Social Data Science: Econometrics and Machine Learning

Week 3

Unsupervised machine learning

Overview

1. **Dimensionality reduction**
   a. Principal Component Analysis
   b. Linear Discriminant Analysis
   c. Quick word on t-SNE and UMAP (not in book)
2. **Clustering**
   a. K-means
   b. C-means
   c. Gaussian mixture models
   d. Evaluating quality of a clustering
      - Elbow
      - Silhuette
   e. Linkage clustering
   f. DBSCAN
3. **Further topics**

Dimensionality reduction

Dimensionality reduction

Gist:

- Many datasets have enormous feature spaces (i.e. many dimensions)
- Often there is high correlation between features
- We want to express the data in terms of fewer features without losing too much variance

Pros:

1. Often enables visualization of very high-dimensional data
2. Less data to store
3. Can improve predictive performance by reducing *curse of dimensionality* problems

Cons:

1. Features are harder/impossible to interpret
2. We lose some variance/information

Dimensionality reduction

> Different methods for different things

Principal Component Analysis → **Linear, unsupervised**

Linear Discriminant Analysis → **Linear, supervised**

Quick word on t-SNE and UMAP → **Non-linear, unsupervised**

Dimensionality reduction

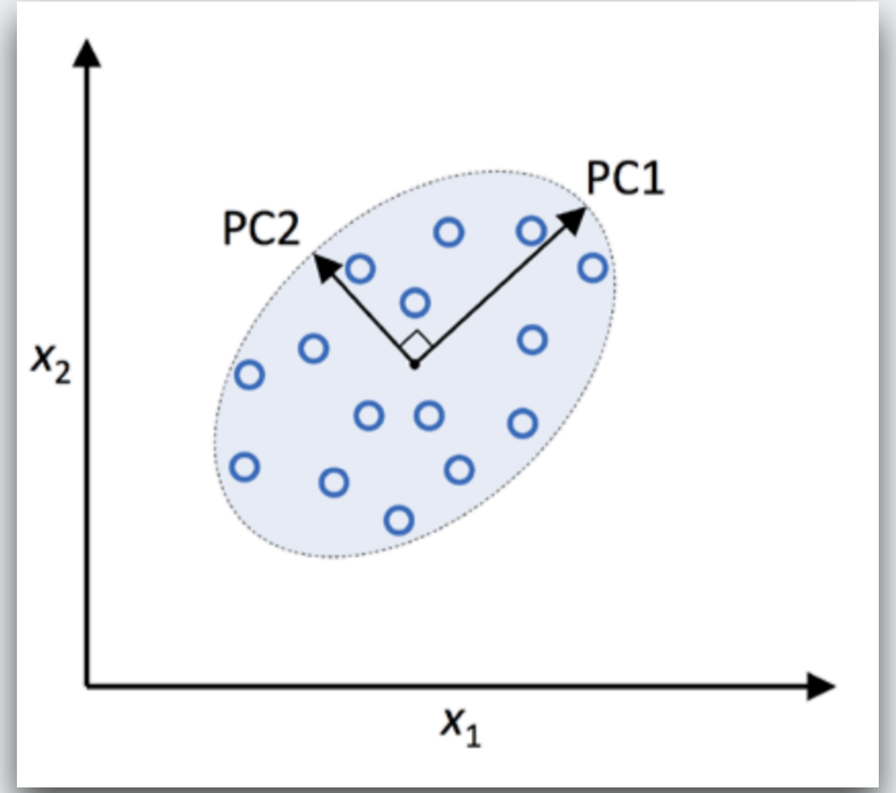
> Principal Component Analysis

Main idea:

- Find the directions of max variance and *rotate* the data so PC1 becomes new x-axis and PC2 becomes new y-axis, and so on…

- *Example:* You have $\mathbf{x} = [x_1, x_2, \ldots, x_d]$, then you find the basis transformation matrix $\mathbf{W}$ that allows you to set $\mathbf{z} = \mathbf{xW} = [z_1, z_2, \ldots, z_k]$

- Since the last PCs will have relatively little data variance, you can just remove them! *Success*: you have reduced the dimensionality of $\mathbf{x}$!



Dimensionality reduction

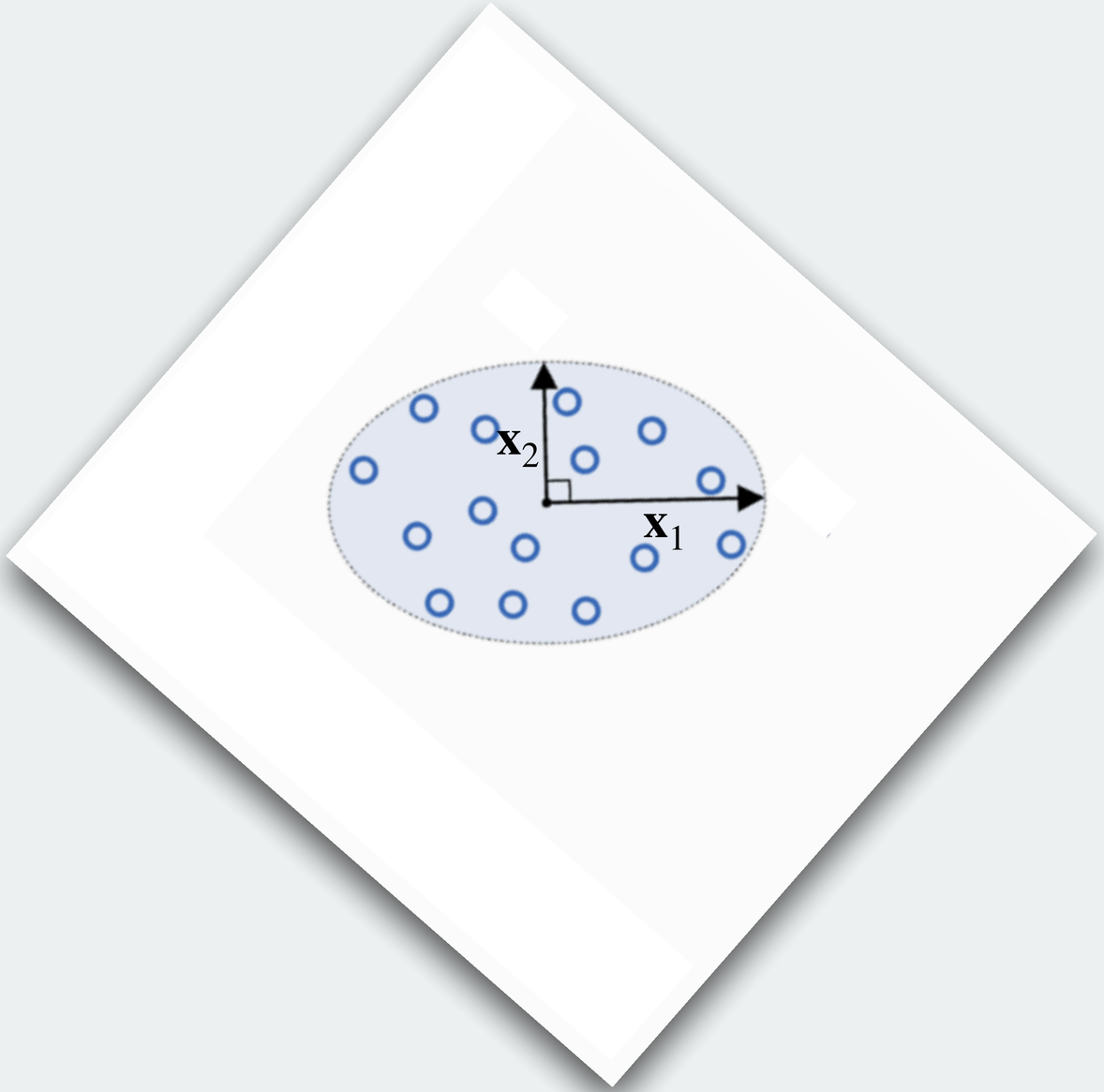
> Principal Component Analysis

Main idea:

- Find the directions of max variance and *rotate* the data so PC1 becomes new x-axis and PC2 becomes new y-axis, and so on…

- *Example:* You have $\mathbf{x} = [x_1, x_2, \ldots, x_d]$, then you find the basis transformation matrix $\mathbf{W}$ that allows you to set $\mathbf{z} = \mathbf{xW} = [z_1, z_2, \ldots, z_k]$

- Since the last PCs will have relatively little data variance, you can just remove them! *Success*: you have reduced the dimensionality of $\mathbf{x}$!



Dimensionality reduction

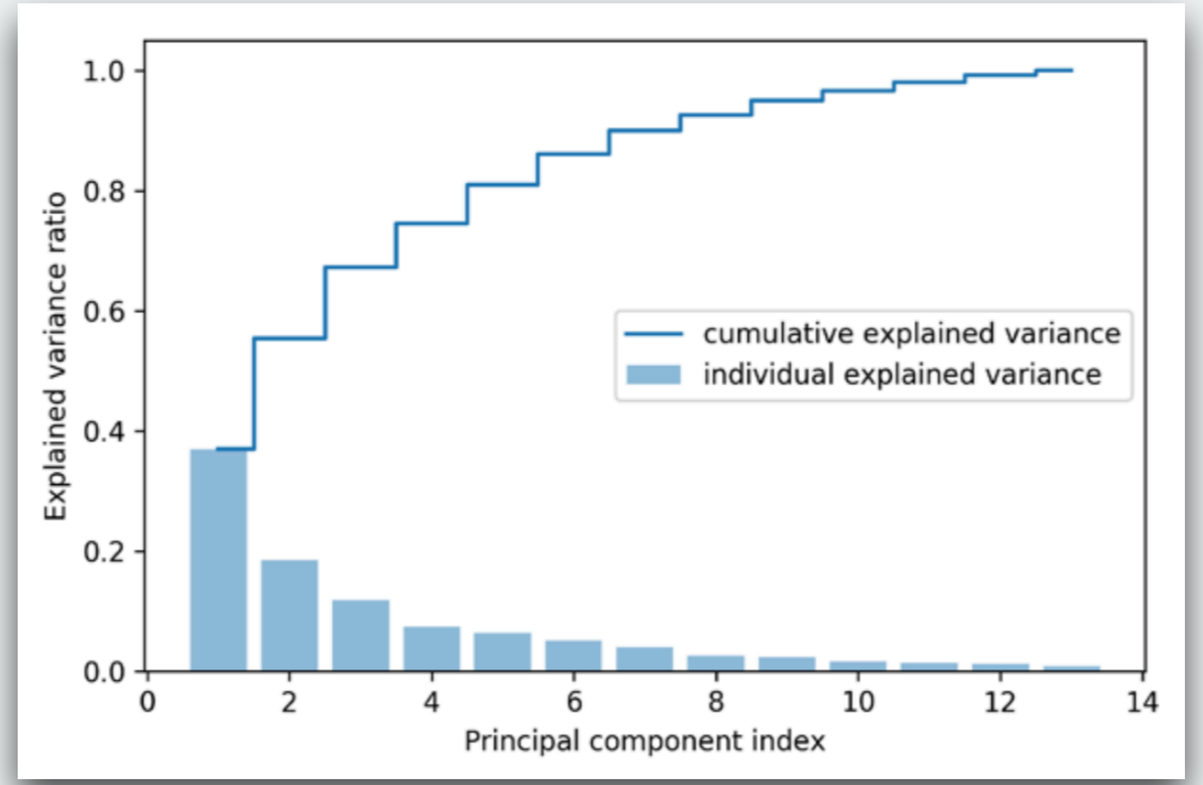
> Principal Component Analysis

Main idea:

- Find the directions of max variance and *rotate* the data so PC1 becomes new x-axis and PC2 becomes new y-axis, and so on…

- *Example:* You have $\mathbf{x} = [x_1, x_2, \ldots, x_d]$, then you find the basis transformation matrix $\mathbf{W}$ that allows you to set $\mathbf{z} = \mathbf{xW} = [z_1, z_2, \ldots, z_k]$

- Since the last PCs will have relatively little data variance, you can just remove them! *Success*: you have reduced the dimensionality of $\mathbf{x}$!



Raschka, 2017

Dimensionality reduction

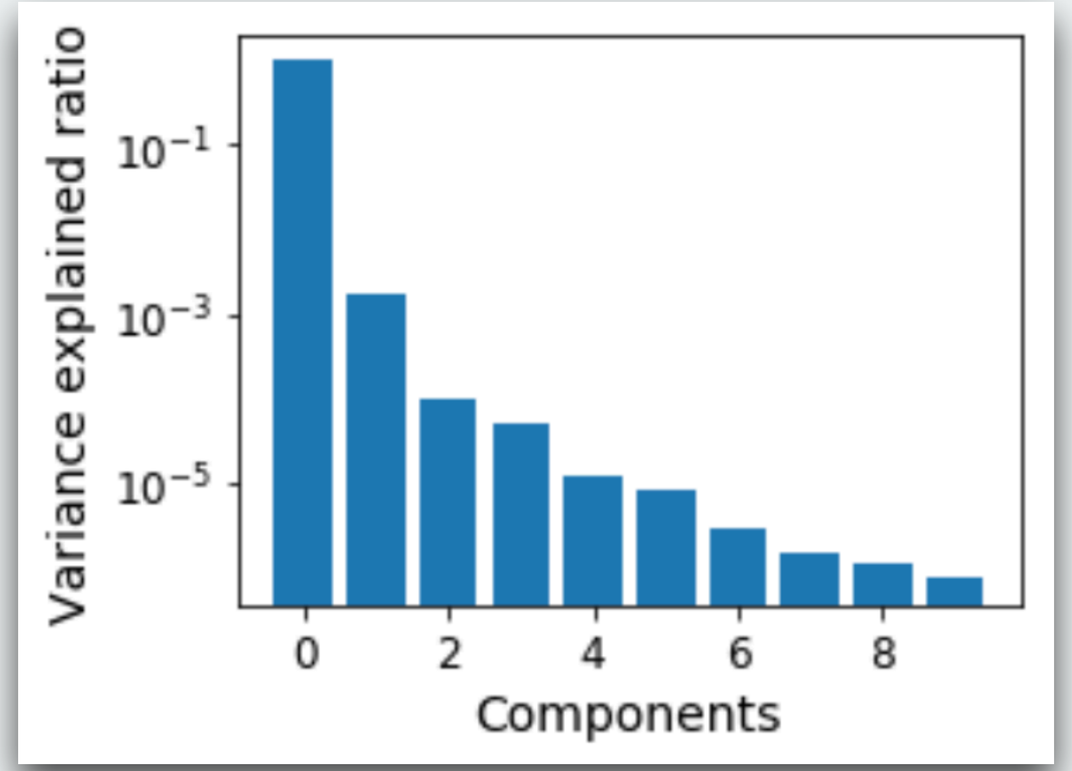
> Principal Component Analysis

Main idea:

- Find the directions of max variance and *rotate* the data so PC1 becomes new x-axis and PC2 becomes new y-axis, and so on…

- *Example:* You have $\mathbf{x} = [x_1, x_2, \ldots, x_d]$, then you find the basis transformation matrix $\mathbf{W}$ that allows you to set $\mathbf{z} = \mathbf{xW} = [z_1, z_2, \ldots, z_k]$

- Since the last PCs will have relatively little data variance, you can just remove them! *Success*: you have reduced the dimensionality of $\mathbf{x}$!



https://archive.ics.uci.edu/ml/datasets/Wine

Dimensionality reduction

> Principal Component Analysis

Algorithm:

1. **Standardize** the d-dimensional dataset.
2. Construct the **covariance matrix**.
3. **Decompose** the covariance matrix into its eigenvectors and eigenvalues.
4. **Sort** the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. **Select k eigenvectors** which correspond to the **k largest eigenvalues**, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a **projection matrix W** from the "top" k eigenvectors [Ulf: horizontally stack the (vertical) eigenvectors].
7. **Transform** the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature subspace.

1
```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_std = sc.fit_transform(X)
```

Dimensionality reduction

> Principal Component Analysis

Algorithm:

1. **Standardize** the d-dimensional dataset.
2. Construct the **covariance matrix**.
3. **Decompose** the covariance matrix into its eigenvectors and eigenvalues.
4. **Sort** the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. **Select k eigenvectors** which correspond to the **k largest eigenvalues**, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a **projection matrix W** from the "top" k eigenvectors [Ulf: horizontally stack the (vertical) eigenvectors].
7. **Transform** the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature subspace.

1
```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_std = sc.fit_transform(X)
```

2
```
>>> cov_mat = np.cov(X_std.T)
```

Dimensionality reduction

> Principal Component Analysis

Algorithm:

1. **Standardize** the d-dimensional dataset.
2. Construct the **covariance matrix**.
3. **Decompose** the covariance matrix into its eigenvectors and eigenvalues.
4. **Sort** the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. **Select k eigenvectors** which correspond to the **k largest eigenvalues**, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a **projection matrix W** from the "top" k eigenvectors [Ulf: horizontally stack the (vertical) eigenvectors].
7. **Transform** the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature subspace.

1
```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_std = sc.fit_transform(X)
```

2
```
>>> cov_mat = np.cov(X_std.T)
```

3
```
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
```

Dimensionality reduction

> Principal Component Analysis

Algorithm:

1. **Standardize** the d-dimensional dataset.
2. Construct the **covariance matrix**.
3. **Decompose** the covariance matrix into its eigenvectors and eigenvalues.
4. **Sort** the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. **Select k eigenvectors** which correspond to the **k largest eigenvalues**, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a **projection matrix W** from the "top" k eigenvectors [Ulf: horizontally stack the (vertical) eigenvectors].
7. **Transform** the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature subspace.

1
```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_std = sc.fit_transform(X)
```

2
```
>>> cov_mat = np.cov(X_std.T)
```

3
```
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
```

4
```
>>> eigen_vals, eigen_vecs = sorted(
    zip(eigen_vals, eigen_vecs),
    key=lambda kv: kv[0], reverse=True
)
```

Dimensionality reduction

> Principal Component Analysis

Algorithm:

1. **Standardize** the d-dimensional dataset.
2. Construct the **covariance matrix**.
3. **Decompose** the covariance matrix into its eigenvectors and eigenvalues.
4. **Sort** the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. **Select k eigenvectors** which correspond to the **k largest eigenvalues**, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a **projection matrix W** from the "top" k eigenvectors [Ulf: horizontally stack the (vertical) eigenvectors].
7. **Transform** the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature subspace.

1
```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_std = sc.fit_transform(X)
```

2
```
>>> cov_mat = np.cov(X_std.T)
```

3
```
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
```

4
```
>>> eigen_vals, eigen_vecs = sorted(
    zip(eigen_vals, eigen_vecs),
    key=lambda kv: kv[0], reverse=True
)
```

5
```
>>> top_k_eigen_vecs = eigen_vecs[:k]
```

Dimensionality reduction

> Principal Component Analysis

Algorithm:

1. **Standardize** the d-dimensional dataset.
2. Construct the **covariance matrix**.
3. **Decompose** the covariance matrix into its eigenvectors and eigenvalues.
4. **Sort** the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. **Select k eigenvectors** which correspond to the **k largest eigenvalues**, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a **projection matrix W** from the "top" k eigenvectors [Ulf: horizontally stack the (vertical) eigenvectors].
7. **Transform** the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature subspace.

1
```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_std = sc.fit_transform(X)
```

2
```
>>> cov_mat = np.cov(X_std.T)
```

3
```
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
```

4
```
>>> eigen_vals, eigen_vecs = sorted(
    zip(eigen_vals, eigen_vecs),
    key=lambda kv: kv[0], reverse=True
)
```

5
```
>>> top_k_eigen_vecs = eigen_vecs[:k]
```

6
```
>>> W = np.hstack([
    w.reshape(-1, 1) for w in top_k_eigen_vecs
])
```

Dimensionality reduction

> Principal Component Analysis

Algorithm:

1. **Standardize** the d-dimensional dataset.
2. Construct the **covariance matrix**.
3. **Decompose** the covariance matrix into its eigenvectors and eigenvalues.
4. **Sort** the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. **Select k eigenvectors** which correspond to the **k largest eigenvalues**, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a **projection matrix W** from the "top" k eigenvectors [Ulf: horizontally stack the (vertical) eigenvectors].
7. **Transform** the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature subspace.

1
```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_std = sc.fit_transform(X)
```

2
```
>>> cov_mat = np.cov(X_std.T)
```

3
```
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
```

4
```
>>> eigen_vals, eigen_vecs = zip(*sorted(
    zip(abs(eigen_vals), eigen_vecs.T),
    key=lambda kv: kv[0], reverse=True
))
```

5
```
>>> top_k_eigen_vecs = eigen_vecs[:k]
```

6
```
>>> W = np.hstack([
    w.reshape(-1, 1) for w in top_k_eigen_vecs
])
```

7
```
>>> Z = np.dot(X_std, W)
```

Dimensionality reduction

> Principal Component Analysis

Algorithm:

1. **Standardize** the d-dimensional dataset.
2. Construct the **covariance matrix**.
3. **Decompose** the covariance matrix into its eigenvectors and eigenvalues.
4. **Sort** the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. **Select k eigenvectors** which correspond to the **k largest eigenvalues**, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a **projection matrix W** from the "top" k eigenvectors [Ulf: horizontally stack the (vertical) eigenvectors].
7. **Transform** the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature subspace.

with sklearn

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA()
>>> Z = pca.fit_transform(X_std)
```

Dimensionality reduction

> (Fisher's) Linear Discriminant Analysis

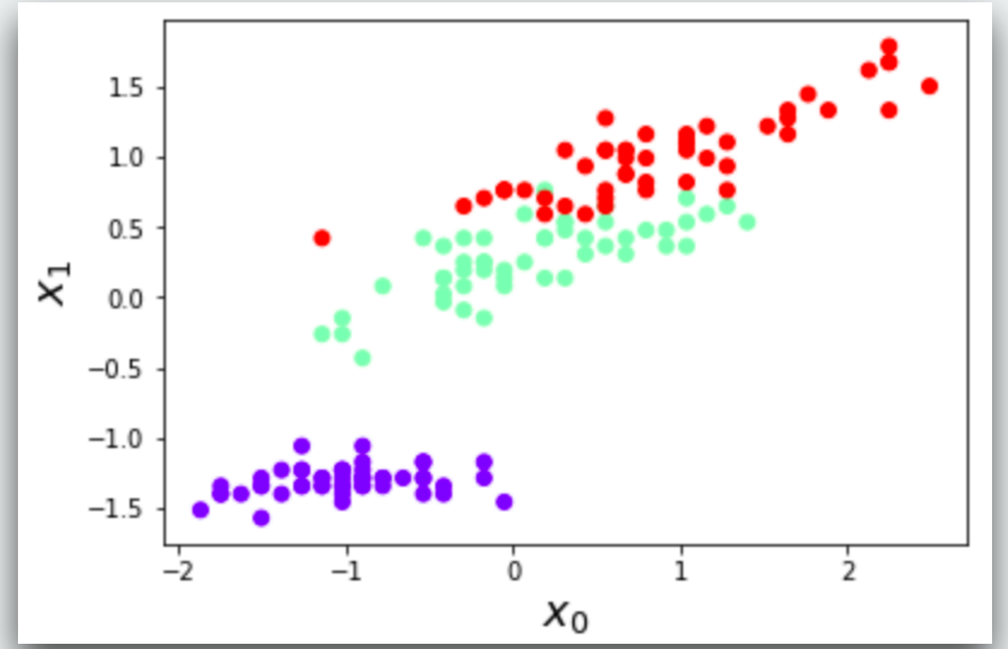
Main idea:

- Find a low-dimensional projection that maximizes the **seperation of classes**.

same as PCA

- *Example:* You have $\mathbf{x} = [x_1, x_2, \ldots, x_d]$, then you find the basis transformation matrix $\mathbf{W}$ that allows you to set $\mathbf{z} = \mathbf{xW} = [z_1, z_2, \ldots, z_k]$

- Always gives you a better sense of how separable classes are, compared to PCA



Dimensionality reduction

> (Fisher's) Linear Discriminant Analysis

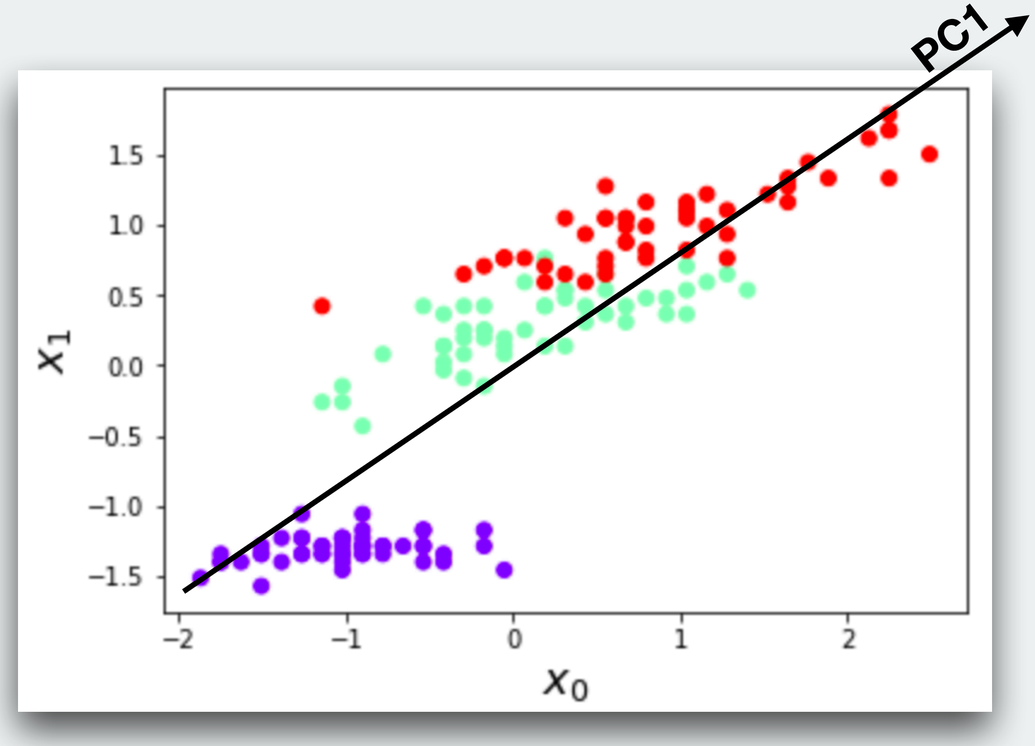
Main idea:

- Find a low-dimensional projection that maximizes the **seperation of classes**.

same as PCA

- *Example:* You have $\mathbf{x} = [x_1, x_2, \ldots, x_d]$, then you find the basis transformation matrix $\mathbf{W}$ that allows you to set $\mathbf{z} = \mathbf{xW} = [z_1, z_2, \ldots, z_k]$

- Always gives you a better sense of how separable classes are, compared to PCA



Dimensionality reduction

> (Fisher's) Linear Discriminant Analysis

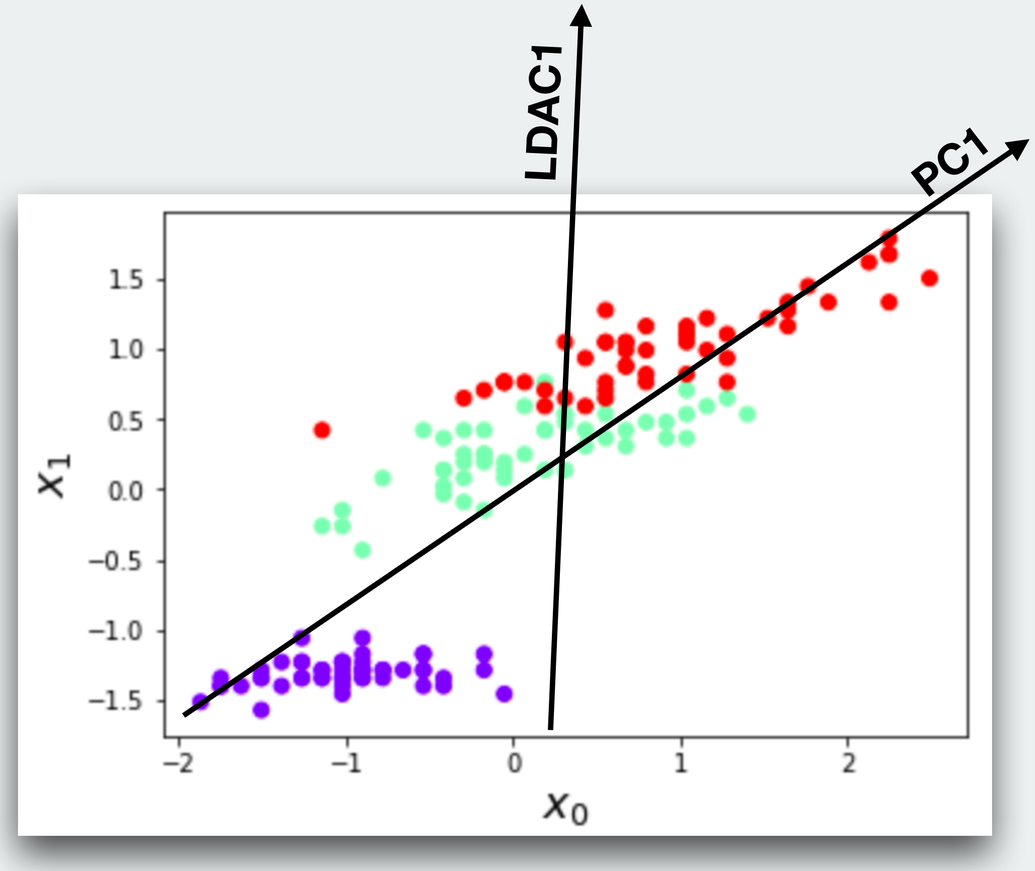
Main idea:

- Find a low-dimensional projection that maximizes the **seperation of classes**.

same as PCA

- *Example:* You have $\mathbf{x} = [x_1, x_2, \ldots, x_d]$, then you find the basis transformation matrix $\mathbf{W}$ that allows you to set $\mathbf{z} = \mathbf{xW} = [z_1, z_2, \ldots, z_k]$

- Always gives you a better sense of how separable classes are, compared to PCA



with sklearn

```
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
>>> lda = LinearDiscriminantAnalysis()
>>> X_lda = lda.fit_transform(X)
```

Dimensionality reduction

> (Fisher's) Linear Discriminant Analysis

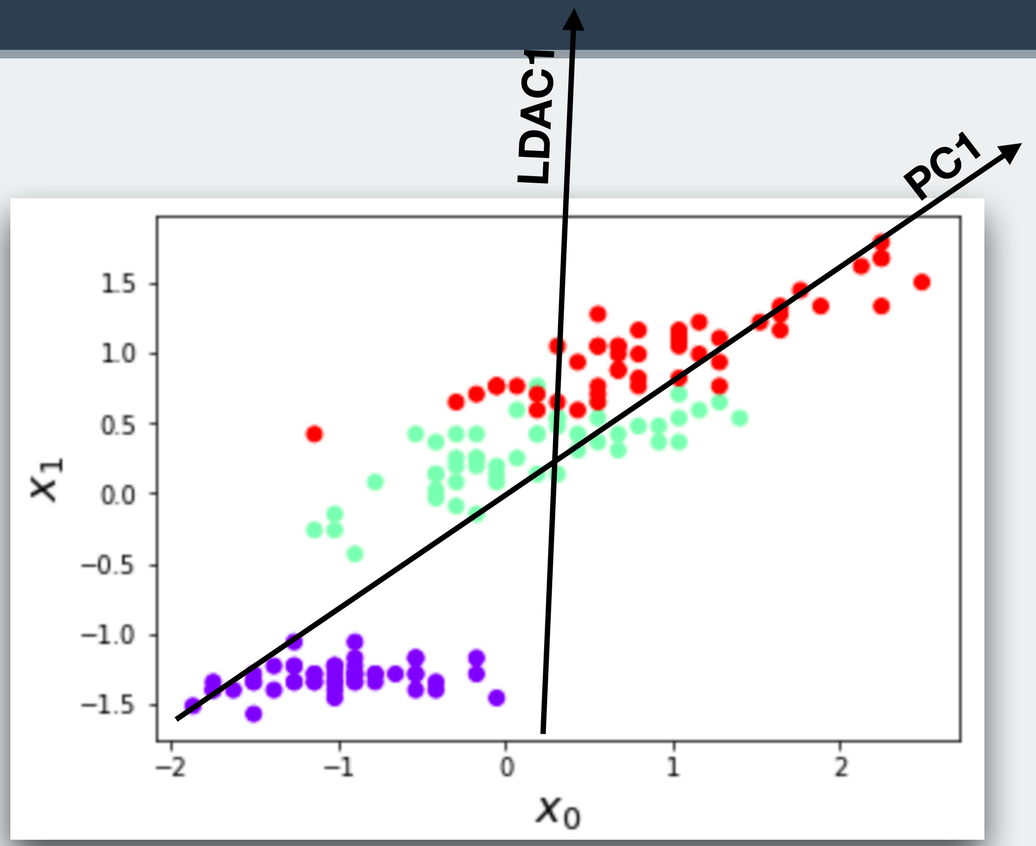
Main idea:

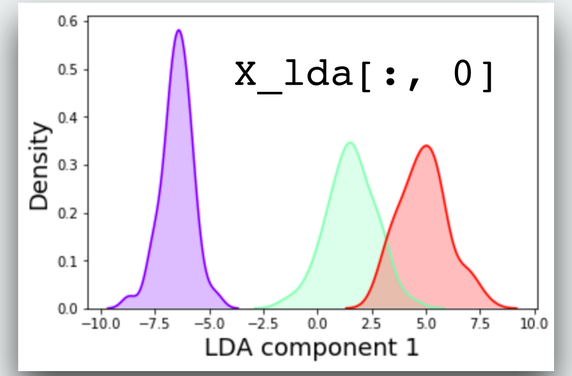
- Find a low-dimensional projection that maximizes the **seperation of classes**.

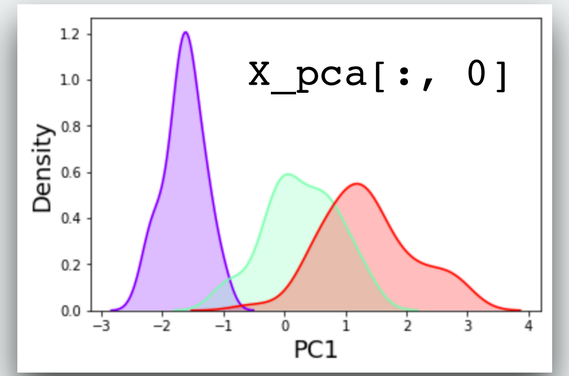
same as PCA

- *Example:* You have $\mathbf{x} = [x_1, x_2, \ldots, x_d]$, then you find the basis transformation matrix $\mathbf{W}$ that allows you to set $\mathbf{z} = \mathbf{xW} = [z_1, z_2, \ldots, z_k]$

- Always gives you a better sense of how separable classes are, compared to PCA







Dimensionality reduction

> Nonlinear unsupervised: t-SNE and UMAP

Main idea:

- In very high-dimensional data, **rarely** can we find a linear projection which gives us a sense of how points are distributed in higher-dimensions.

- BUT! If we **forget linearity** (i.e. allow projection onto any kind of surface/manifold) we can put points close to each other in the projection if they are close in the original data space and vice versa.

- t-SNE: "*minimizes the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding*"

Dimensionality reduction

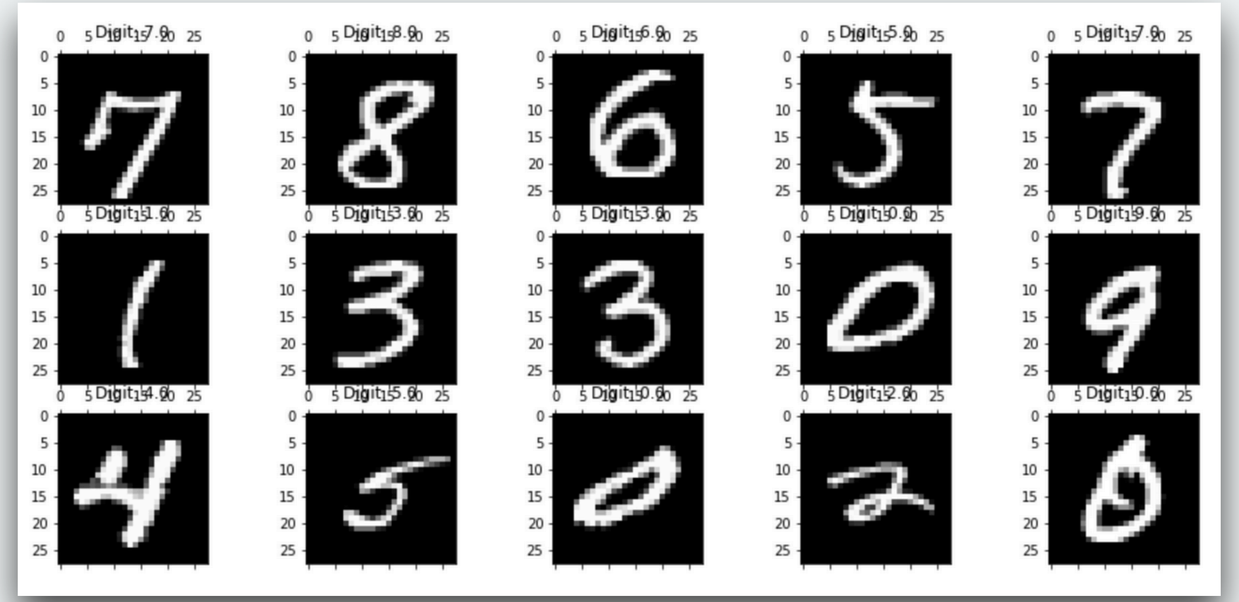
> Nonlinear unsupervised: t-SNE and UMAP

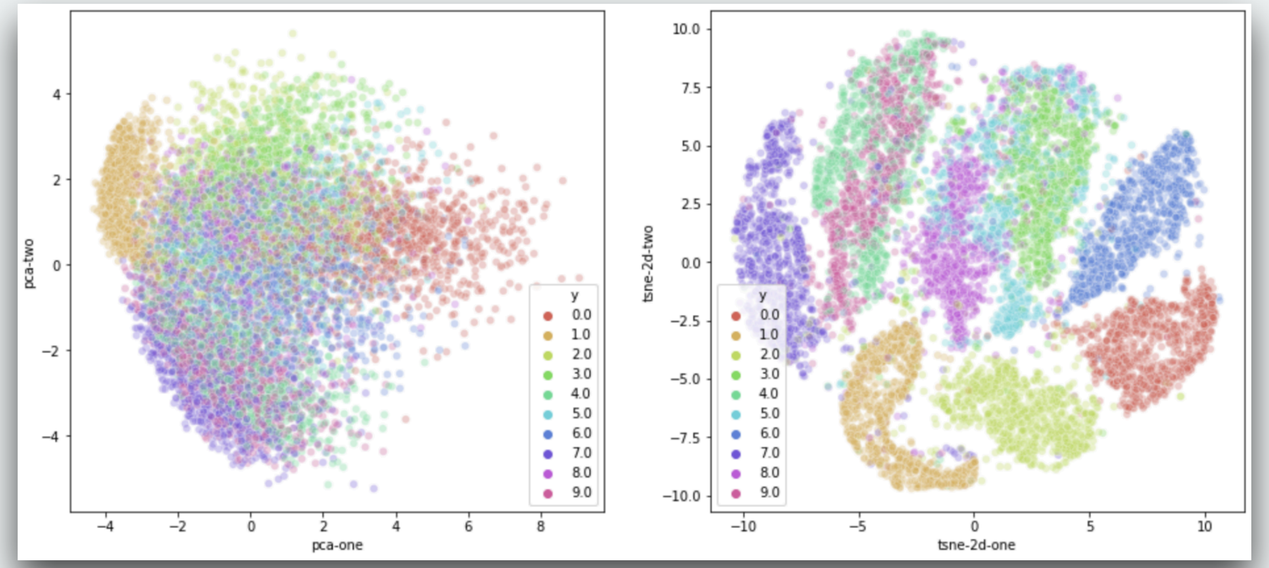
Main idea:

- In very high-dimensional data, **rarely** can we find a linear projection which gives us a sense of how points are distributed in higher-dimensions.

- BUT! If we **forget linearity** (i.e. allow projection onto any kind of surface/manifold) we can put points close to each other in the projection if they are close in the original data space and vice versa.

- t-SNE: "*minimizes the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding*"

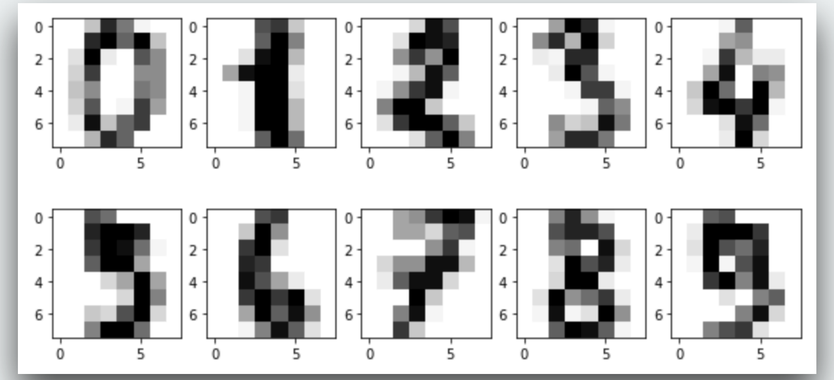




Dimensionality reduction

> Comparing PCA, LDA, t-SNE and UMAP

dataset: 8x8 pixel b/w images

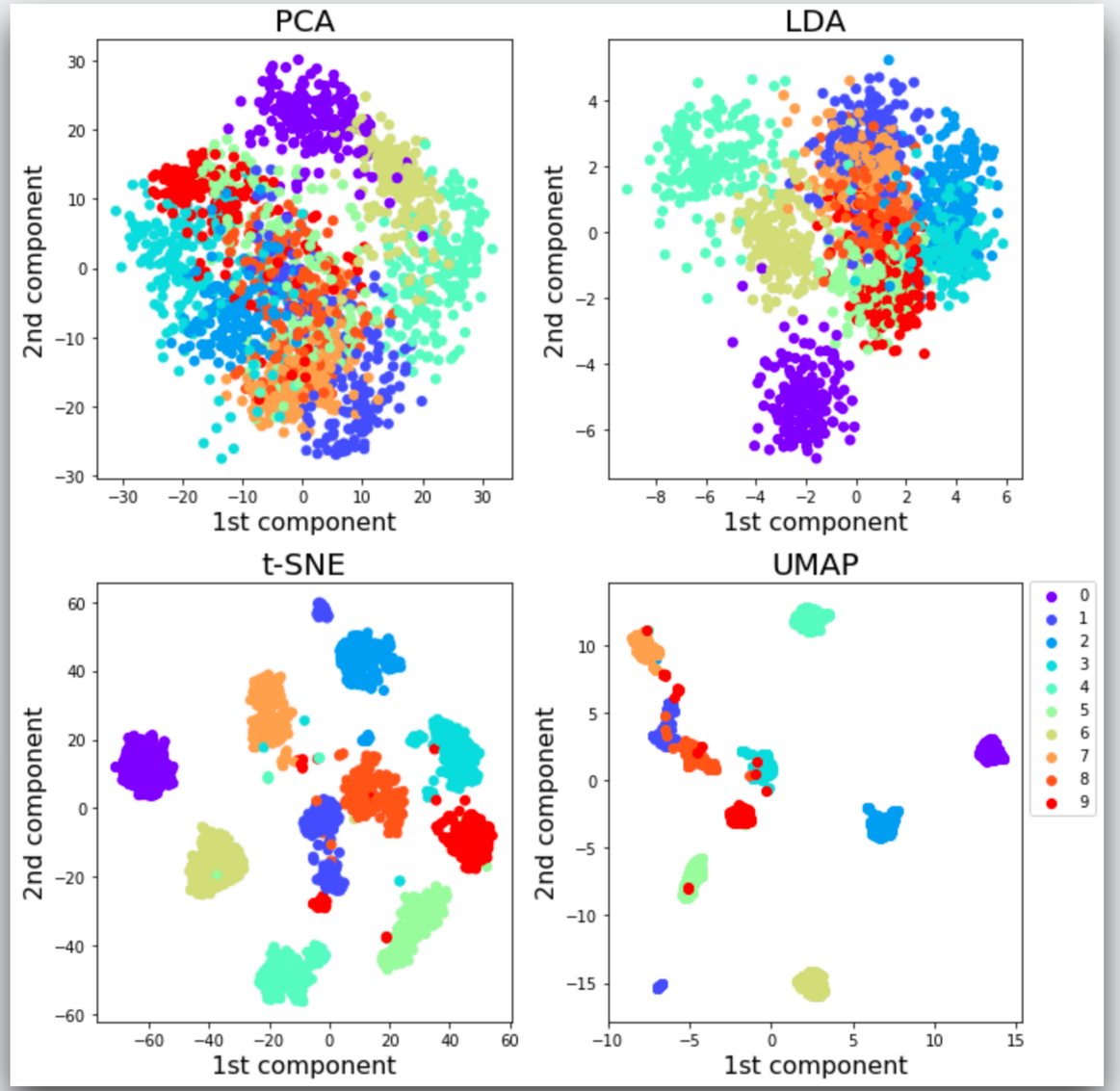


... reshaped to 64 values long arrays

```
1  X[0]
```

executed in 6ms, finished 20:34:22 2020-02-20

```
array([ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.,  0.,  0., 13., 15., 10.,
       15.,  5.,  0.,  0.,  3., 15.,  2.,  0., 11.,  8.,  0.,  0.,  4.,
       12.,  0.,  0.,  8.,  8.,  0.,  0.,  5.,  8.,  0.,  0.,  9.,  8.,
        0.,  0.,  4., 11.,  0.,  1., 12.,  7.,  0.,  0.,  2., 14.,  5.,
       10., 12.,  0.,  0.,  0.,  0.,  6., 13., 10.,  0.,  0.,  0.])
```

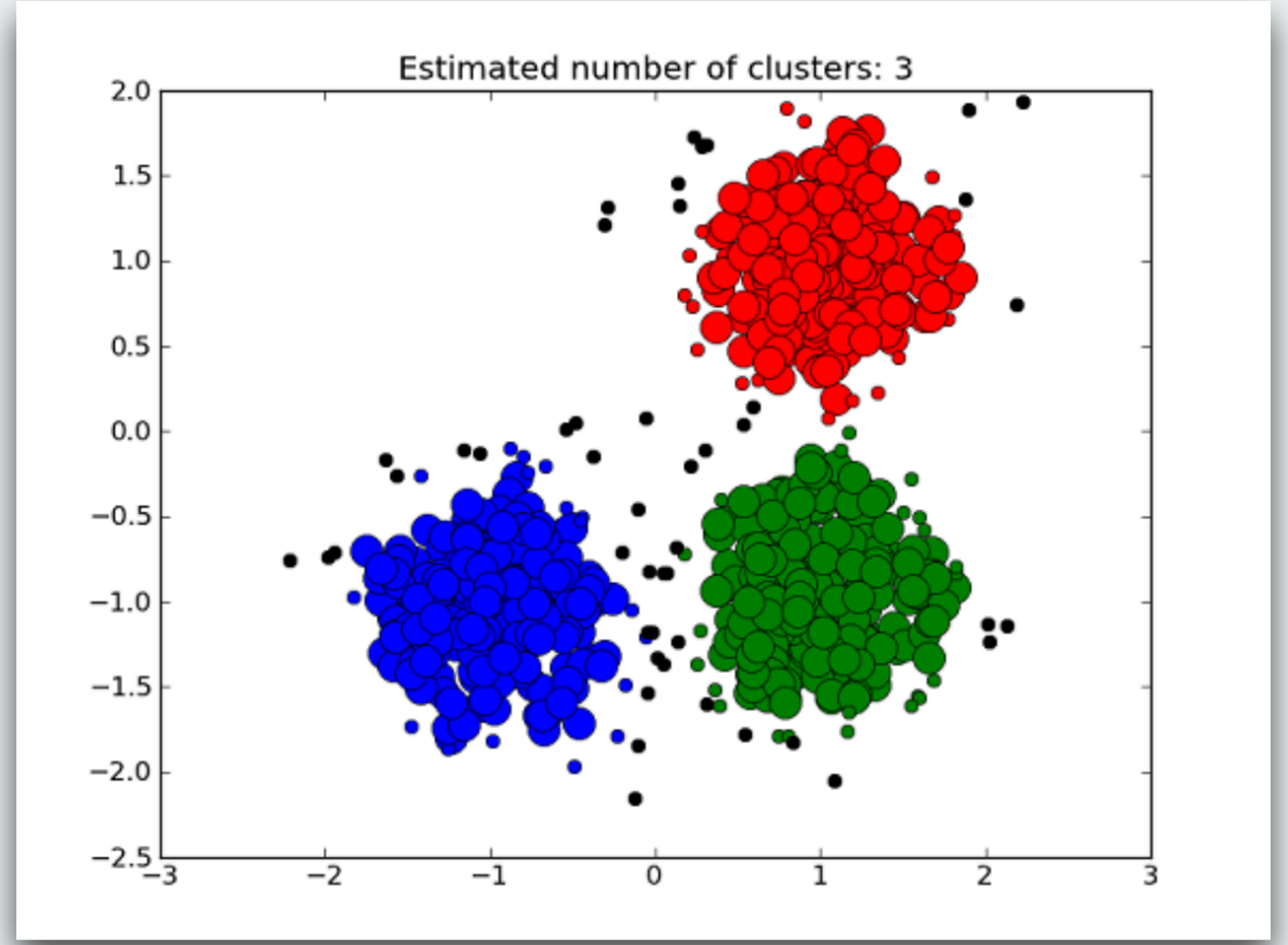


Clustering

Clustering

Gist:

- Data typically has structure
- Clustering simplifies and highlights important structure
- There are MANY different clustering algorithms, because data takes many forms



Clustering

> Different methods for different things

Parametric: K-means, C-means, GMM

- Main hyper parameter: number of clusters
- Found clusters will be convex (circular or ellipsoidal)
- No notion of noise, and all points will belong to a cluster

Non-parametric: DBSCAN, Linkage clustering

- Main hyper parameter: distance threshold
- Finds clusters of any shape
- Distant points are labeled as outliers

Clustering

> Parametric: K-Means (hard partitions)

https://observablehq.com/@brookslybrand/visualizing-k-means-algorithm

Algorithm

1. Randomly initiate K centroids in your data space
2. **Assign** each point to its nearest centroid
3. **Move** each centroid to the mean position of its assigned points
4. Repeat 2 and 3 until no further position updates can be made

Minimizes this cost function: $\text{SSE} = \sum_{i}^{n} \sum_{j}^{k} w_{i,j} (\mathbf{x}_i - \mu_j)^2$

Clustering

> Parametric: C-Means (soft partitions)

If there are *K* clusters, each data point is described by a *K* long membership vector. It is a probability vector estimating the prob. of belonging in each cluster

https://observablehq.com/@liu246542/fuzzy-c-means-vs-k-means

Algorithm

1. Specify K and randomly assign cluster membership for each datapoint
2. **Compute** cluster centroids μ
3. **Update** cluster membership vectors for each data points to lower the cost
4. Repeat steps 2 and 3 for a fixed number of iterations or until updates are tiny

Minimizes this cost function: $\text{SSE} = \sum_{i}^{n} \sum_{j}^{k} w(m)_{i,j} (\mathbf{x}_i - \mu_j)^2$

Clustering

> Parametric: Gaussian mixture model (soft partitions)

https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm

***Again*: With *K* clusters, each data point is described by a *K* long membership vector. It is a probability vector estimating the prob. of belonging in each cluster**

E-M algorithm (oversimplified)

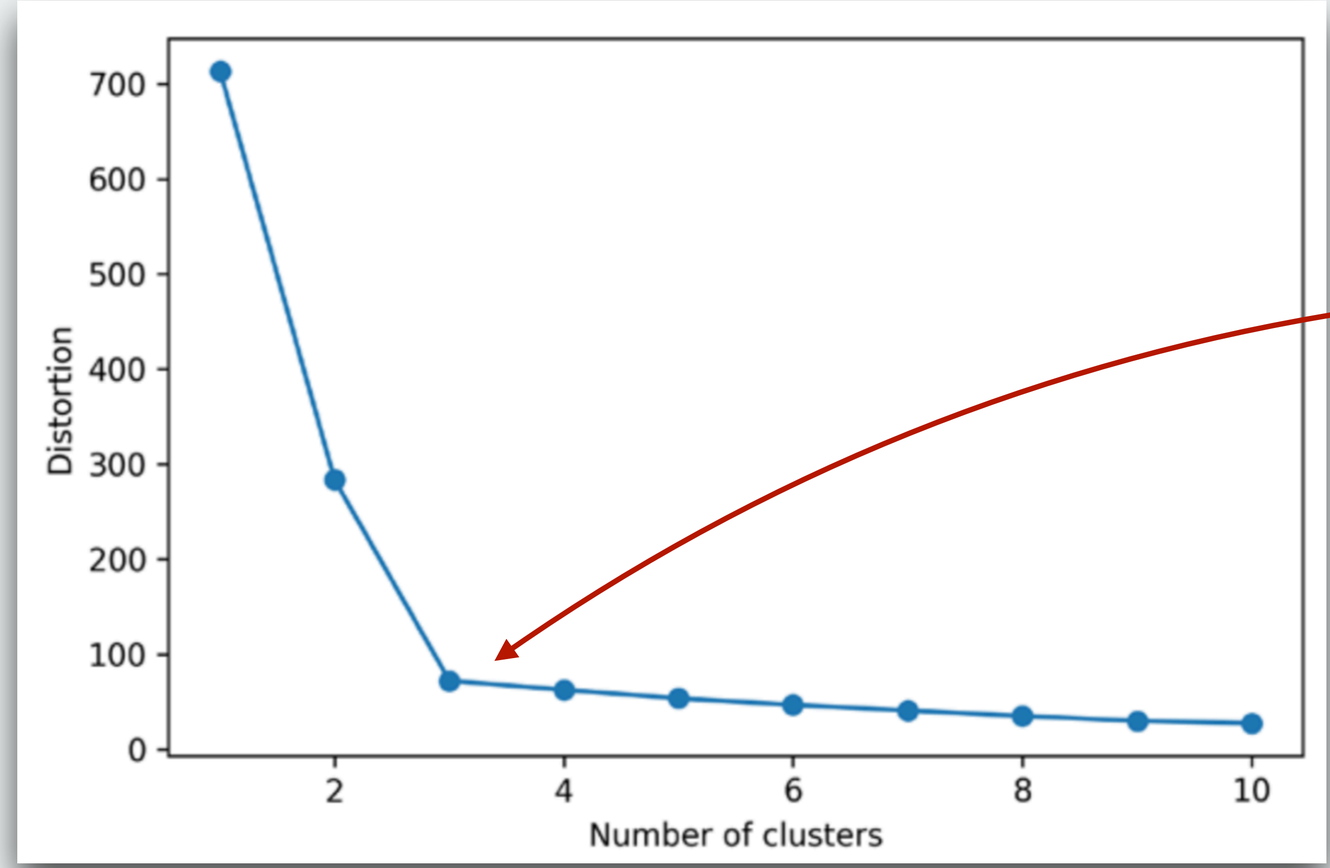
1. Create K Gaussians with random μ and $\mathbf{\Sigma}$ (means and covariances)
2. *E-step*: **Compute** a membership vector for each datapoint
    1. Compute likelihood of datapoint given each Gaussian (K values)
    2. To get membership of point i in cluster j, normalize the likelihood, i.e.: $p_{i,j} = \frac{L_{i,}}{\sum_{j'} L_{i,j'}}$
3. *M-step*: Given new membership vectors, **update** μ and $\mathbf{\Sigma}$ of each Gaussian
4. Repeat steps 2 and 3 for a fixed number of iterations or until updates are tiny

Here's a great reference!

Clustering

> Evaluating the number of clusters with the Elbow method

Idea: You can measure your cost function value (SSE for K-means and C-means and total likelihood for GMM) as a function of number of kernels.



Heuristic: Choose a number of clusters for which adding more clusters doesn't significantly lower the cost

Clustering

> Evaluating the number of clusters with the Silhouette method

Idea: A point should be closer to points in its own cluster then points in the neighboring cluster. Measure that relative distance and plot it

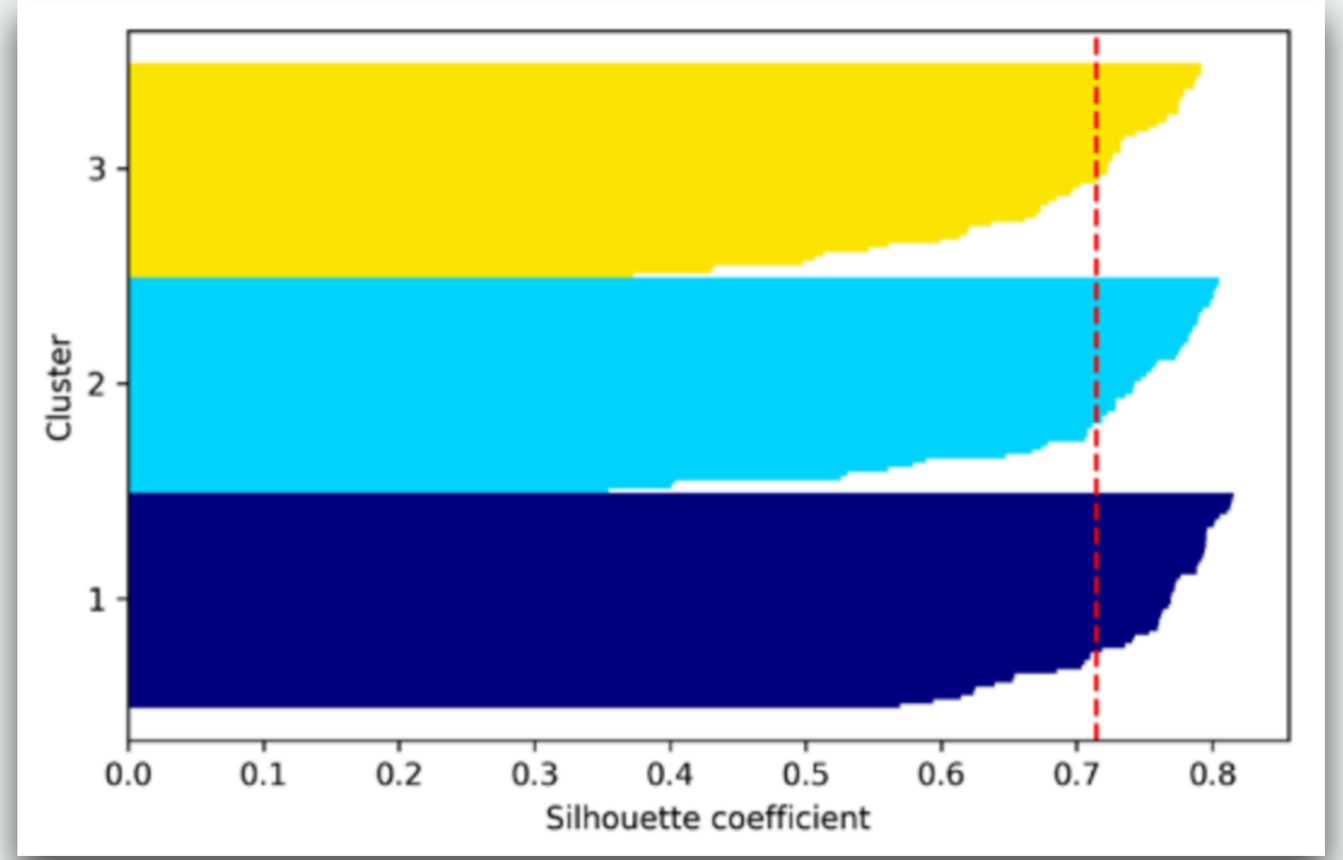
Algorithm

1. For each point i, calculate the own-cluster cohesion a_i as the average distance from points in its cluster
2. Do the same but with average distances to points in neighbor cluster to get b_i
3. Compute its silhouette score $s_i = \dfrac{b_i - a_i}{\max(a_i, b_i)}$
4. Compute for all i and visualize the scored ordered by cluster

Clustering

> Evaluating the number of clusters with the Silhouette method

Idea: A point should be closer to points in its own cluster then points in the neighboring cluster. Measure that relative distance and plot it



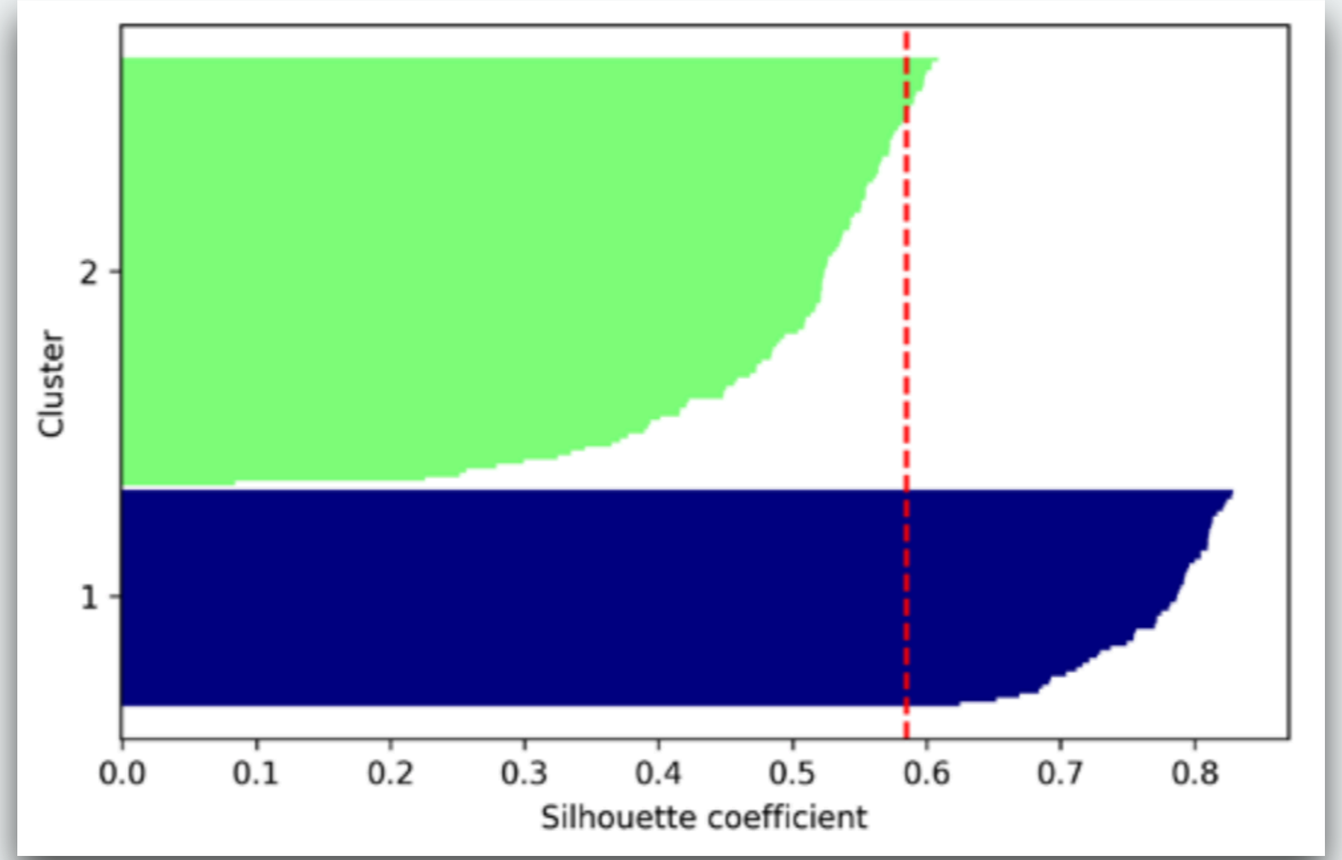
Heuristic: If silhouette scores are **high** and **similar** for most points, it's a good number of clusters

Example

Clustering

> Evaluating the number of clusters with the Silhouette method

Idea: A point should be closer to points in its own cluster then points in the neighboring cluster. Measure that relative distance and plot it



Heuristic: If silhouette scores are **high** and **similar** for most points, it's a good number of clusters

Example

Clustering

> Parametric clustering in a nutshell

1. You have a dataset, $\mathbf{X} = \begin{pmatrix} x_{0,0} & x_{0,1} & \cdots & x_{0,j} & \cdots & x_{0,D} \\ x_{1,0} & x_{1,1} & \cdots & x_{1,j} & \cdots & x_{1,D} \\ \vdots & \vdots & & \vdots & & \vdots \\ x_{i,0} & x_{i,1} & \cdots & x_{i,j} & \cdots & x_{i,D} \\ \vdots & \vdots & & \vdots & & \vdots \\ x_{N,0} & x_{N,1} & \cdots & x_{N,j} & \cdots & x_{N,D} \end{pmatrix}$

2. And you a representation, $\mathbf{Z} = \begin{pmatrix} z_{0,0} & z_{0,1} & \cdots & z_{0,j} & \cdots & z_{0,K} \\ z_{1,0} & z_{1,1} & \cdots & z_{1,j} & \cdots & z_{1,K} \\ \vdots & \vdots & & \vdots & & \vdots \\ z_{i,0} & z_{i,1} & \cdots & z_{i,j} & \cdots & z_{i,K} \\ \vdots & \vdots & & \vdots & & \vdots \\ z_{N,0} & z_{N,1} & \cdots & z_{N,j} & \cdots & z_{N,K} \end{pmatrix}$

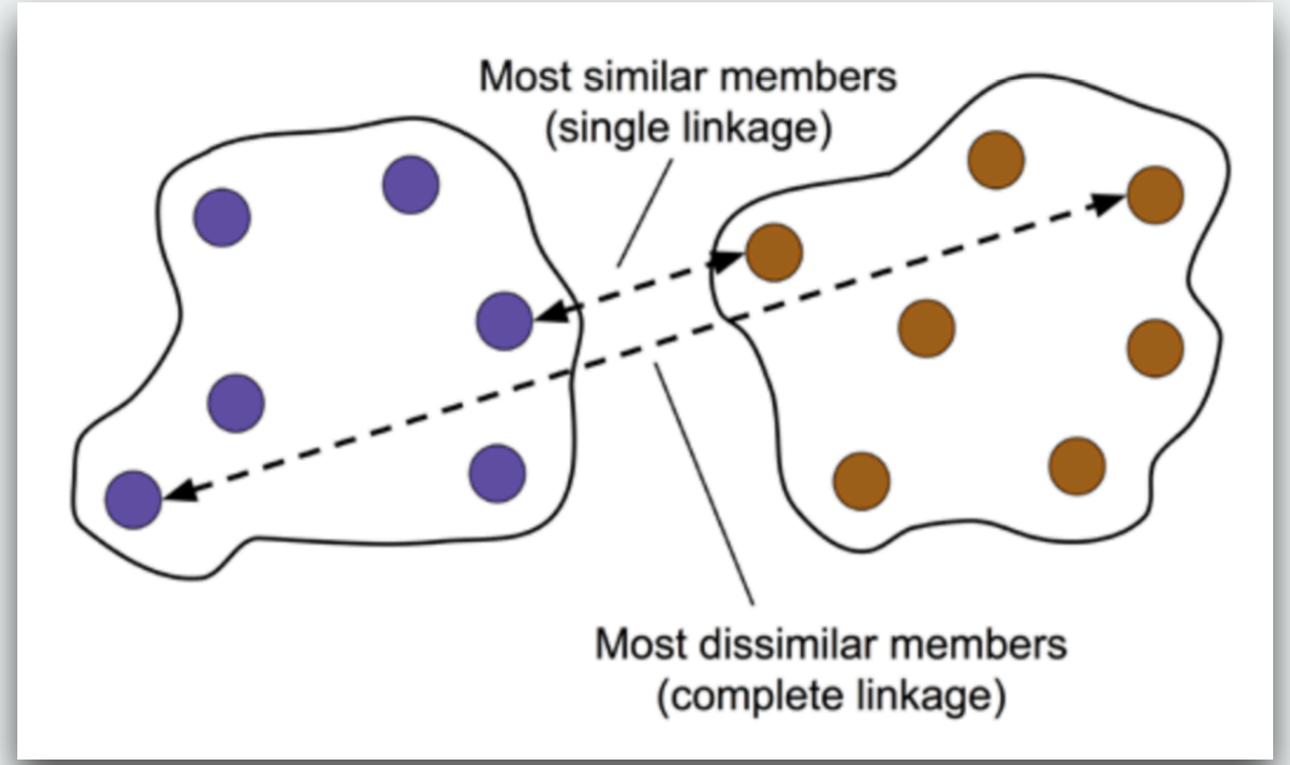
1. Each row in **X/Z** is a datapoint
2. Each vertical vector $\mathbf{z_j}$ in **Z** represents a kernel/centroid indicating to what extend $\mathbf{x_i}$ is belongs to it
3. Not very different from dimensionality reduction! We can even treat **Z** as a non-linear projection of the data and use it for machine learning
4. Key feature: **You have to choose *K*!**

Clustering

> **Non-parametric:** **Linkage clustering** (aka agglomerative/hierarchical clustering)

Algorithm

1. Compute the distance matrix of all samples.
2. Represent each data point as a singleton cluster.
3. Merge the two closest clusters based on the distance between the most dissimilar (distant) members.
4. Update the similarity matrix.
5. Repeat steps 2-4 until one single cluster remains.

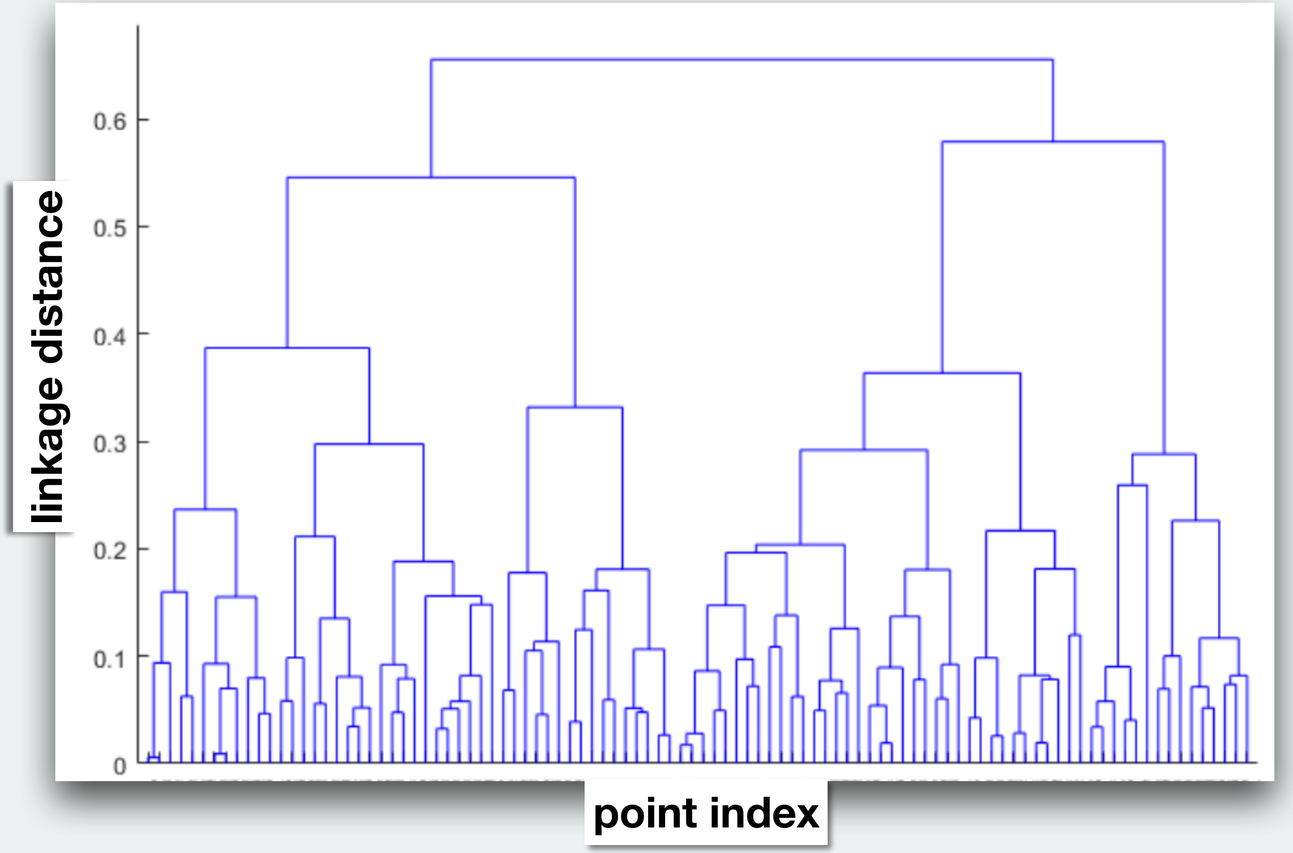


Clustering

> **Non-parametric: Linkage clustering** (aka agglomerative/hierarchical clustering)

Algorithm

1. Compute the distance matrix of all samples.
2. Represent each data point as a singleton cluster.
3. Merge the two closest clusters based on the distance between the most dissimilar (distant) members.
4. Update the similarity matrix.
5. Repeat steps 2-4 until one single cluster remains.

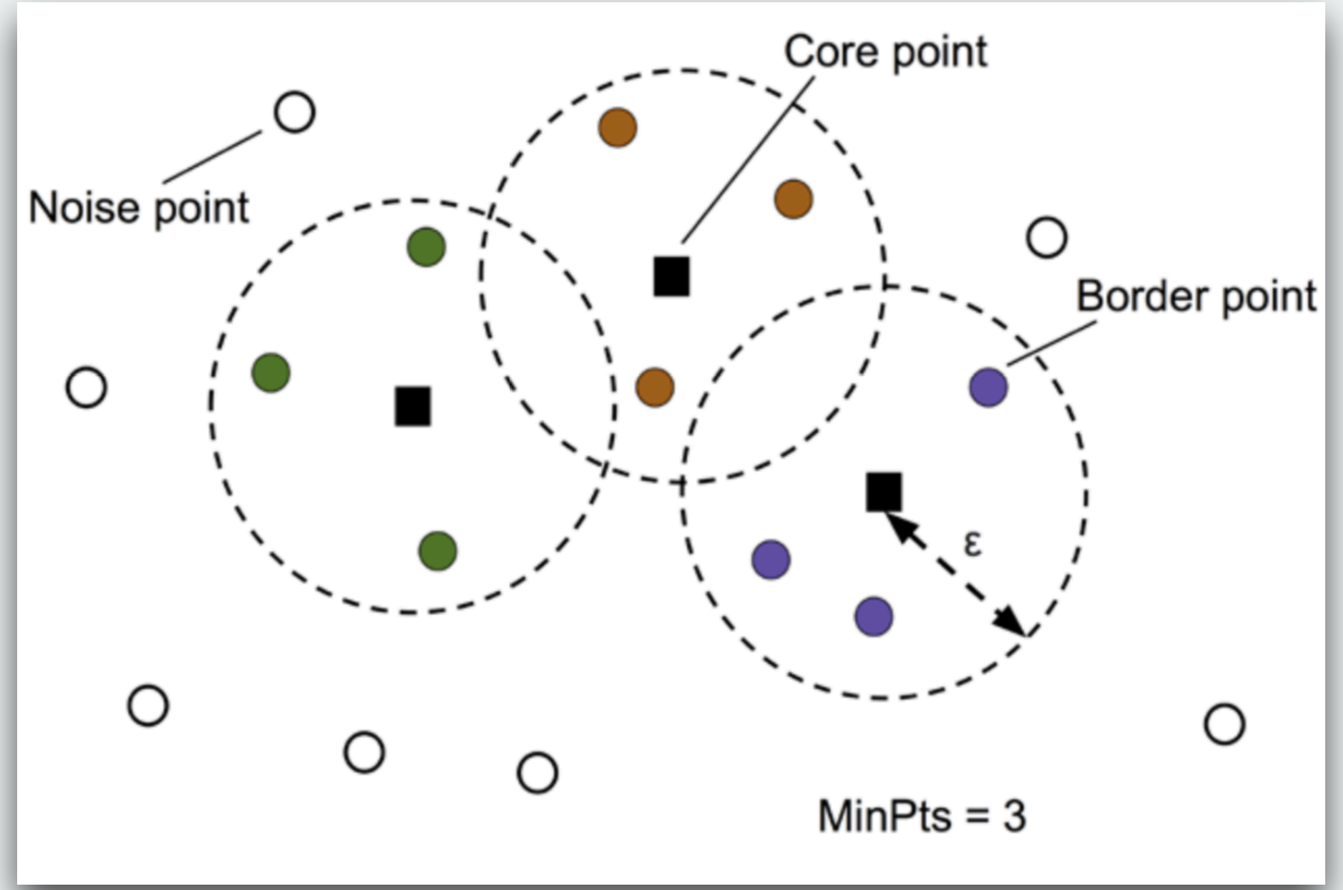


Clustering

> Non-parametric: DBSCAN

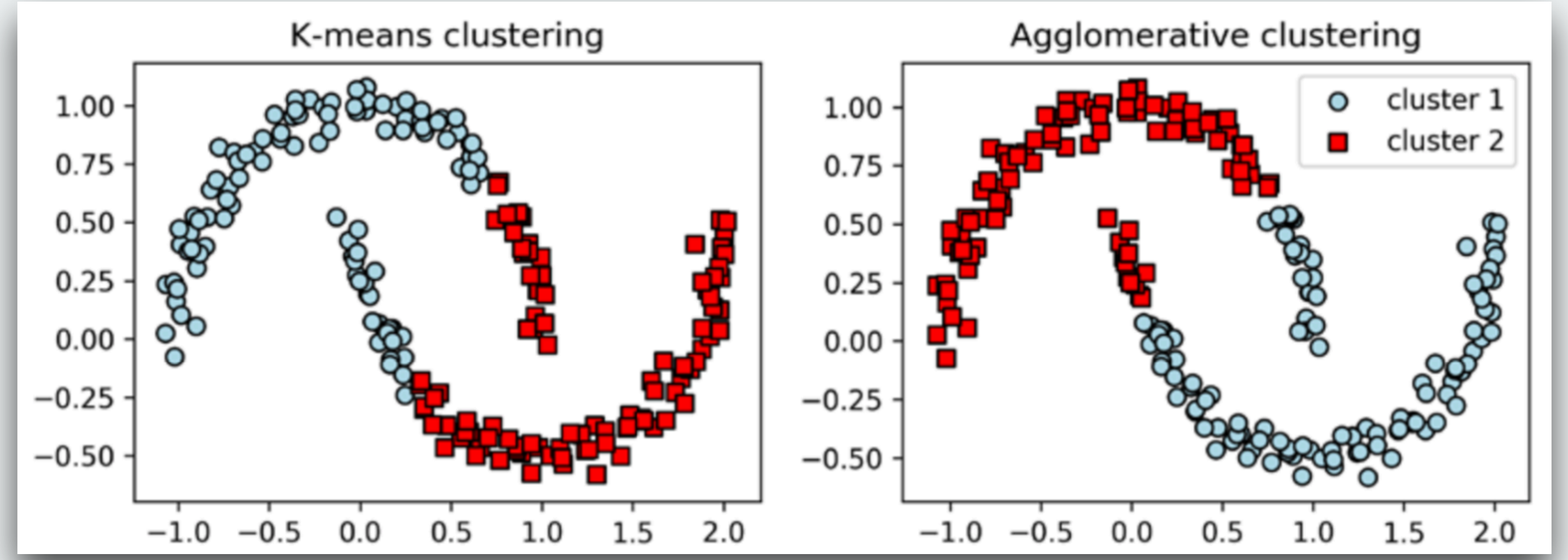
Algorithm

1. Label points as **core points** if they at least MinPts number of neighbors are within a distance of ε. Label points as **border points** if they have more than 1 but less than MinPts neighbor points within ε. The rest is **noise**.
2. Form a separate cluster for each core point or connected group of core points (core points are connected if they are no farther away than ε).
3. Assign each border point to the cluster of its corresponding core point.

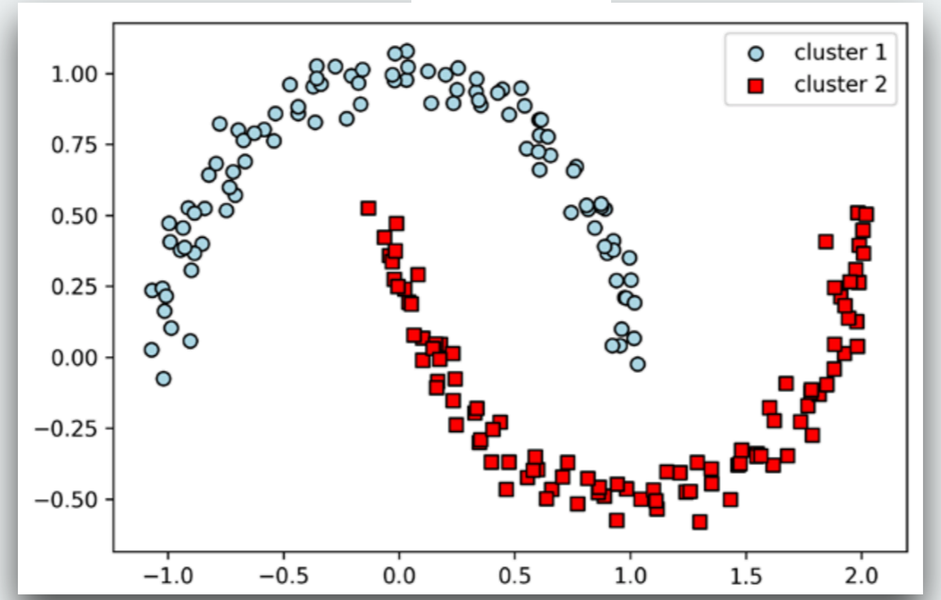


Clustering

> Comparison



DBSCAN



Further topics in unsupervised learning

Factorization methods:

- Kernel PCA (KPCA)
- Factor analysis (FA)
- Non-negative matrix factorization (NMF)
- Independent component analysis (ICA)
- Archetypal analysis (AA)

Clustering

- Community detection (networks)
- HDBSCAN
- OPTICS
- …

Novelty detection

- Outlier detection
- Density estimation
- Change point detection

Neural networks

- Auto-encoders
- Generative adversarial networks
- Boltzmann machines
- Deep belief networks