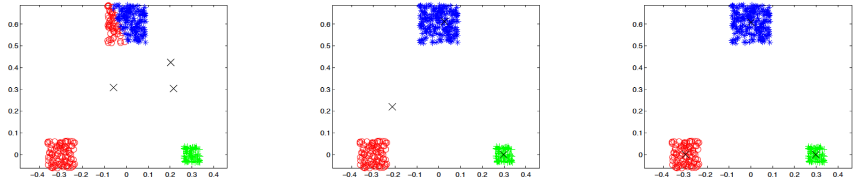
It is still the same problem:

- Given the clusters $C_1, \ldots, C_k$, the optimal choices of $\mathbf{a}_j$ are their centroids.
- Given the reference points $\mathbf{a}_1, \ldots, \mathbf{a}_k$, the optimal clusters $\{C_j\}$ are formed by assigning points to the nearest $\mathbf{a}_j$.

Note that choosing $\{C_j\}, \{\mathbf{a}_j\}$ simultaneously to minimize the total scatter is difficult.

We can adopt an alternating procedure to solve the above problem (approximately):

$$\{\mathbf{a}_j^{(0)}\} \longrightarrow \{C_j^{(1)}\} \longrightarrow \{\mathbf{a}_j^{(1)}\} \longrightarrow \{C_j^{(2)}\} \longrightarrow \cdots$$



Pseudocode of k-means clustering

Input: Data $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset \mathbb{R}^d$, number of clusters k

Output: A partition of X into k clusters $C_1, \ldots, C_k$

Steps:

1: **Initialization**: Randomly select k initial points $\mathbf{a}_1^{(0)}, \ldots, \mathbf{a}_k^{(0)}$

2: Let $t \leftarrow 1$ be the iteration index.

Repeat

(1) For each j, assign to $C_j^{(t)}$ the points that are closest to $\mathbf{a}_j^{(t-1)}$

(2) Center update: $\mathbf{a}_j^{(t)} = \frac{1}{|C_j^{(t)}|} \sum_{\mathbf{x} \in C_j^{(t)}} \mathbf{x}$, for $j = 1, \ldots, k$

(3) $t \leftarrow t + 1$

until some *stopping criterion* has been reached (e.g., $t > 100$ or total scatter stops decreasing)

3: **Return** the final clusters $C_j^{(t)}, \ldots, C_j^{(t)}$

Numerical issues

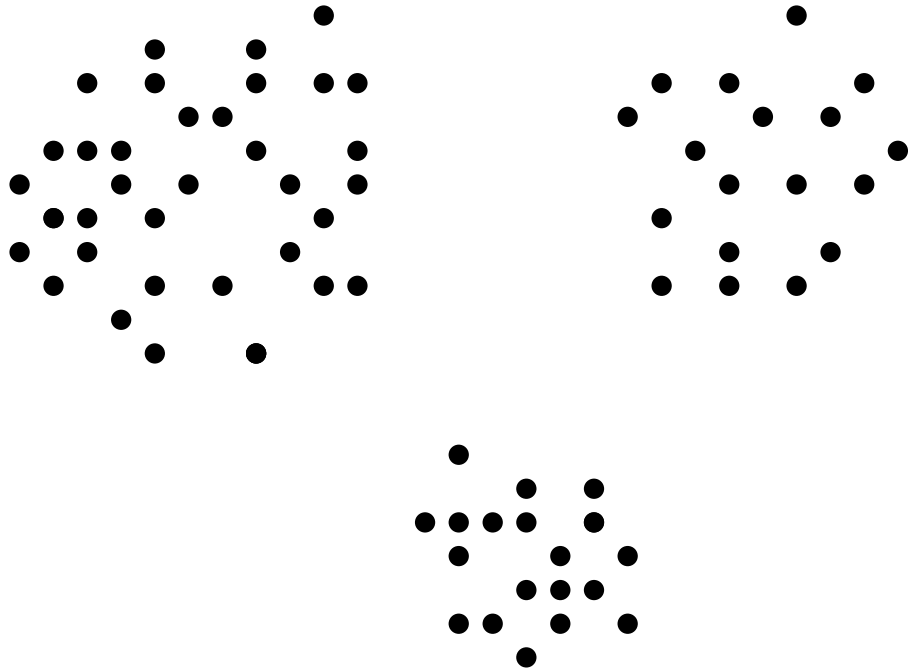
The following issues need to be addressed before perform kmeans clustering:

- **Scaling**: Like PCA, kmeans clustering is also distance based. Therefore, it is important to standardize the variables (when their magnitudes differ considerably) first so that no variable will dominate the rest.
- **High dimensional data**: Data sets that have many features cause challenges in terms of memory, speed, and accuracy. One can apply PCA as a preprocessing step to help kmeans.

How to initialize *k*-means

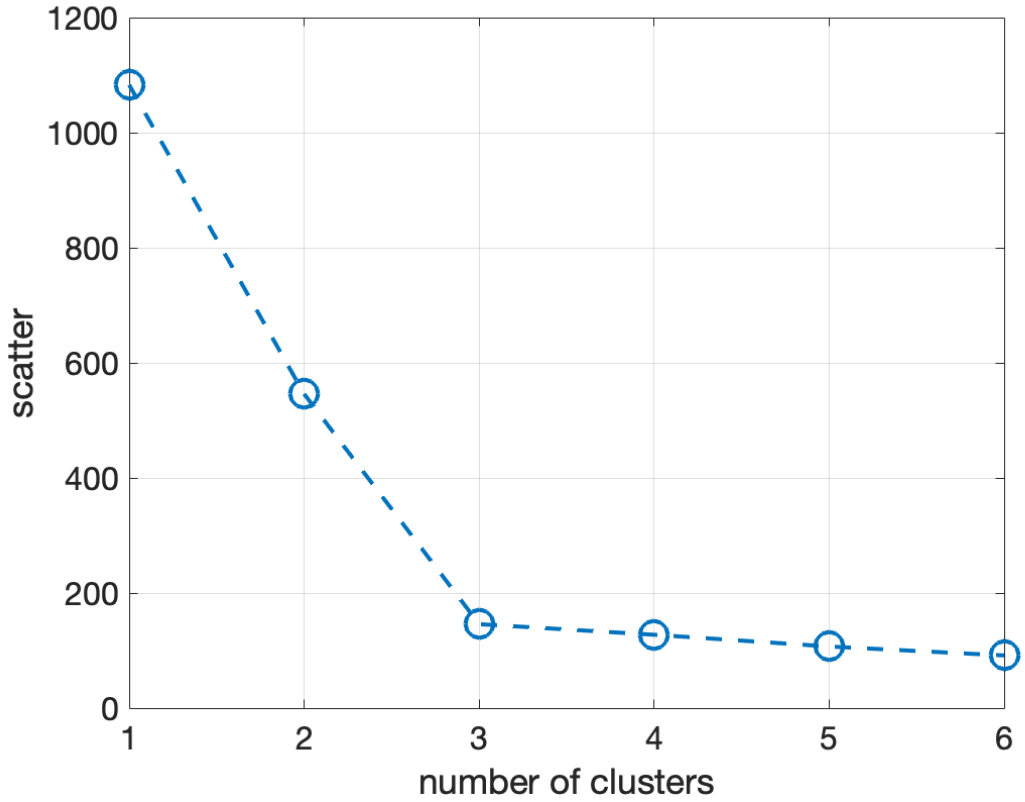
Some common ways of selecting initial seed points:

- Sample k points from the data set uniformly at random
- *k*means++: pick points with probability proportional to distance to closest existing center
- Perform preliminary clustering phase on random 10% of the data



No initialization method is guaranteed to work; use multiple restarts!

Determining k via the elbow method



Technique: Set k based on the "elbow point" of the curve (where the scatter stops decreasing drastically)

Unfortunately, this method only works on easy data sets (for more complex data sets, the curve tends to have a smooth decay).

Demonstrations

See

- Instructor's Python scripts for kmeans clustering
- Python documentation for kmeans clustering[2]

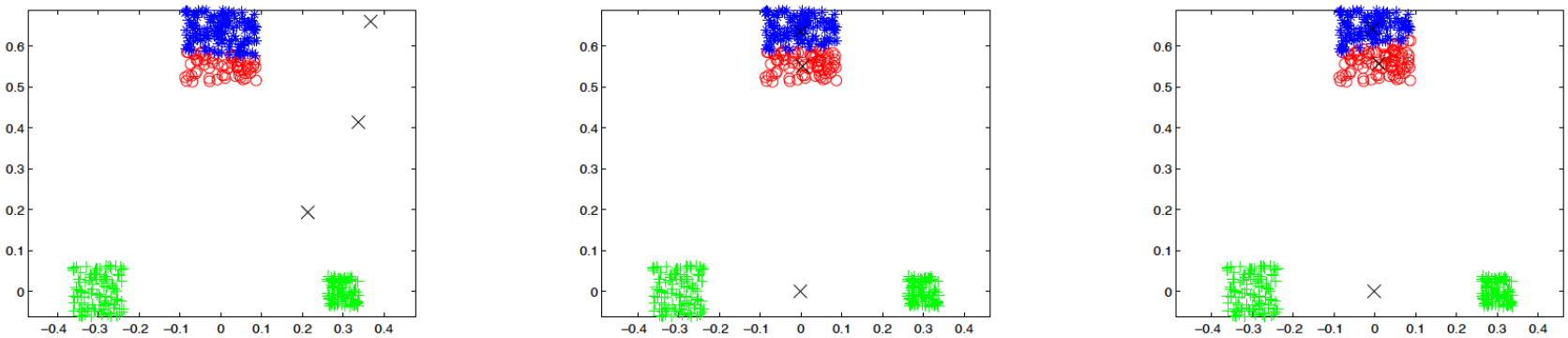
[2] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

Advantages and disadvantages of k-means

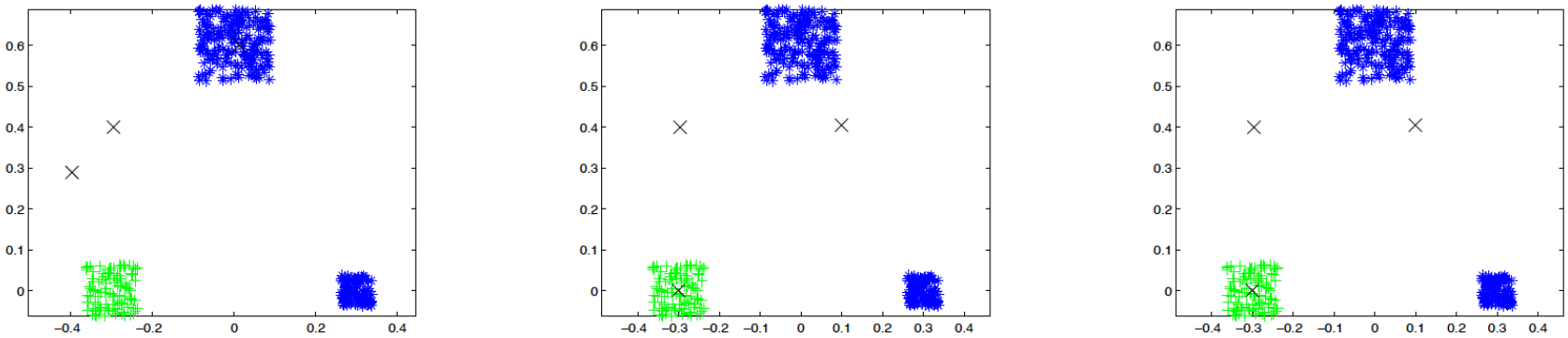
- Advantages
  - simple and fast (often, but not always)
  - always converges
- Disadvantages
  - does not always return the best partition;
  - cannot handle nonconvex data

Local minima

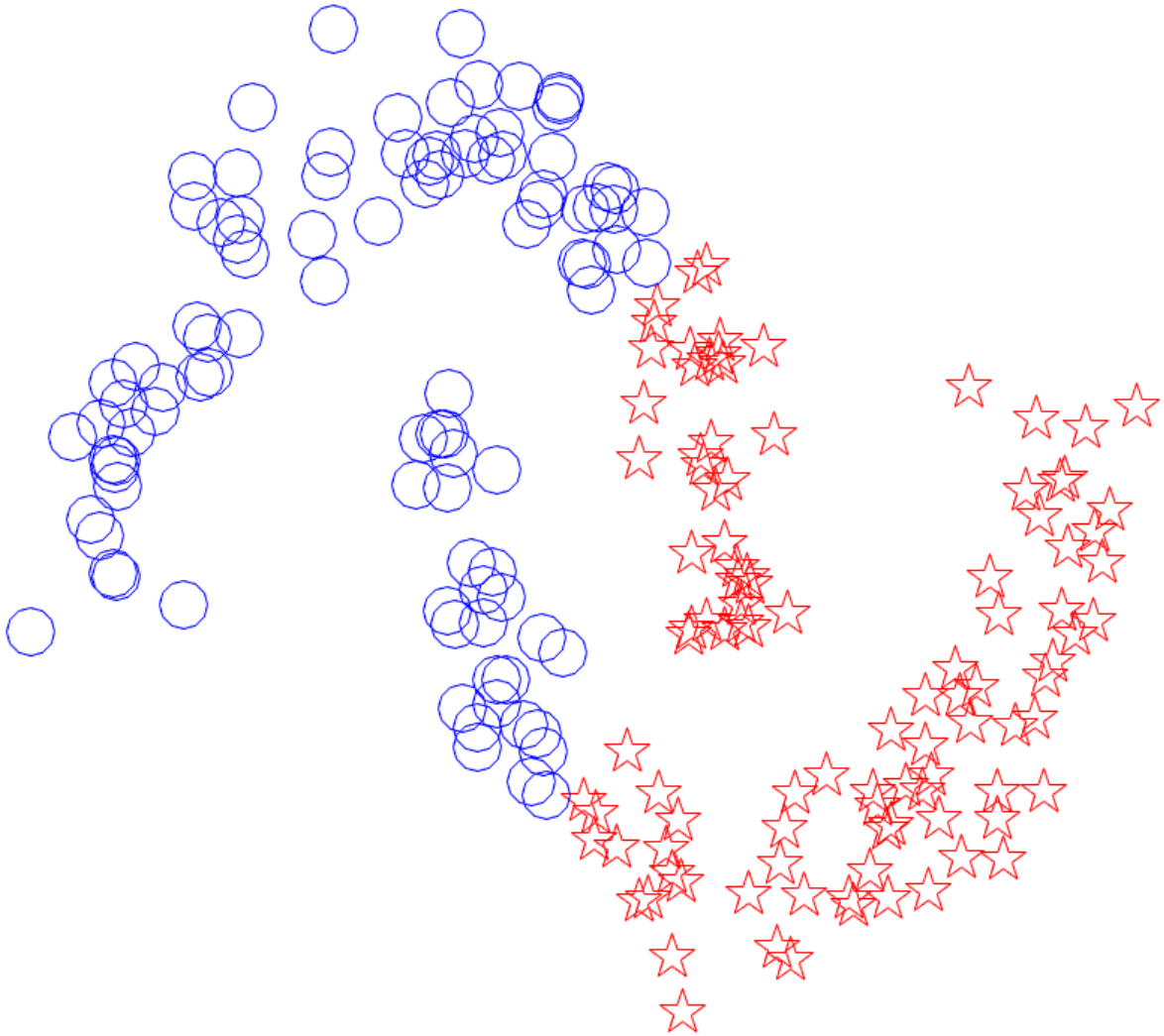
Bad case 1 (suboptimal solution):



Bad case 2 (empty cluster):



Nonconvex clusters



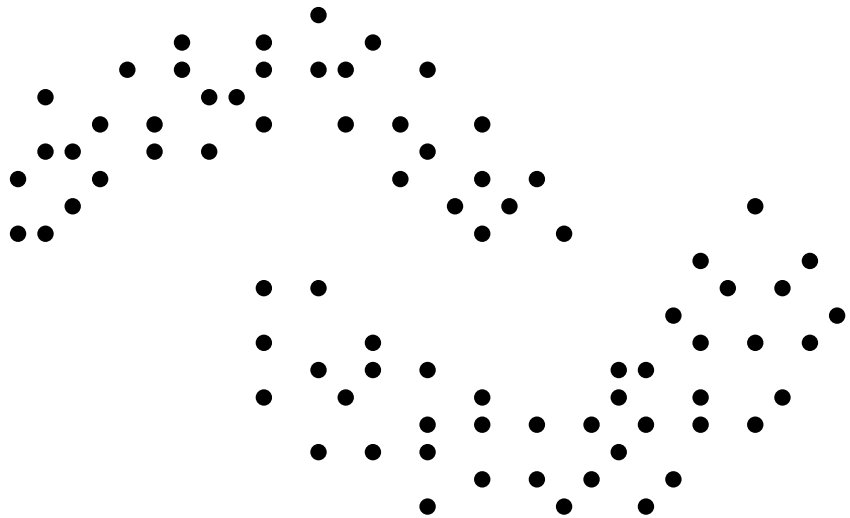
Spectral clustering

Spectral clustering refers to a family of clustering algorithms that utilize the matrix eigendecomposition to embed the data.

Given a set of data points in Euclidean space, $\vec{x}_1, \ldots, \vec{x}_n \in \mathbb{R}^d$, the first step of spectral clustering is to construct a pairwise similarity matrix $\mathbf{W} = (w_{ij}) \in \mathbb{R}^{n \times n}$ with

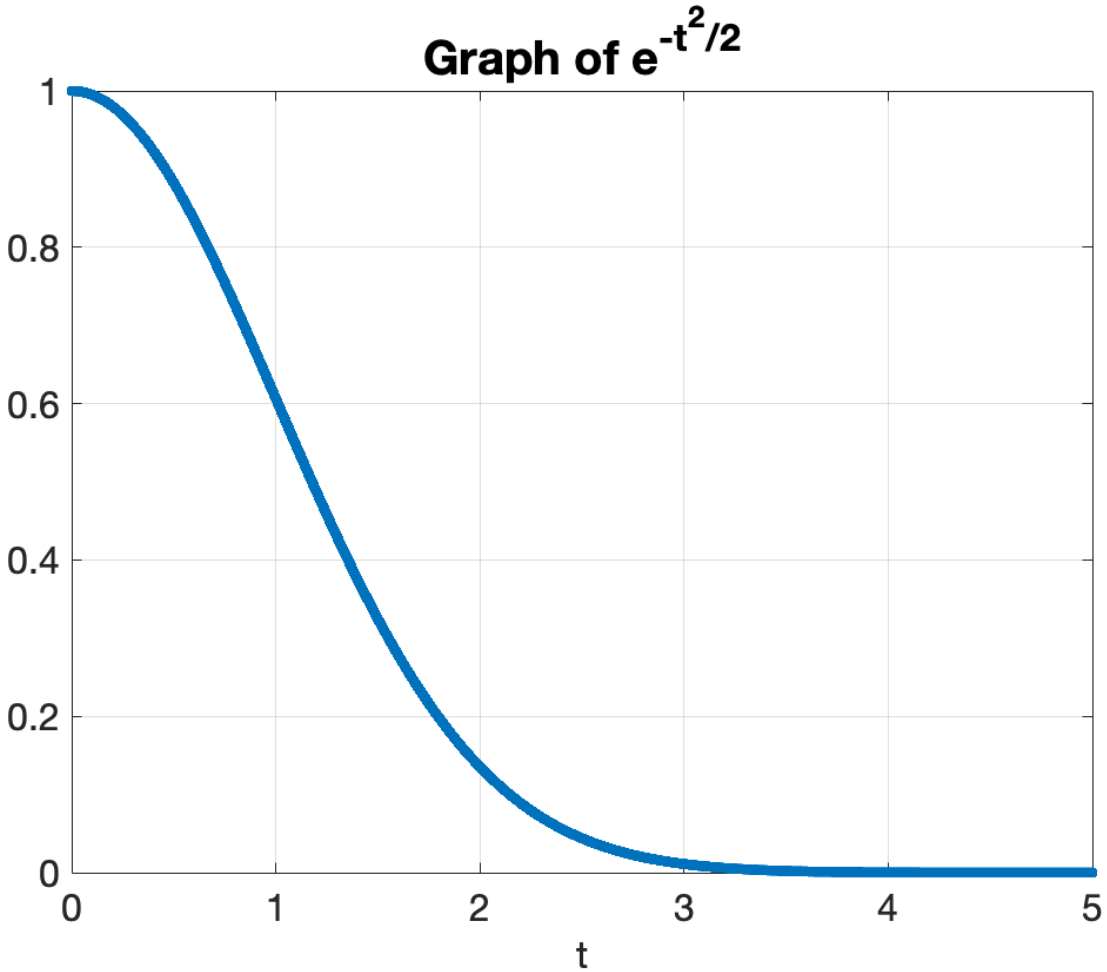
$$w_{ij} = \begin{cases} e^{-\frac{\|\vec{x}_i - \vec{x}_j\|^2}{2\sigma^2}}, & i \neq j \\ 0, & i = j \end{cases}$$

where $\sigma > 0$ is a scale parameter set by the user.



Remark. The entries of $\mathbf{W}$ are all between 0 and 1:

- If $\|\vec{x}_i - \vec{x}_j\| \gg \sigma$ (two faraway points), then $w_{ij} \approx 0$;
- If $\|\vec{x}_i - \vec{x}_j\| \ll \sigma$ (two close points), then $w_{ij} \approx 1$.

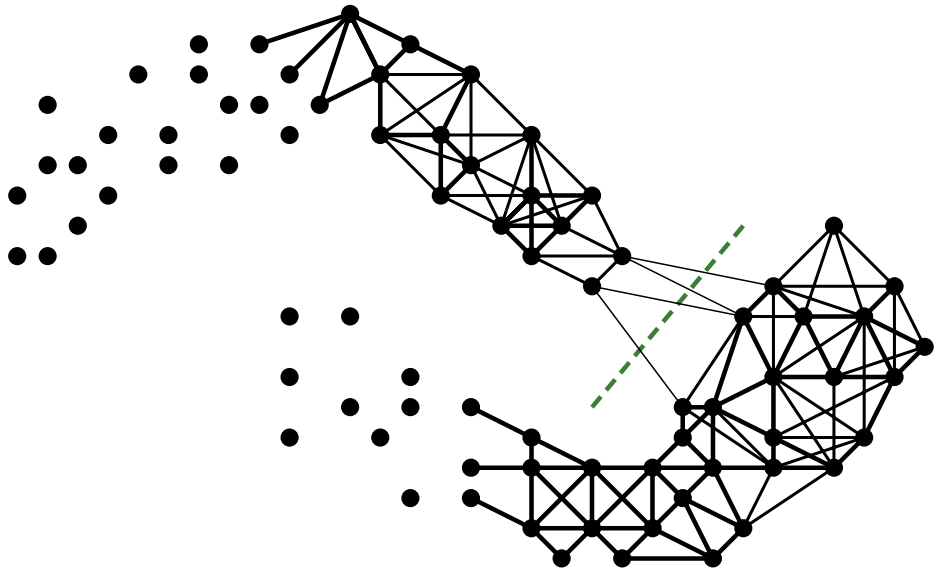


A graph cut point of view for clustering

W (as a weight matrix) defines a weighted graph on the given data:

- Vertices represent the given data points
- Edges are weighted by w_{ij} (there is an edge between $\mathbf{x}_i$ and $\mathbf{x}_j$ if and only if $w_{ij} > 0$)

Accordingly, the problem of clustering is equivalent to finding an optimal cut (under some criterion).



The Normalized Cut (NCut) algorithm

Using the graph cut perspective, Shi and Malik (2000) developed the following NCut algorithm:

Input: Data $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$, #clusters k, scale parameter σ
Output: A partition of the data into clusters $C_1, \ldots, C_k$

1: Construct the matrix of pairwise similarities

$$\mathbf{W} = (w_{ij}), \quad w_{ij} = e^{-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}}, \ i \neq j$$

2: Divide each row of $\mathbf{W}$ by the corresponding row sum (so that the new row sum is 1). Denote the resulting matrix by $\mathbf{P}$.
3: Find the 2nd to kth largest eigenvectors of $\mathbf{P}$ and form a matrix

$$\mathbf{V} = [\mathbf{v}_2 \ldots \mathbf{v}_k] \in \mathbb{R}^{n \times (k-1)}.$$

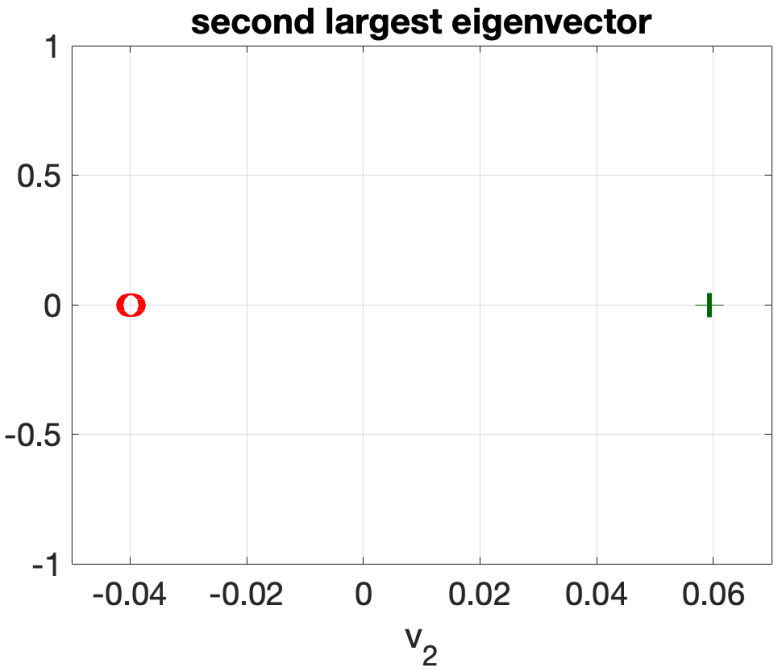
(It can be shown that $\mathbf{v}_1 = \mathbf{1}$. It is a trivial one and thus discarded)
4: Apply k-means to group the rows of $\mathbf{V}$ into k clusters.

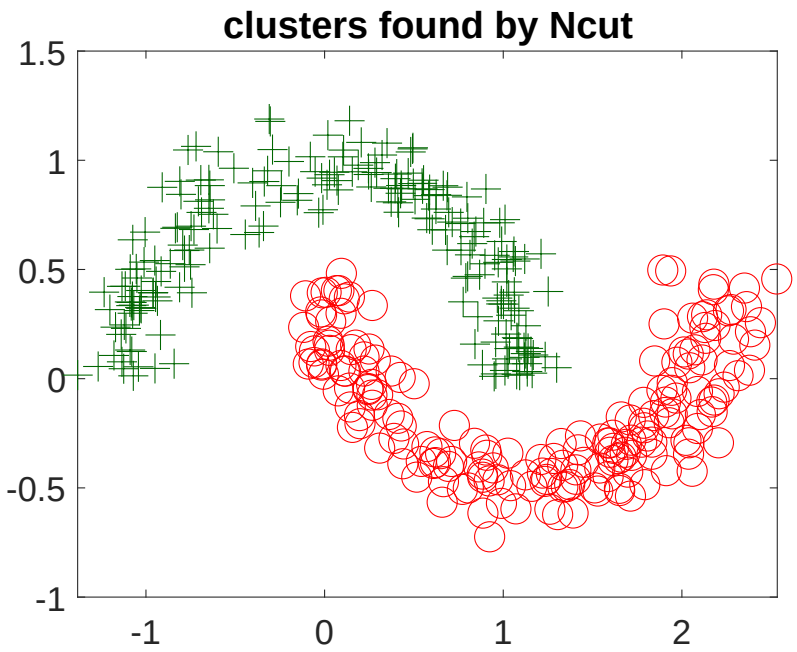
Demonstration

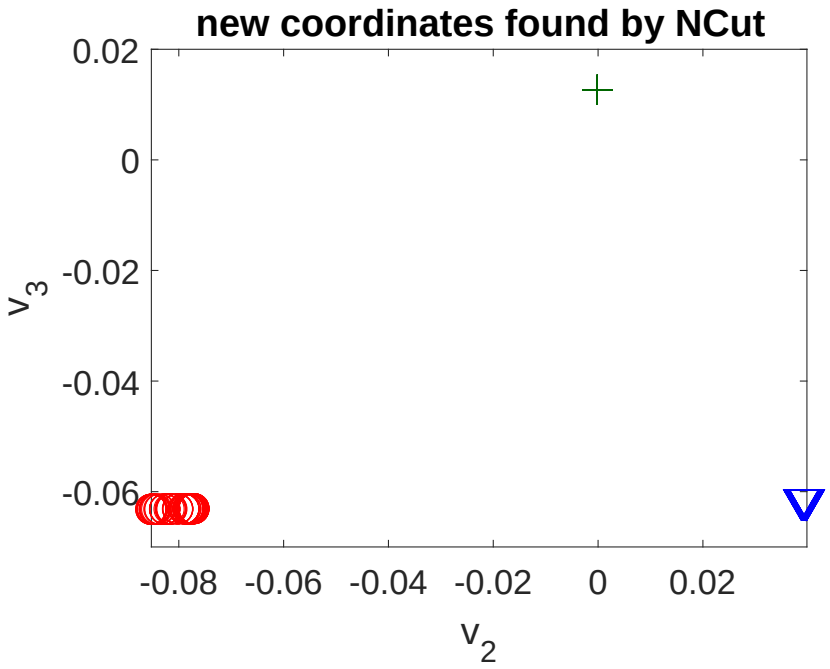
See

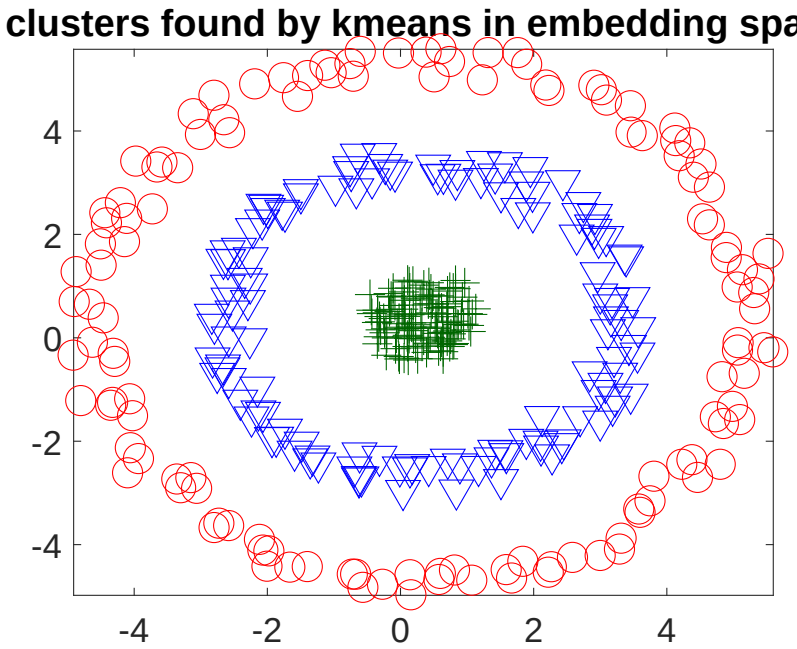
- Instructor's Python scripts for spectral clustering
- Python documentation for spectral clustering[3]

[3] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.SpectralClustering.html


second largest eigenvector
1
0.5
0
-0.5
-1
-0.04
-0.02
0
0.02
0.04
0.06
v2


clusters found by Ncut
1.5
1
0.5
0
-0.5
-1
-1
0
1
2





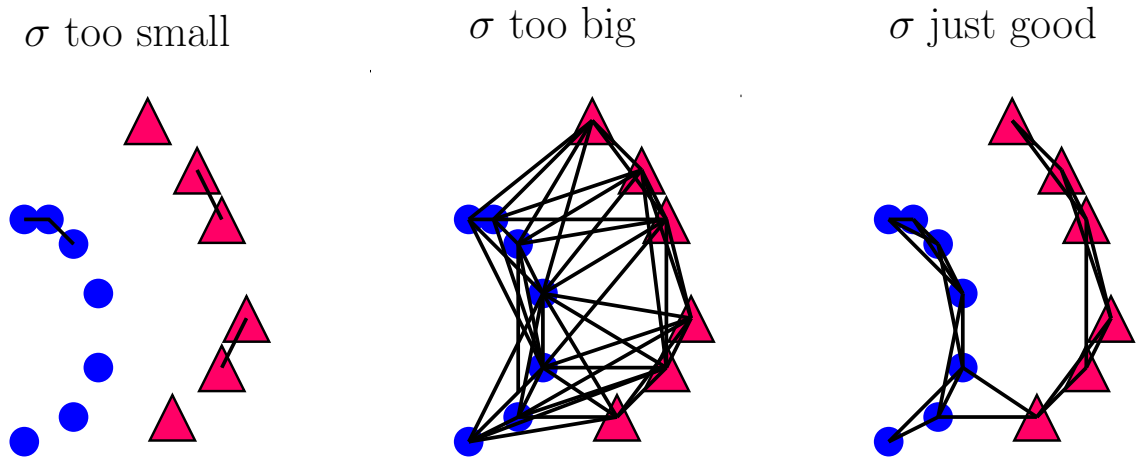
Remark. Spectral clustering = nonlinear embedding by eigenvectors + kmeans

How to set the scale parameter σ

The NCut algorithm contains a sensitive parameter (σ) whose value must be tuned carefully.

The value of σ should be set such that

- Points in the same cluster have large similarities (large σ);
- Points in different clusters have small similarities (small σ).



In practice, we can set σ to be the average distance between a data point and its rth nearest neighbor in the data set (e.g., $r = 6$).

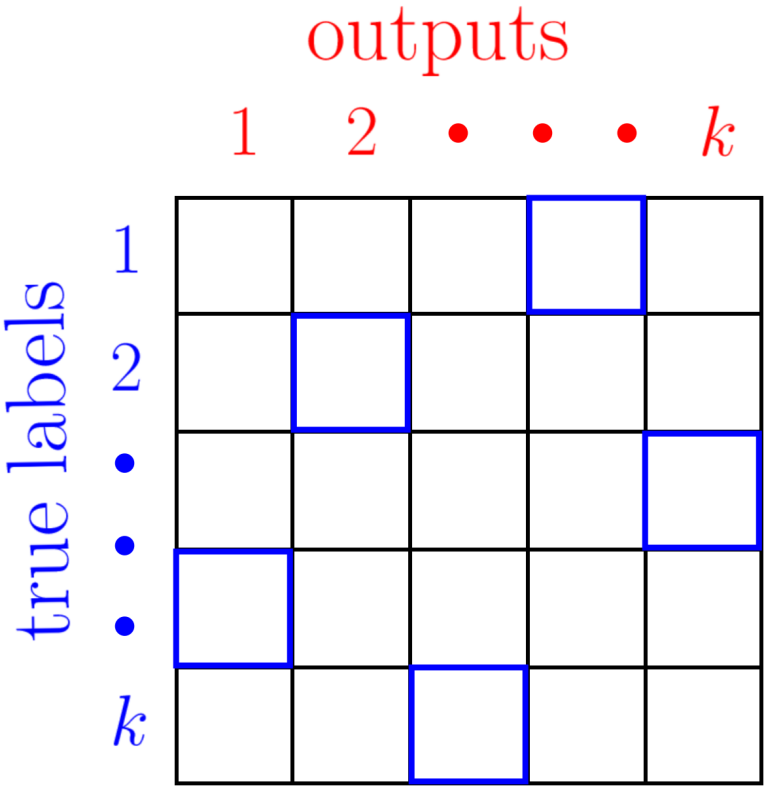
Evaluation criteria

When the true labels are given, clustering results can be evaluated as follows:

- **Clustering accuracy**

$$= \frac{\#\text{“correctly labeled” points}}{\#\text{all data points}}$$

- **Confusion matrix** $\longrightarrow\longrightarrow\longrightarrow$
- **Running time**: CPU time, or wall clock time



Computational challenges

Spectral clustering has achieved superior results in many applications (such as image segmentation, documents clustering, social network partitioning), but requires significant computational power:

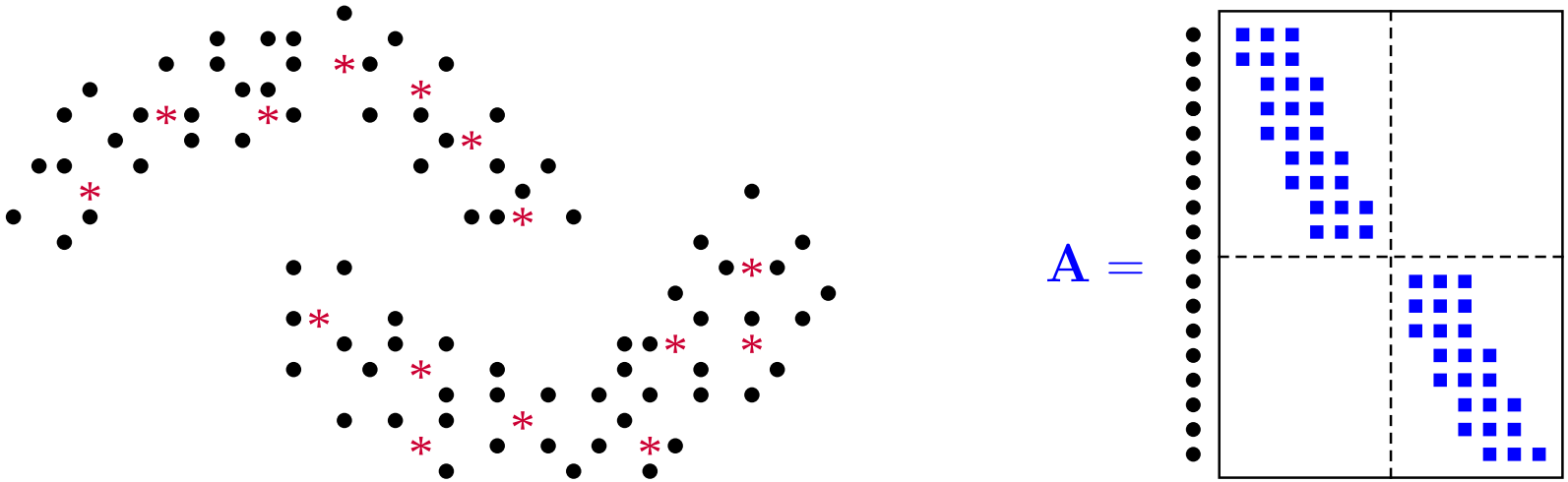
- Extensive memory requirement by $\mathbf{W} \in \mathbb{R}^{n \times n}$: $\mathcal{O}(n^2)$
- High computational cost:
  - Construction of $\mathbf{W}$: $\mathcal{O}(n^2 d)$
  - Spectral decomposition of $\mathbf{W}$: $\mathcal{O}(n^3)$

Consequently, there is a need to develop **fast**, **approximate** spectral clustering algorithms that are **scalable to large data**.

Landmark-based scalable methods

Most existing scalable methods use a small landmark set $\mathbf{y}_1, \ldots, \mathbf{y}_m \in \mathbb{R}^d$, selected from the **given data** $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$ (e.g., uniformly at random or via k-means), to construct a (sparse) similarity matrix between them:

$$\mathbf{A} = (a_{ij}) \in \mathbb{R}^{n \times m} \ (m \ll n), \quad a_{ij} = e^{-\|\mathbf{x}_i - \mathbf{y}_j\|^2/2\sigma^2} \text{ for nearby } \mathbf{y}_j$$



They mainly differ in how to use the matrix $\mathbf{A}$ to find clusters.

My own research

I have worked on landmark-based scalable spectral clustering during the past few years and have published quite a few papers (some with students).

A few open problems that I am working on now:

- The memory scalability. My previous research is on the speed scalability of spectral clustering and always assumes full access to the data. I am extending the research to the setting of massive data sets which are too large to be fully loaded into computer memory.
- Finding high quality landmark points. Currently, the two ideas both have drawbacks: uniform sampling (fast but not good quality), kmeans sampling (good quality but slow).
- New applications.

Papers I have published in this area

- "A fast incremental spectral clustering algorithm with cosine similarity", R. Li and G. Chen. *The 23rd IEEE International Conference on Data Mining (ICDM) - IncrLearn Workshop*, Shanghai, China, December 2023
- "Fast, memory-efficient spectral clustering with cosine similarity", R. Li and G. Chen. *The 26th Iberoamerican Congress on Pattern Recognition*, Coimbra, Portugal, November 2023
- "A General Framework for Scalable Spectral Clustering Based on Document Models". G. Chen. *Pattern Recognition Letters*, 125: 488-493, July 2019
- "Scalable Spectral Clustering with Cosine Similarity", G. Chen. *The 24th International Conference on Pattern Recognition (ICPR)*, Beijing, China, August 2018
- "Large-scale Spectral Clustering using Diffusion Coordinates on Landmark-based Bipartite Graphs", K. Pham and G. Chen. *The 12th Workshop on Graph-based Natural Language Processing (TextGraphs-12)*, New Orleans, Louisiana, June 2018