COMP 652: Machine Learning

Lecture 16

Today: Continuing dimensionality reduction

- Kernel PCA
- Multi-dimensional scaling
- Self-organizing maps

Recall: Principal Component analysis (PCA)

- Let $\mathbf{x}_1, \ldots \mathbf{x}_m \in \Re^n$ be the data
- Consider the *scatter matrix* (covariance matrix):

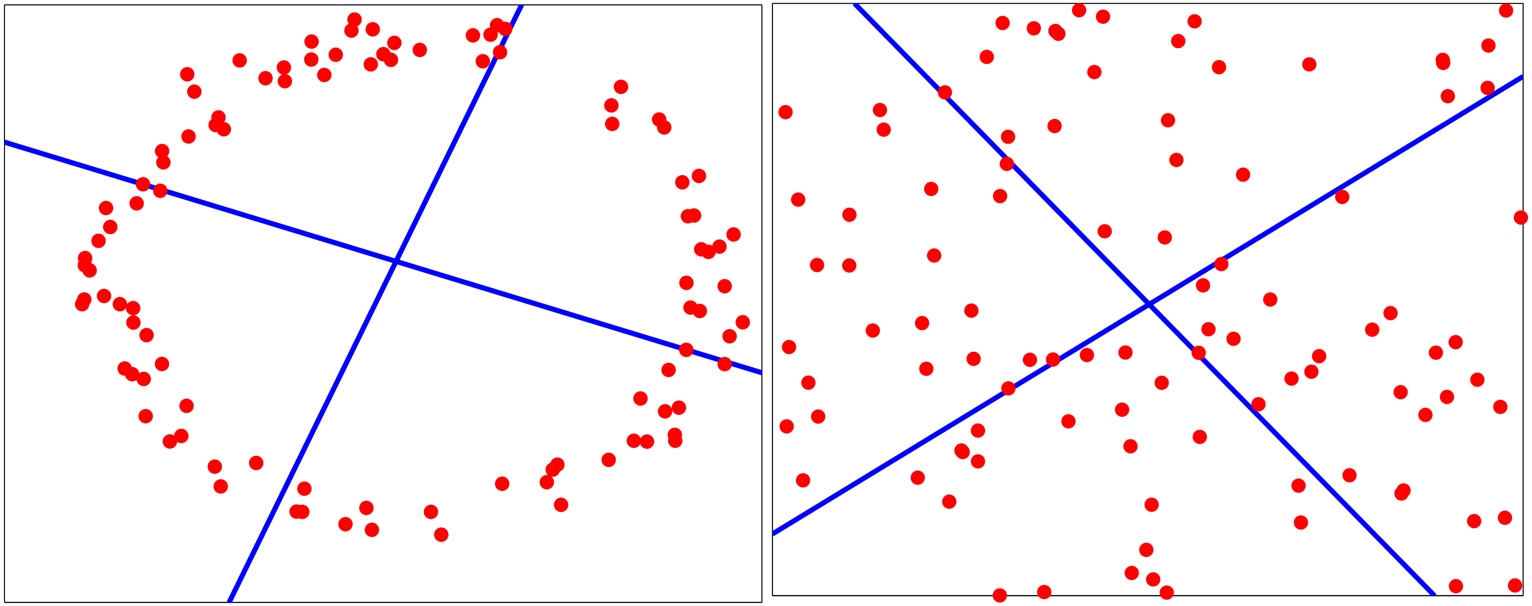
$$\mathbf{S} = \frac{1}{m} \sum_{i=1}^{m} \mathbf{x}_i \mathbf{x}_i^T$$

- The *principal components* $\mathbf{v}_j$ are the eigenvectors of $\mathbf{S}$:

$$\mathbf{S}\mathbf{v}_j = \lambda_j \mathbf{v}_j, j = 1, \ldots n$$

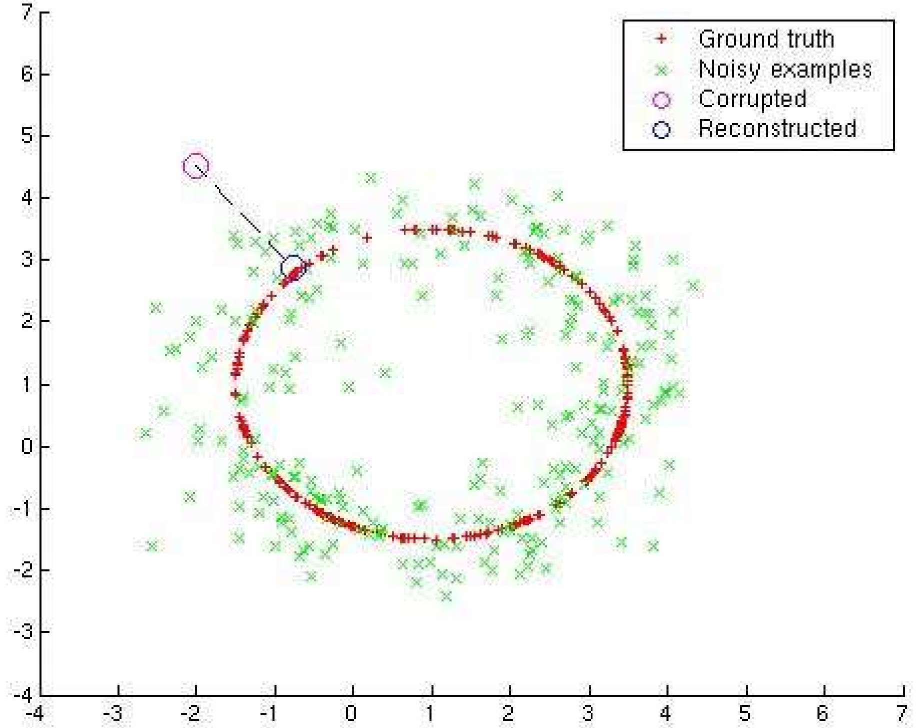
- The eigenvectors are normalized: $\mathbf{v}_i^T \mathbf{v}_i = 1$
- We sort these vectors in the decreasing order of the corresponding eigenvalues
- You can pick the first k components, or determine k based on how much variance is accounted for
- The data will be represented by projecting it onto $\mathbf{v}_i, i = 1, \ldots k$

Recall: Difficult example



PCA does equally poorly on both examples.

What we want


Ground truth
Noisy examples
Corrupted
Reconstructed
7
6
5
4
3
2
1
0
-1
-2
-3
-4
-4
-3
-2
-1
0
1
2
3
4
5
6
7

Making PCA non-linear

- Suppose that instead of using the points $\mathbf{x}_i$ as is, we wanted to go to some different *feature space* $\phi(\mathbf{x}_i) \in \Re^N$
- E.g. using polar coordinates instead of cartesian coordinates would help us deal with the circle
- In the higher dimensional space, we can then do PCA
- The result will be non-linear in the original data space!
- Similar idea to support vector machines

PCA in feature space (I)

- Suppose for the moment that the mean of the data in feature space is $\sum_{i=1}^{m} \phi(\mathbf{x}_i) = 0$
- The covariance matrix is:

$$\mathbf{C} = \frac{1}{m} \sum_{i=1}^{m} \phi(\mathbf{x}_i)\phi(\mathbf{x}_i)^T$$

- The eigenvectors are:

$$\mathbf{C}\mathbf{v}_j = \lambda_j \mathbf{v}_j, j = 1, \ldots N$$

- We want to avoid explicitly going to feature space - instead we want to work with ***kernels***:

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j)$$

PCA in feature space (II)

- Re-write the PCA equation:

$$\frac{1}{m}\sum_{i=1}^{m}\phi(\mathbf{x}_i)\phi(\mathbf{x}_i)^T\mathbf{v}_j = \lambda_j\mathbf{v}_j, j = 1, \dots N$$

- So the eigenvectors can be written as a linear combination for features:

$$\mathbf{v}_j = \sum_{i=1}^{m} a_{ji}\phi(\mathbf{x}_i)$$

- So finding the eigenvectors is equivalent to finding the coefficients $a_{ji}, j = 1, \dots N, i = 1, \dots m$

PCA in feature space (III)

- By substituting this back into the equation we get:

$$\frac{1}{m}\sum_{i=1}^{m}\phi(\mathbf{x}_i)\phi(\mathbf{x}_i)^T\left(\sum_{l=1}^{m}a_{jl}\phi(\mathbf{x}_l)\right)=\lambda_j\sum_{l=1}^{m}a_{jl}\phi(\mathbf{x}_l)$$

- We can re-write this as:

$$\frac{1}{m}\sum_{i=1}^{m}\phi(\mathbf{x}_i)\left(\sum_{l=1}^{m}a_{jl}K(\mathbf{x}_i,\mathbf{x}_l)\right)=\lambda_j\sum_{l=1}^{m}a_{jl}\phi(\mathbf{x}_l)$$

- A small trick: multiply this by $\phi(\mathbf{x}_k)^T$ to the left:

$$\frac{1}{m}\sum_{i=1}^{m}\phi(\mathbf{x}_k)^T\phi(\mathbf{x}_i)\left(\sum_{l=1}^{m}a_{jl}K(\mathbf{x}_i,\mathbf{x}_l)\right)=\lambda_j\sum_{l=1}^{m}a_{jl}\phi(\mathbf{x}_k)^T\phi(\mathbf{x}_l)$$

- By plugging in the kernel and rearranging we get: $\mathbf{K}^2\mathbf{a}_j=m\lambda_j\mathbf{K}\mathbf{a}_j$

PCA in feature space (IV)

- We can remove a factor of $\mathbf{K}$ from both sides of the matrix (the only issue is over repeated eigenvalues, which doesn't really affect things anyway):

$$\mathbf{K}\mathbf{a}_j = m\lambda_j \mathbf{a}_j$$

- We have a normalization condition for the $\mathbf{a}_j$ vectors:

$$\mathbf{v}_j^T \mathbf{v}_j = 1 \Rightarrow \sum_{k=1}^{m} \sum_{l=1}^{m} a_{jl} a_{jk} \phi(\mathbf{x}_l)^T \phi(\mathbf{x}_k) = 1 \Rightarrow \mathbf{a}_j^T \mathbf{K} \mathbf{a}_j = 1$$

- Using the above equation again we get: $\lambda_j m \mathbf{a}_j^T \mathbf{a}_j = 1, \forall j$
- For a new point $\mathbf{x}$, its projection onto the principal components is:

$$\phi(\mathbf{x})^T \mathbf{v}_j = \sum_{i=1}^{m} a_{ji} \phi(\mathbf{x})^T \phi(\mathbf{x}_i) = \sum_{i=1}^{m} a_{ji} K(\mathbf{x}, \mathbf{x}_i)$$

Normalizing the feature space

- In general, the features $\phi(\mathbf{x}_i)$ may not have mean 0
- We want to work with $\tilde{\phi}(\mathbf{x}_i) = \phi(\mathbf{x}_i) - \frac{1}{m}\sum_{k=1}^{m}\phi(\mathbf{x}_k)$
- The corresponding kernel matrix entries are given by:

$$\tilde{K}(\mathbf{x}_k, \mathbf{x}_l) = \tilde{\phi}(\mathbf{x}_l)^T \tilde{\phi}(\mathbf{x}_j)$$

- After some algebra, we get:

$$\tilde{\mathbf{K}} = \mathbf{K} - 2\mathbf{1}_{1/m}\mathbf{K} + \mathbf{1}_{1/m}\mathbf{K}\mathbf{1}_{1/m}$$

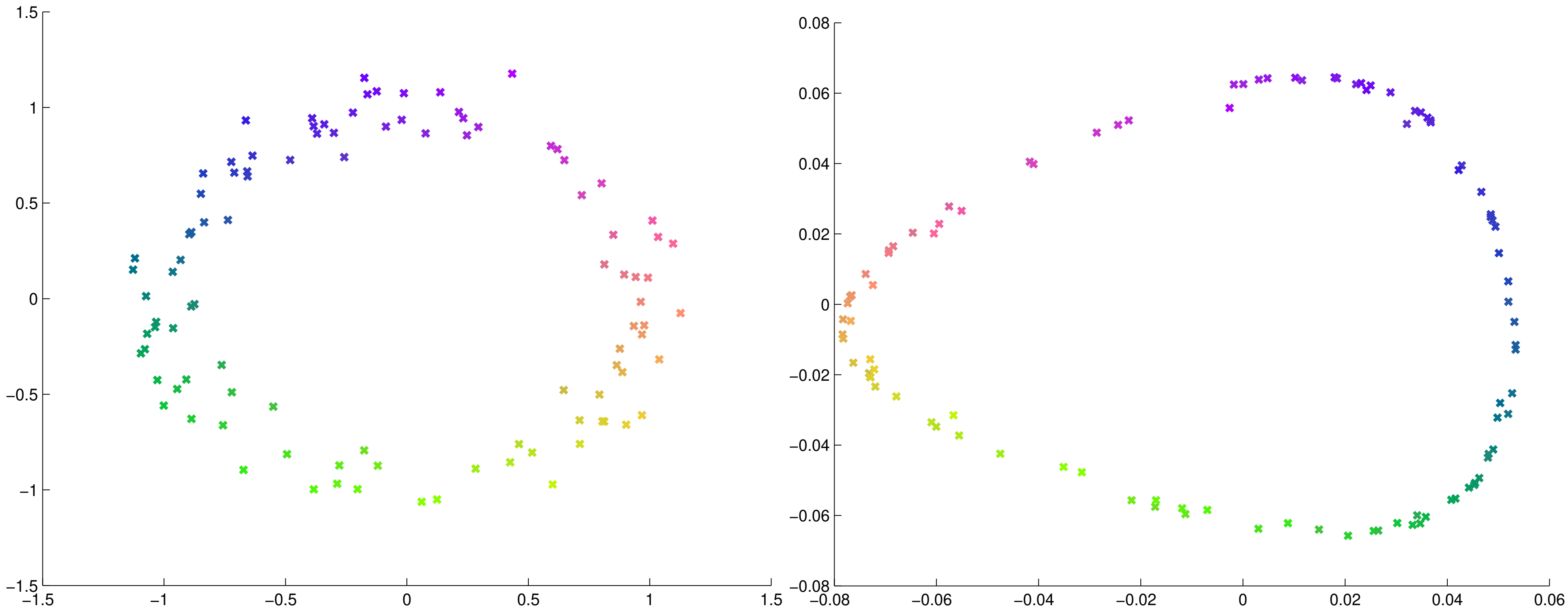
where $\mathbf{1}_{1/m}$ is the matrix with all elements equal to $1/m$

Summary of kernel PCA

1. Pick a kernel
2. Construct the normalized kernel matrix $\tilde{\mathbf{K}}$ of the data (this will be of dimension $m \times m$)
3. Find the eigenvalues and eigenvectors of this matrix λ_j, $\mathbf{a}_j$
4. For any data point (new or old), we can represent it as the following set of features:

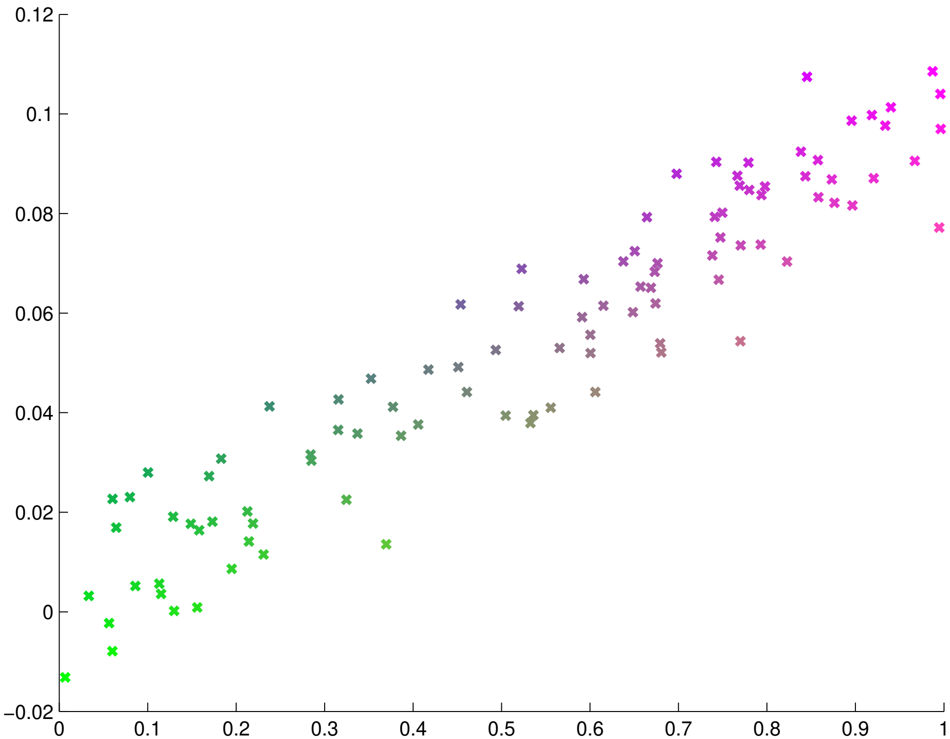
$$\mathbf{y}_j = \sum_{i=1}^{m} a_{ji} \tilde{K}(\mathbf{x}, \mathbf{x}_i), j = 1, \ldots m$$

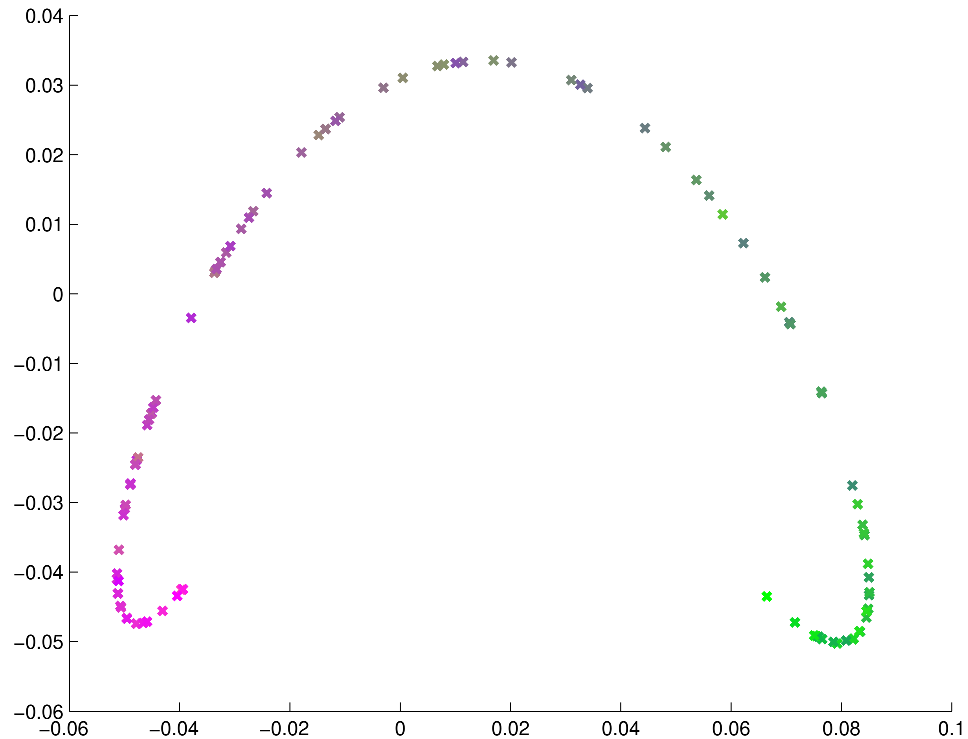
Artificial examples



- □ Gaussian kernel: $K(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2)$

Artificial examples

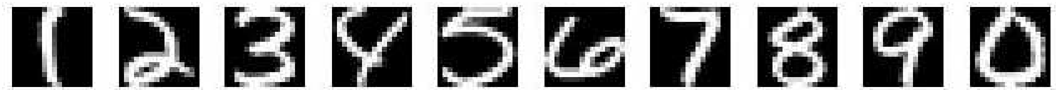




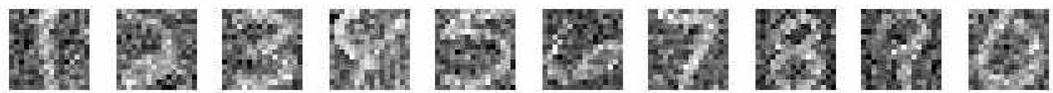
- Gaussian kernel: $K(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|^2)$

Example: De-noising images

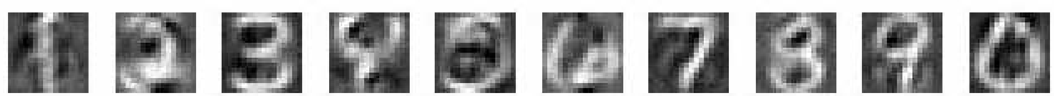
Original data



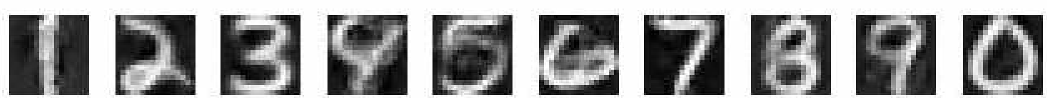
Data corrupted with Gaussian noise



Result after linear PCA



Result after kernel PCA, Gaussian kernel



PCA vs Kernel PCA

- Kernel PCA can give a good re-encoding of the data when it lies along a non-linear manifold
- The kernel matrix is $m \times m$, so kernel PCA will have difficulties if we have lots of data points
- For general kernels, we may not be able to easily visualize what the image of a point is in the input space

Multi-dimensional scaling

- Input:
  - An $m \times m$ dissimilarity matrix $\mathcal{DS}$, where $\mathcal{DS}(i,j)$ is the distance between instances $\mathbf{x}_i$ and $\mathbf{x}_j$
  - Desired dimension d of the embedding.
- Output:
  - Coordinates $\mathbf{z}_i \in \Re^d$ for each instance i that minimize a "stress" function quantifying the mismatch between distances in $\mathcal{DS}$ and distances of the data representation in $\Re^d$.

Stress functions

Common stress functions include:

- The least-squares or Kruskal-Shephard criterion:

$$\sum_{i=1}^{m} \sum_{j \neq i} (\mathcal{DS}(i,j) - \|\mathbf{z}_i - \mathbf{z}_j\|)^2$$

- The Sammon mapping:

$$\sum_{i=1}^{m} \sum_{j \neq i} \frac{(\mathcal{DS}(i,j) - \|\mathbf{z}_i - \mathbf{z}_j\|)^2}{\mathcal{DS}(i,j)} \ ,$$

  which emphasizes getting small distances correct.

Gradient-based optimization is usually used to find $\mathbf{z}_i$

Self-organizing maps

- If the instances are vectors in $\Re^n$, try to stretch a "grid" of points in n dimensions to approximate the data.
- The indices of the grid points indicate *neighborhood* relationships
- E.g., in 2D, $G(i, j)$ is neighbor with $G(i-1, j)$, $G(i+1, j)$, $G(i, j-1)$, $G(i, j+1)$.
- The grid points are iteratively moved, "pulled", by data points, similar to how the centroids of K-means clustering move around.
- The data can then be visualized by mapping each object to the nearest grid point.

Self-organizing maps

- Inputs:
  - A set $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ of n-dimensional real vectors.
  - A dimension for the grid (1,2 or 3 if we want to plot it.)
  - Number of grid points along each dimension.
- Output: Coordinates G in $\Re^n$ for each grid-point.

SOM learning algorithm

- Initialize the grid points.
- Repeat
  - Choose a data point $\mathbf{x}$ at random.
  - Find the nearest grid point; e.g., in 2D:

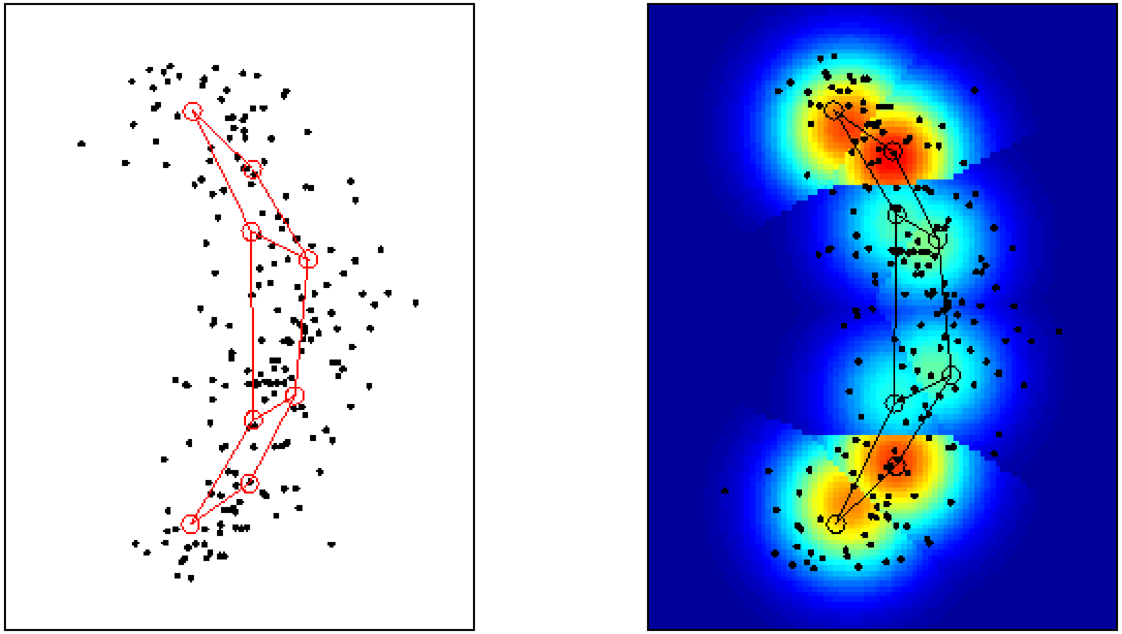
$$G(i^*, j^*) = \arg\min_{i,j} \|G(i,j) - \mathbf{x}\|$$

  - Find the "neighborhood" of $G^*(i,j)$
  - Move all points $\mathbf{G}$ in the neighborhood towards $\mathbf{x}$:

$$\mathbf{G} \leftarrow \mathbf{G} + \alpha s(\mathbf{x}, \mathbf{G})(\mathbf{x} - \mathbf{G})$$

    where $s(\mathbf{x}, \mathbf{G})$ is a similarity function, equal to 1 if $\mathbf{x} = \mathbf{G}$ and decreasing with $\|\mathbf{x} - \mathbf{G}\|$ (e.g. Gaussian)

Example



Remarks

- Typically the learning rate $\alpha \to 0$ with time
- The SOM builds a *topographical map* of the input space, putting more points where the data is dense
- Instances that are close in the input space will be mapped to units which are neighbors in the grid.
- If the data approximately lies on a curve or surface, the SOM may capture that structure, but:
  - Different runs can find different solutions.
  - If we try to fit data on a 2D surface with a 1D grid, well...
- More sophisticated versions of SOMs use different updating rules, different neighboring functions