Machine Learning I

Chapter 11 - Dimensionality Reduction

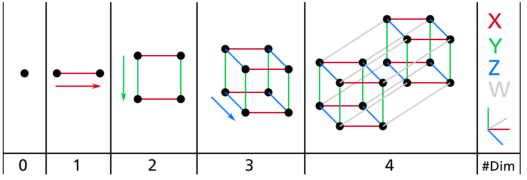
Prof. Dr. Sandra Eisenreich

January 11 2024

Hochschule Landshut

Motivation: The curse of dimensionality

Curse of dimensionality: If you have a cube with length 2-sides in n dimensional feature space, its volume is 4 for $n = 2$, 8 for $n = 3$, 16 for $n = 4$... $\Rightarrow$ volume of a cube grows exponentially. $\Rightarrow$ Any fixed number of instances will spread out and become exponentially more isolated with growing number of features.



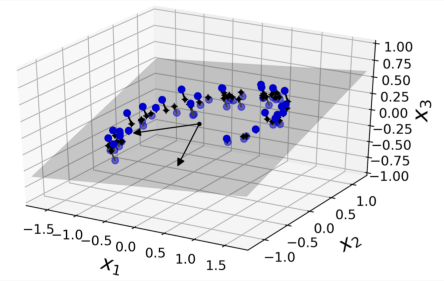
Problems of high-dimensional feature spaces: ?

Solution: reduce dimension of feature space, ideally without losing much information.

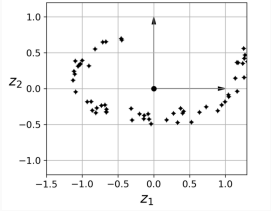
The task, and basic idea

The task: dimensionality reduction, unsupervised learning.

Basic idea: In most real-world problems, training instances are NOT spread out uniformly accross all dimensions: but lie within or close to a much lower-dimensional subspace. (manifold hypothesis).

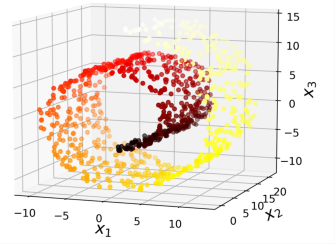


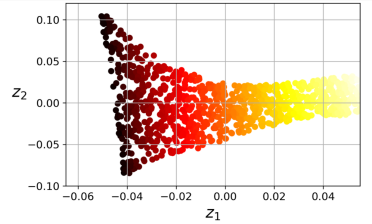
In the above case, one could just project all datapoints onto the 2-dimensional subspace to get:



Manifold learning; When to use

One can also use non-linear maps to project down to lower-dimensional spaces (this is called manifold learning), example:





When to use methods of dimensionality reduction:

- For visualization: project the data down to 2 or 3 dimensions.
- To reduce the dimension of feature space before using a different model.
- First try out linear projections, only if those don't work well use non-linear methods.

Principal Component Analysis (PCA)

PCA - Motivation

Aim:

Basic idea: Project/Compress the data on the hyperspace of dimension d that keeps most of the information/variance in the data.

Question: Projection = linear map. So how can we express the projection?

Answer: ?

Question: What is the best projection?

Simple idea: ?

$\Rightarrow$ minimize the MSE cost function (which is called the reconstuction error in this case): ?

When to use: For dimensionality reduction always first try out PCA. Easy, fast, popular.

PCA

PCA (Principal Components Analysis) is the ML algorithm with:

- Projection function: $h(\mathbf{x}) = W\mathbf{x}$, (the entries of $\mathbf{W}$ are the parameters).
- Loss function = reconstruction error

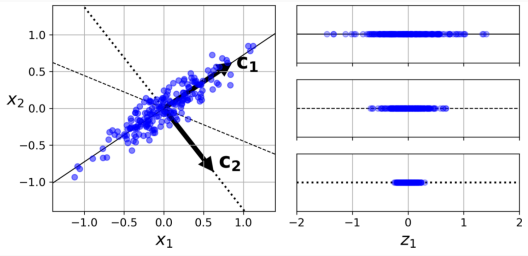
$$\text{MSE}(W) = \frac{1}{m}\sum_{i=1}^{m}\left\|\mathbf{x}^{(i)} - W^T W\mathbf{x}^{(i)}\right\|$$

How to optimize this loss function: In this case, we do not need to use Gradient Descent or other optimization algorithms, since the solution can be derived with Linear Algebra.

Principal Components

One can divide the entire feature space into so-called principal components (PC):

- The 1st PC is the direction (vector) which accounts for most of the variance of the data (here: c_1).
- The 2nd PC is the axis orthogonal to the first PC which accounts for the largest amount of the remaining variance (here: c_2), etc.



Math shows that the best hyperplane to project onto is given by the span of the first d principal components.

Computing the PCs of data 1

Question: Variance in higher dimensions is captured in the...?

Answer: covariance matrix of the data.

Definition

Seien $X_1, \ldots, X_d$ Zufallsvariablen. Dann definiert man ihre Kovarianzmatrix als die $d \times d$-Matrix

$$Cov(X_1, \ldots, X_d) = \begin{pmatrix} Var(X_1) & Cov(X_1, X_2) & \ldots & Cov(X_1, X_d) \\ Cov(X_2, X_1) & Var(X_2) & \ldots & Cov(X_2, X_d) \\ \vdots & \vdots & \ddots & \vdots \\ Cov(X_d, X_1) & Cov(X_d, X_2) & \ldots & Var(X_d) \end{pmatrix}$$

Es gilt:

- Die Kovarianzmatrix ist symmetrisch, da $Cov(X_i, X_j) = Cov(X_j, X_i)$.
- Die Kovarianzmatrix ist diagonal $\Leftrightarrow$ Die Zufallsvariablen sind unabhängig.

Computing the PCs of data 2

The covariance matrix of n-dimensional data ("Stichprobe") $(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)})$ is given by

$$\Sigma = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{x}^{(i)} - \overline{\mathbf{x}}) \cdot (\mathbf{x}^{(i)} - \overline{\mathbf{x}})^T, \text{ where } \overline{\mathbf{x}} \text{ is the mean}$$

One can show:

The principal components of data $(\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)})$ are given by the eigenvectors of the covariance matrix Σ. The amount of variance captured by an eigenvector corresponds to the size of its eigenvalue, i.e. if you sort the eigenvectors by the size of their eigenvalues (in descending order), the first eigenvector is the first PC, etc.

How to PCA and Complexity

With this, one can show (e.g. see Section 10.1.2 of Murphy, Probabilistic Machine Learning):

> The matrix W minimizing the reconstruction loss is the transpose of the one which contains the first d PCs as columns.

So what you have to do to compute the PCA of data is: ?

> The computational complexity of computing the PCA is $O(m \cdot n^2) + O(n^3)$.

PCA with Scikit-Learn

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

(PCA is unsupervised $\rightarrow$ no train-test-split). Here, `n_components` is the number of dimensions you want to project down onto. After fitting the PCA, its `components_` attribute holds the transpose of W. `.explained_variance_ratio_` gives access to the explained variance ratio = the proportion of the dataset's variance that lies along each principal component.

How to choose the number of dimensions

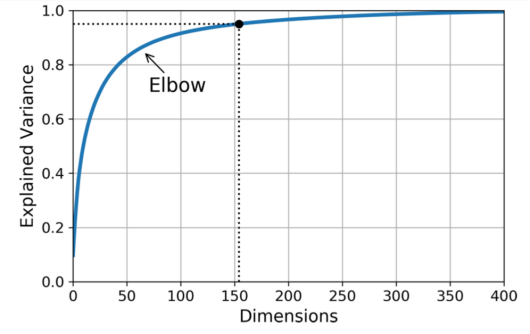
For Visualization Purposes: 2 or 3

Otherwise you want the smallest number that still keeps most information of the datapoints (i.e. most of the variance, e.g. 95%). To do that, set `n_components` to a float between 0.0 and 1.0 indicating the ratio of variance you wish to preserve:

```
pca = PCA (n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

Plot of Explained Variance vs. Dimensions

Alternatively: plot the explained variance as a function of the number of dimensions. This is called the Scree Plot:



With Scikit-Learn after training a PCA model:

```
PC_values = np.arange(pca.n_components_) + 1
plt.plot(PC_values, pca.explained_variance_ratio_)
```

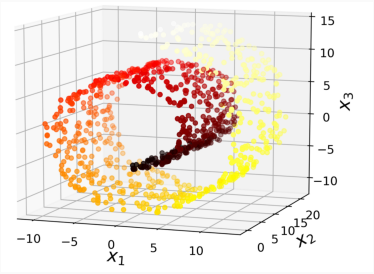
Kernel PCA

When to use and basic idea

When to use: When the space the instances lie in is not a linear subspace, but a so-called manifold.

A *d*-dimensional manifold in $\mathbb{R}^n$ is a subset that locally can be projected onto a *d*-dimensional hyperplane.

Example: here you would rather "unroll" the data rather than squashing them down.



Basic idea: Recall the kernel trick: replacing terms "$\mathbf{x}^{(i)T}\mathbf{x}^{(j)}$" with "$K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$" for a kernel K makes things non-linear. We can make linear PCA non-linear by doing the same thing (but that is mathematically not trivial). The resulting algorithm is called Kernel PCA.

Kernel PCA with Scikit Learn

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

How to find the best kernel and hyperparameter values? PCA is unsupervised, so there is no performance measure. One method: select kernel and hyperparameters that yield the lowest reconstruction error, which you get as follows:

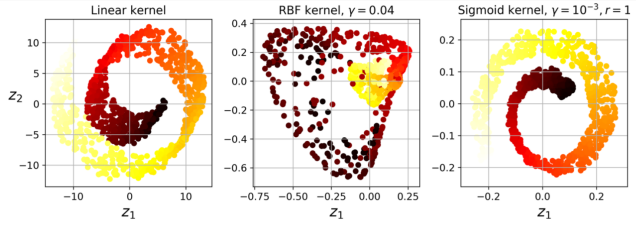
```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                    fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)

from sklearn.metrics import mean_squared_error
mean_squared_error(X, X_preimage)
```

Locally Linear Embedding

Motivation

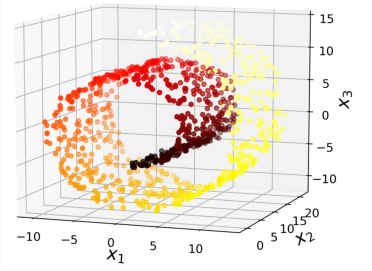
Unrolling the example earlier with Kernel PCA for different kernels yields:

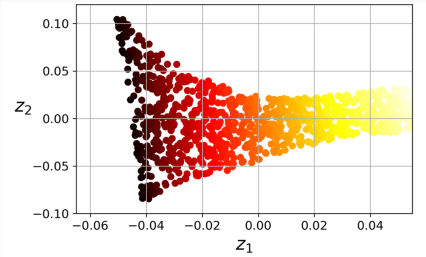


This is not the unrolling we wanted! $\rightarrow$ a different non-linear dimensionality reduction method, which actually "unrolls" the roll: : Locally Linear Embedding (LLE).

Locally Linear Embeddings: Overview

When to use: For unrolling manifolds:





Disadvantage: becomes slow for large datasets! So for many data use something else.

The Locally Linear Embedding (LLE) algorithm for dimensionality reduction works as follows: For each instance, find the closest neighbors, and describe how the vectors are related geometrically (e.g. by writing the instance as a linear combination of the others). Try to reproduce this relation of neighbors in low-dimensional space.

LLE Algorithm: Step 1

Given: training data.
Aim: express each instance as a linear combination of its closest neighbors.
Method: for each training instance $\mathbf{x}^{(i)}$, the algorithm identifies its k nearest neighbours, then tries to write $\mathbf{x}^{(i)}$ as a linear combination of these neighbours: find weights $w_{i,j}$ such that $\sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)}$ approximates $\mathbf{x}^{(i)}$ as well as possible (where $w_{i,j} = 0$ if $\mathbf{x}^{(j)}$ is not a nearest neighbour of $\mathbf{x}^{(i)}$), i.e. the aim is to

$$\text{minimize}_{w_{i,j}} \|\mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)}\|_2^2.$$

This is the first optimization step. Normalize the $w_{i,j}$ such that $\sum_{j=1}^{m} w_{i,j} = 1$ for all $i = 1, \ldots, m$. Write these weights into a matrix $\mathbf{W} = (w_{i,j})$.

LLE Algorithm: Step 2

Given: $w_{i,j}$ that express each instance (almost) as a linear combination of their closest neighbors.
Aim: Find a projection to lower-dimensional space that "preserves" these linear combination. Let $\mathbf{z}^{(i)}$ denote the image of $\mathbf{x}^{(i)}$ in this d-dimensional space. Then like we want the same relationship as above for $\mathbf{x}^{(i)}$ to also hold for $\mathbf{z}^{(i)}$, i.e.

$$\text{minimize}_{\mathbf{z}^{(j)}} \|\mathbf{z}^{(i)} - \sum_{j=1}^{m} \hat{w}_{i,j}\mathbf{z}^{(j)}\|_2^2,$$

(only this time the weights $\hat{w}_{i,j}$ are fixed and we optimize for the coordinates of the $\mathbf{z}^{(i)}$.)

Complexity and LLE with Scikit Learn

Without proof (one would need a lot of mathematical theory to derive that):

> The complexity of LLE is $O(n \log km \log m) + O(dmk^3) + O(dm^2)$, where n = input dimension, d = output dimension, m = number of samples, k = number of nearest neighbors.

Use ScikitLearn's LocallyLinearEmbedding class

```
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

Chapter 10 Summary

- Motivation: curse of dimensionality: higher-dimensional feature space means the instances are farther apart and the model can overfit more easily. Also, many features make training and inference slow $\rightarrow$ it's often good to reduce the dimension of feature space.
- task: dimensionality reduction. Unsupervised learning.
- PCA = Principal Components Analysis: Finding a projection matrix $W \in \mathbb{R}^{d \times n}$ onto a linear d dimenional subspace such that the reconstruction error (= the difference between a vector and the decompressed compressed vector) is minimal:

$$\text{MSE}(W) = \frac{1}{m} \sum_{i=1}^{m} \left\| \mathbf{x}^{(i)} - W^T W \mathbf{x}^{(i)} \right\|$$

  = identify the directions where the data have most variance. Project the data down onto the span of the first d such vectors.
  Then project all instances $\mathbf{x}$ down to the compressed vector $W\mathbf{x}$.
- Kernel PCA is PCA with the kernel trick to make linear PCA non-linear.
- Computational Complexity of PCA/Kernel-PCA: $O(m \cdot n^2) + O(n^3)$

- If Kernel PCA doesn't work, (e.g. for more complex manifolds), one can use methods of manifold learning like Locally Linear Embedding (LLE):
  - Step One: Given: training data.
    Aim: express each instance as a linear combination of its closest neighbors, i.e.

$$\text{minimize}_{w_{i,j}} \|\mathbf{x}^{(i)} - \sum_{j=1}^{m} w_{i,j}\mathbf{x}^{(j)}\|_2^2.$$

  - Step 2: Given: $w_{i,j}$ from Step 1.
    Aim: Find a projection to lower-dimensional space $\mathbf{z}^{(i)}$ of all training instances $\mathbf{x}^{(i)}$ that "preserves" these linear combination, i.e.

$$\text{minimize}_{\mathbf{z}^{(i)}} \|\mathbf{z}^{(i)} - \sum_{j=1}^{m} \hat{w}_{i,j}\mathbf{z}^{(j)}\|_2^2.$$

- Computational Complexity of LLE: $O(n \log km \log m) + O(dmk^3) + O(dm^2)$, where $n =$ input dimension, $d =$ output dimension, $m =$ number of samples, $k =$ number of nearest neighbors.