Data Science and Artificial Intelligence for Undergraduates Volume 3: Unsupervised Learning

Jens Flemming

Feb 04, 2024

CONTENTS

Data Science and Artificial Intelligence for Undergraduates

This book covers a wide range of topics in data science and artificial intelligence. It's an attempt to provide self-contained learning material for first-year students in data science related courses. Most, not all, of the material is tought in the undergradute course on data science[1] at Zwickau University of Applied Sciences[2].

Starting teaching data science in 2019 the author[3] faced the problem that there seems to be no text book covering math, computer science, statistical data science, artificial intelligence and related topics in a well structured, accessible, thorough way. Basic Python[4] programming should be covered as well as state of the art deep reinforcement learning for controlling autonomous robots. All this with hands-on experience for students, interesting real-world data sets, and sufficiently rich theoretical background.

Classical paper books or PDF ebooks do not suit the needs for this project. Working with data requires lots of source code, interactive visualizations, data listings, and easy to follow pointers to online resources. Jupyter Book[5] is an awesome software tool for publishing book-like interactive content. For the author writing this book is also a journey of discovery to the possible future of publishing. Having authored two paper books the author knows the tight limits of paper books and publishing companies. The greater his enthusiasm is for the freedom in writing and publishing provided by Jupyter Book and its community, The Executable Books Project[6].

The author expresses its gratitude towards all the more or less anonymous people developing the wonderful open source tools used in this book and for writing the book. There are too many tools to list them here. The author also thanks his students and colleagues at Zwickau University, especially Hendrik Weiß, who constantly find typos and make suggestions for improving the book.

Jens Flemming[7], Zwickau, February 2024

[1] https://datascience.fh-zwickau.de
[2] https://www.fh-zwickau.de
[3] https://www.fh-zwickau.de/~jef19jdw
[4] https://www.python.org
[5] https://jupyterbook.org
[6] https://executablebooks.org
[7] https://www.fh-zwickau.de/~jef19jdw

Part I

Unsupervised Learning

CHAPTER

ONE

TYPICAL TASKS

Unsupervised machine learning algorithms extract information from data sets without the need for examples of the information to extract. The algorithm determines what to extract and humans have to interpret the extracted information to obtain insights into the data. Thus, it's very easy to collect data sets suitable for unsupervised learning, but interpretation of results may be difficult.

Unsupervised learning is an extremely wide field. Here we focus on *clustering* and also consider *dimensionality reduction*. Further, we'll ahve a first galnce at *generative models*. But there exist other subfields, too, *anomaly detection* and *association analysis* for instance. Most techniques can be applied to different tasks.

Like for supervised learning we denote the data space by X and the items of the data set under consideration by $x_1, \ldots, x_n$. Almost always we will have $X = \mathbb{R}^m$. Again we need a training data set for fitting a model. Validation and test sets are used for hyperparameter optimization and model evaluation as before.

1.1 Clustering

Clustering aims at finding subsets of similar items in large data sets.

Details depend an the application in mind.

- Do we want to find *hard clusters* (either a sample belongs to a cluster or not) or *soft clusters* (score/probability for each combination of samples and clusters)?
- Shall all samples belong to some cluster or do we allow for *outliers*?
- May clusters overlap (some samples belong to more than one cluster)?

There exist many different approaches for clustering algorithms. Main classes are:

- *Centroid-based algorithms* represent each cluster by one point (midpoint or centroid). Which samples belong to which clusters is determined by some rule involving those midpoints.
- *Density-based algorithms* look at the distances between samples and define clusters to be subsets of closely spaced samples.
- *Distribution-based algorithms* represent each cluster be a probability distribution. A sample belongs to the cluster for which the sample's probability is highest.
- *Hierarchical algorithms* generate a sequence of clusterings. Either starting with as many clusters as there are samples (and then coarsening the clustering) or starting with one cluster (and then refining).

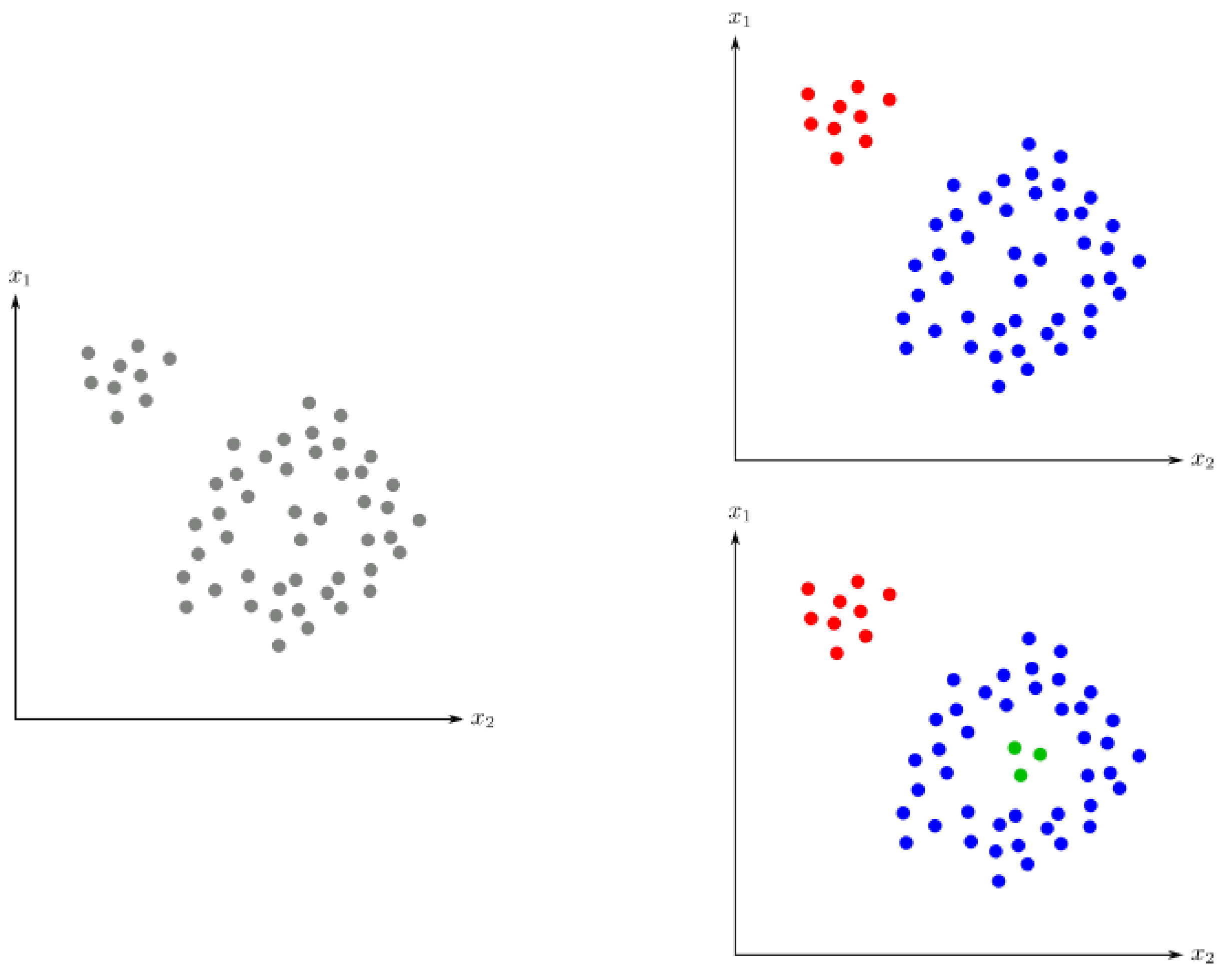


Fig. 1.1: How to define the term *cluster* is not as straight-forward as it seems.

1.2 Dimensionality Reduction

Dimensionality reduction tries to reduce the number of features without loosing too much information. We already know principal component analysis as a linear dimensionality reduction technique. Here, 'linear' means that the mapping from the high dimensional data space to the lower dimensional space is linear (a matrix).

Nonlinear dimensionality reduction techniques are more powerful, but also much more computationally expensive.

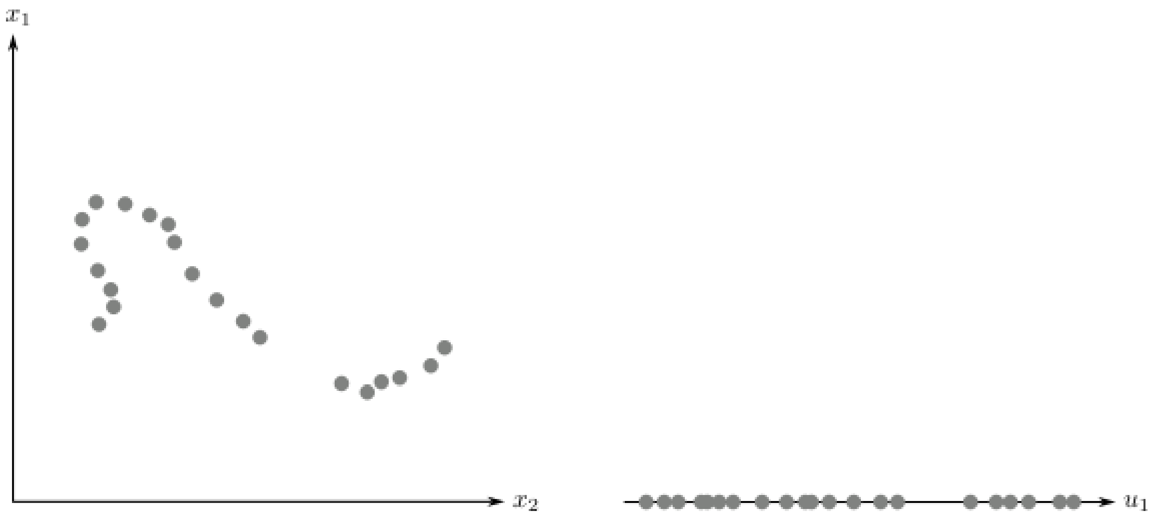


Fig. 1.2: Often data is not scattered over the whole space but lives close to a lower dimensional nonlinear manifold.

1.3 Generative Models

Unsupervised learning algorithms may not only learn to distinguish between similar and dissimilar samples. Some algorithms yield so called *generative models*. Generative models contain all information necessary to automatically create new samples similar to samples in the training data set.

Generative models can be used for generating natural looking artifical images[8] or for generating art[9], for instance.

[8] https://en.wikipedia.org/wiki/StyleGAN
[9] https://en.wikipedia.org/wiki/Edmond_de_Belamy

CHAPTER

TWO

QUALITY MEASURES FOR CLUSTERING

Evaluating the quality of a clustering is not as simple as it looks at first glance. Almost always we do not have a ground truth at hand (external evaluation). Instead we can only look at size and shape of clusters themselves (internal evaluation). There exist lots of internal evaluation metrics. Below we only consider two examples. There is no best metric, because cluster evaluation heavily depends on the intended application. The only reliable evaluation metric is human inspection. But human inspection is restricted to visualizations, which are not available for high dimensional data sets. Dimensionality reduction techniques may help.

In the following we represent clusters as subsets of our data set. In other words, a cluster C is a subset of $\{x_1, \ldots, x_n\}$.

2.1 Silhouette Score

The silhouette score relates intra-cluster distances to inter-cluster distances. It is defined for each sample of a data set. The silhouette score of a whole data set then is the mean score of all samples.

Given some distance measure $d: X \times X \to [0, \infty)$ (usually Euclidean distance) the intra-cluster distance of a sample x and the cluster C the sample belongs to is the mean distance of the sample to all other samples in the cluster:

$$\text{intra}(x, C) := \frac{1}{|C| - 1} \sum_{\tilde{x} \in C} d(x, \tilde{x}).$$

The inter-cluster distance of a sample x to a cluster $\tilde{C}$ the sample does not belong to is the mean distance of the sample and all samples in the cluster under consideration:

$$\text{inter}(x, \tilde{C}) := \frac{1}{|\tilde{C}|} \sum_{\tilde{x} \in \tilde{C}} d(x, \tilde{x}).$$

Now the silhouette score of a sample x is the ratio of the intra-cluster distance and the smallest inter-cluster-distance:

$$a := \text{intra}(x, C), \quad b := \min_{\tilde{C} \neq C} \text{inter}(x, \tilde{C}), \quad \text{silhouette}(x) := \frac{b - a}{\max\{a, b\}}.$$

If x is the only element in C, then this formula does not work and one sets the silhouette score to zero.

Silhouette score always lie in $[-1, 1]$. It is the higher the lower the intra-cluster distance is and the higher the inter-cluster distance is. Thus, high silhouette score for a sample indicates that it belongs to a cluster well separated from all other clusters. Score close to 0 indicates that the sample belongs to overlapping clusters. A score close to -1 indicates a missclustering (sample is closer to other clusters than to its own cluster).

Silhouette score of a data set represents the average clustering quality. Many missclustered samples result in negative silhouette score and so on. Note, that a silhouette score close to zero may indicate that half the samples have been missclustered as well as that there is no clustering (all clusters heavily overlap).

Important: Silhouette score depends on the chosen distance and, thus, on scaling of each feature. If some feature has much higher numerical values than other features, then that feature will dominate the distances.

Another noteworthy point is that silhouette scores are more reliable if clusters are convex. Else inter-cluster distances might be smaller than intra-cluster distances although clusters were correctly identified.

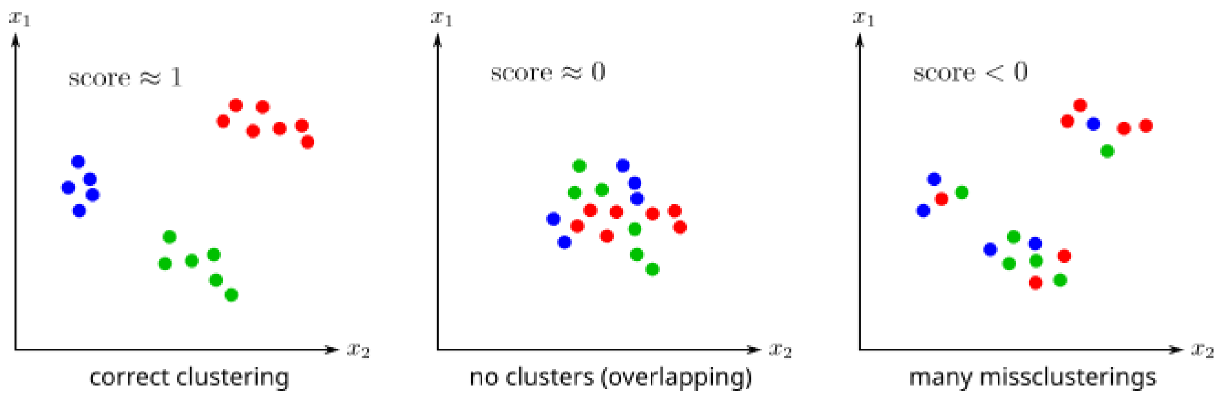


Fig. 2.1: Silhouette scores for different clustering results.

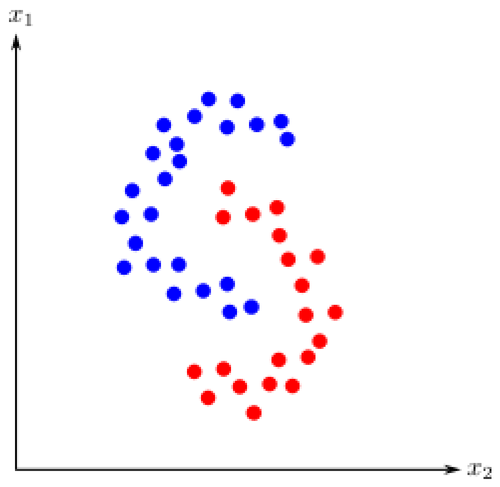


Fig. 2.2: For non-convex clusters silhouette score may indicate bad clustering results although clusters have been identified correctly.

2.2 Davies-Bouldin Index

The Davies-Bouldin index relates cluster diameters to distances between clusters. It is defined for each pair of clusters. The Davies-Bouldin index of a single cluster is the worst Davies-Bouldin index of each pair containing the cluster under consideration. The Davies-Bouldin index of a whole clustering is the mean Davies-Bouldin index of all clusters.

To define the Davies-Bouldin index we need the *centroid* of a cluster C. It's the coordinatewise arithmetic mean of all samples in the cluster:

$$\text{cent}(C) := \frac{1}{|C|}\sum_{x\in C} x.$$

The cluster radius can be defined as the mean distance of samples to the cluster's centroid:

$$r(C) := \frac{1}{|C|},\sum_{x\in C} d(x,\text{cent}(C))$$

with some distance measure d. Usually d is the Euclidean distance, because the notion of centroid is based on considerations involving Euclidean distances. For other distance measures introducing a sensible notion of centroids is difficult. Given two clusters C_1 and C_2 we may define the cluster distance as the distance of their centroids:

$$\text{dist}(C_1,C_2) := d(\text{cent}(C_1),\text{cent}(C_2)).$$

The Davies-Bouldin index of two clusters C_1 and C_2 is

$$\text{DB}(C_1,C_2) := \frac{r(C_1)+r(C_2)}{\text{dist}(C_1,C_2)}.$$

It takes values in $[0,\infty)$ and is the closer to zero the smaller the clusters are and the higher the distance between clusters is.

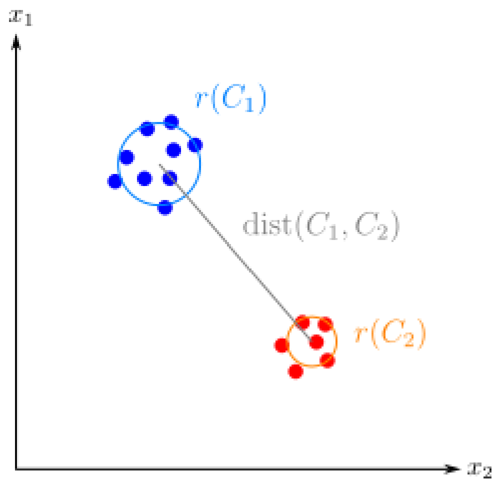


Fig. 2.3: Davies-Bouldin index relates distance between clusters to cluster diameters.

A Davies-Bouldin index above 1 indicates overlapping clusters (at least if the clusters' shapes are close to spheres). If clusters are not sphere shaped the Davies-Boulding index does not yield useful information.

Note that the Davies-Bouldin index of two clusters is symmetric, that is, does not depend on the ordering of the clusters. If there are more than two clusters, the Davies-Bouldin index of each cluster is

$$\text{DB}(C) := \max_{\tilde{C}\neq C}\text{DB}(C,\tilde{C})$$

and the Davies-Bouldin index of the whole clustering is mean Davies-Bouldin index of all clusters.

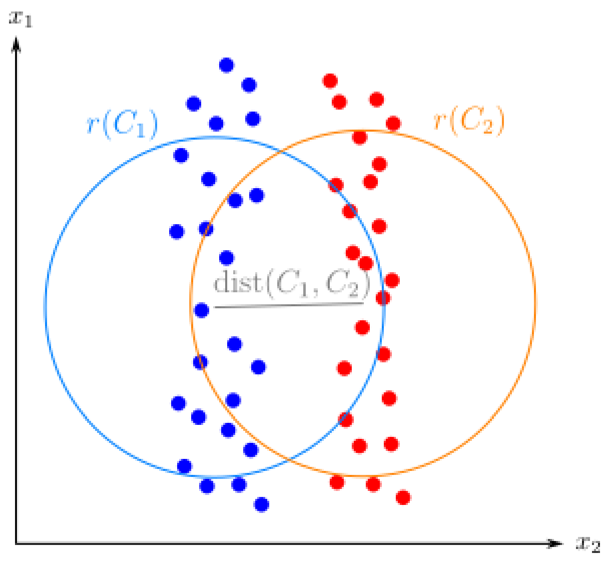


Fig. 2.4: If clusters are not sphere shaped Davies-Bouldin index may indicate bad clustering although clustering is correct.

CHAPTER

THREE

CENTROID-BASED CLUSTERING (K-MEANS)

Centroid-based clustering algorithms represent clusters by single points in the data space (center points, mostly the clusters' centroids). Corresponding clusters then are given by some distance condition. All points closer to some center point than to all others belong to one cluster, for instance. Consequently, centroid-based methods yield sphere-shaped clusters with concrete shape depending on the chosen distance measure.

The major centroid-based clustering method is k-means. Others, not discussed here, are k-medians[10] and k-medoids[11].

Related projects:

- *MNIST Character Recognition* (page 117)
  - *Semisupervised Classification* (page 117)
- *Supermarket Customers* (page 127)

3.1 k-Means Idea

Clusters obtained from k-means method are determined by center points. Given k points $u_1, \ldots, u_k$ in $\mathbb{R}^m$ (the centers) corresponding clusters are defined by

$$C(u_l) := \{x \in \{x_1, \ldots, x_n\} : d(x, u_l) \leq d(x, u_\lambda) \text{ for all } \lambda \neq l\}.$$

A cluster contains all samples which are closer to the cluster's center than to other clusters' centers. In case of equality a sample formally belongs to two or more clusters. In practice, a cluster is chosen by chance to obtain mutually disjoint clusters.

A prescribed clustering (collection of subsets of our data set) may or may not have a set of center points, that is, a set such that for each point in the set all points in the corresponding cluster are closer to that point than to any other cluster's center (simple example: one cluster belongs to the convex hull of another cluster).

Although k-means is closely related to centroids (see below), a cluster's centroid not necessarily is a center point.

The k-means method aims at minimizing the average distance of samples to the closest cluster center. The number k of clusters has to be provided in advance. The outcome are the cluster centers.

In formulas, we want to minimize the function

$$(u_1, \ldots, u_k) \mapsto \frac{1}{n} \sum_{\kappa=1}^{k} \sum_{x \in C(u_l)} d(x, u_l)$$

with respect to all possible cluster centers $\{u_1, \ldots, u_k\}$. The distance measure d almost always is the squared Euclidean distance.

Finding optimal cluster centers is closely related to finding optimal locations for warehouses, hospitals, fire stations, and so on (see Voronoi diagram[12]).

[10] https://en.wikipedia.org/wiki/K-medians_clustering
[11] https://en.wikipedia.org/wiki/K-medoids
[12] https://en.wikipedia.org/wiki/Voronoi_diagram

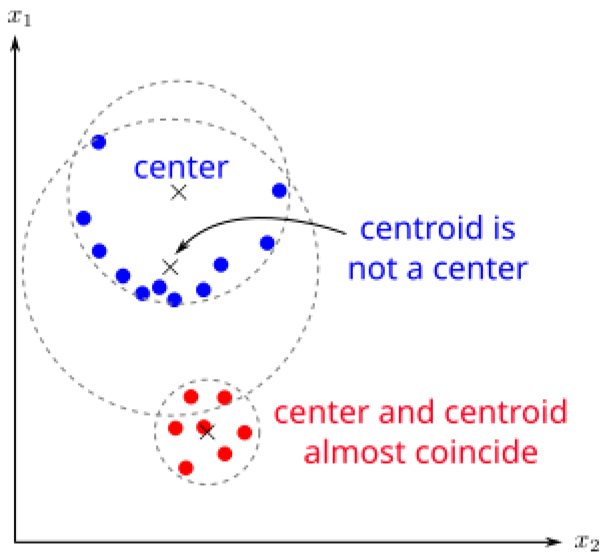


Fig. 3.1: Centroids not necessarily are center points.

3.2 Naive k-Means Algorithm

Naive k-means algorithm (also known as Lloyd's algorithm) starts with a set of initial center points $u_1, \ldots, u_k$ and repeats the following two steps:

- Build clusters $C(u_1), \ldots, C(u_k)$ for current centers $u_1, \ldots, u_k$.
- Update $u_1, \ldots, u_k$ to be the centroids of the current clusters.

Iteration is stopped if clusters do not change anymore. One can show that this method always converges (that is, stops) and that the resulting center points are a local (not a global!) minimizer of the above objective function. From the stopping criterion we immediately see, that centers computed by naive k-means always are centroids of their clusters.

The described algorithm takes two parameters:

- the number k of clusters to find,
- initial center points $u_1, \ldots, u_k$ defining the initial clustering.

There exist several methods to choose these parameters. In some cases k can be obtained from domain knowledge. Else k has to be obtained from typical hyperparameter optimization techniques. Center points may be initialized randomly or by some specialized initialization routines (see below).

3.3 Implementation from Scratch

Before we have a closer look at different methods for choosing k and initial centers we implement naive k-means from scratch. Later we will use Scikit-Learn's implementation.

First we create some synthetic data to be clustered.

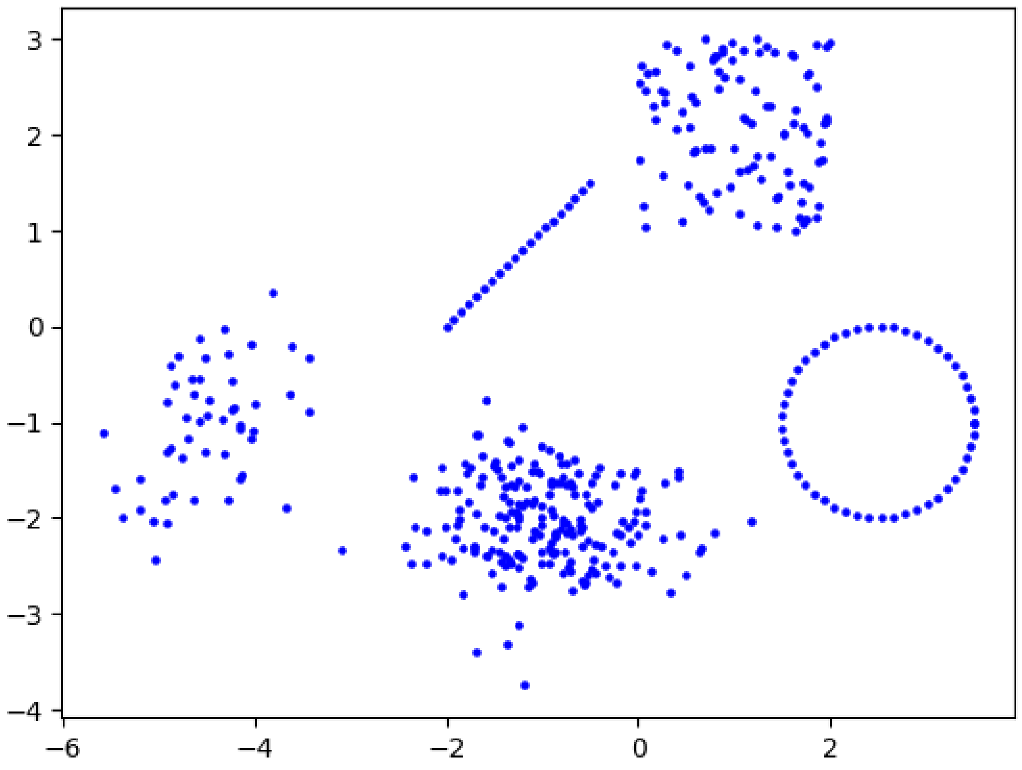
```
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(0)
```

```
# 5 clusters in 2 dimensions

n1, n2, n3, n4, n5 = 100, 200, 20, 50, 50
```

(continues on next page)

(continued from previous page)

```
n = n1 + n2 + n3 + n4 + n5

X1 = rng.uniform((0, 1), (2, 3), (n1, 2))
X2 = rng.multivariate_normal((-1, -2), ((0.5, 0), (0, 0.2)), n2)
X3 = np.linspace(-1, 0.5, n3).reshape(-1, 1) * np.ones((1, 2)) + np.array([[-1,␣
↪1]])
X4 = rng.multivariate_normal((-4.5, -1), ((0.2, 0.1), (0.1, 0.4)), n4)
phi = np.linspace(0, 2 * np.pi, n5).reshape(-1, 1)
X5 = np.array([[2.5, -1]]) + np.concatenate((np.cos(phi), np.sin(phi)), axis=1)

X = np.concatenate((X1, X2, X3, X4, X5))

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c='b')
ax.axis('equal')
plt.show()
```



k-means takes the number of clusters `k` and initial centers `init_centers` as parameters. Iteration is stopped if current and previous clustering coincide. To prevent too many iteration we also set a maximum number of iterations `max_iter`.

Center points will be stored rowwise in NumPy arrays (like for the samples in `X`). In each iteration we have to calculate all distances between samples and center points. Those distances will be stored in an `(n, k)` NumPy array `dists`. A clustering will be represented by a 1d NumPy array `clusters` of length `n` containing for each sample the index of the closest center point. When updating the centers we have to keep the old centers to check the stopping criterion. After each k-means step we plot the intermediate result.

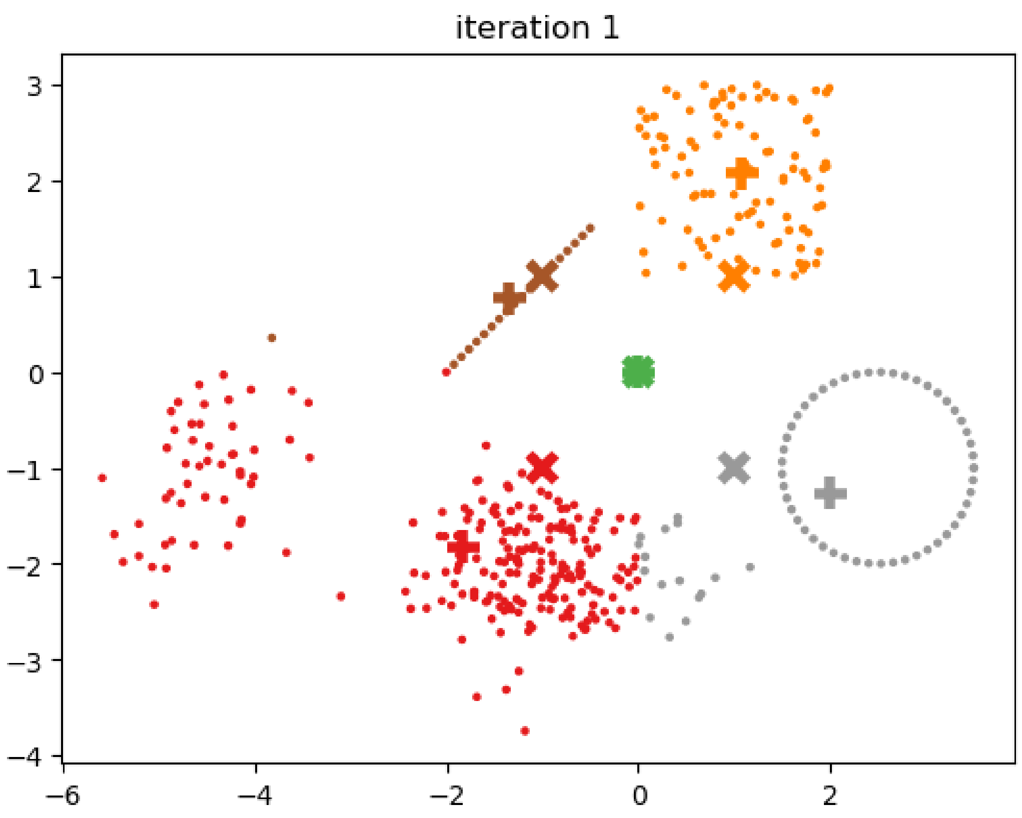
```
k = 5
init_centers = np.array([[-1, -1], [0, 0], [1, 1], [-1, 1], [1, -1]], dtype=float)
#k = 4
#init_centers = np.array([[-1, -1], [1, 1], [-1, 1], [1, -1]], dtype=float)
#k = 3
```

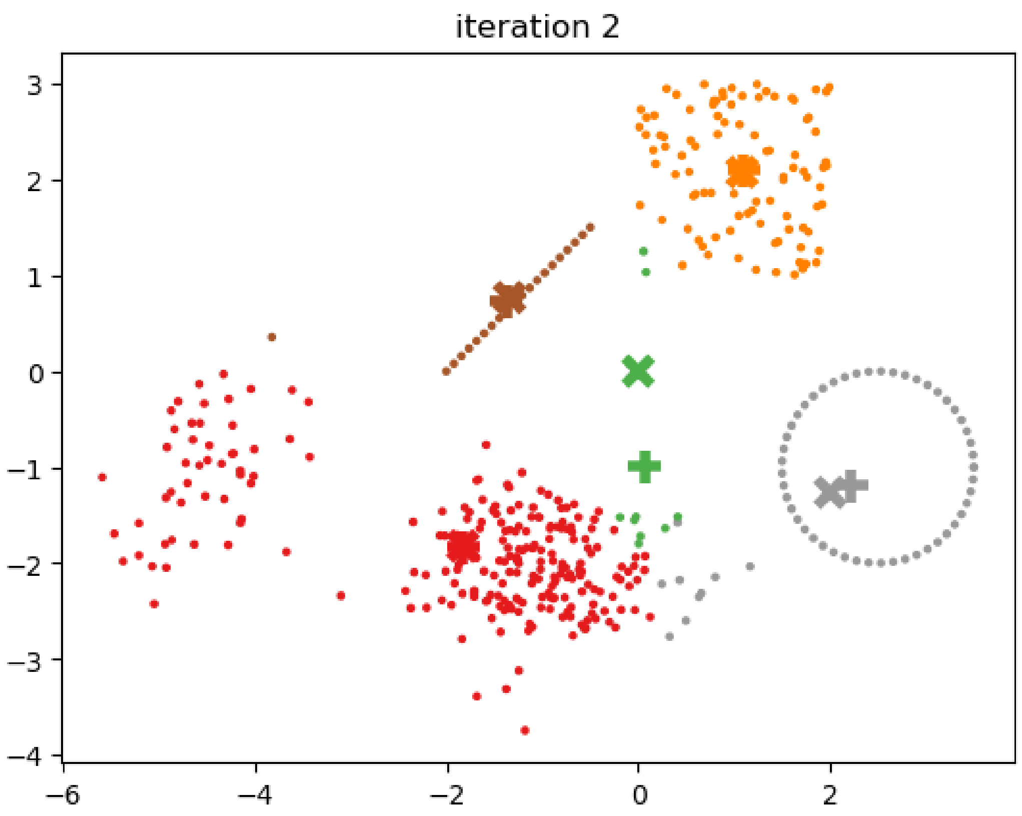
(continues on next page)

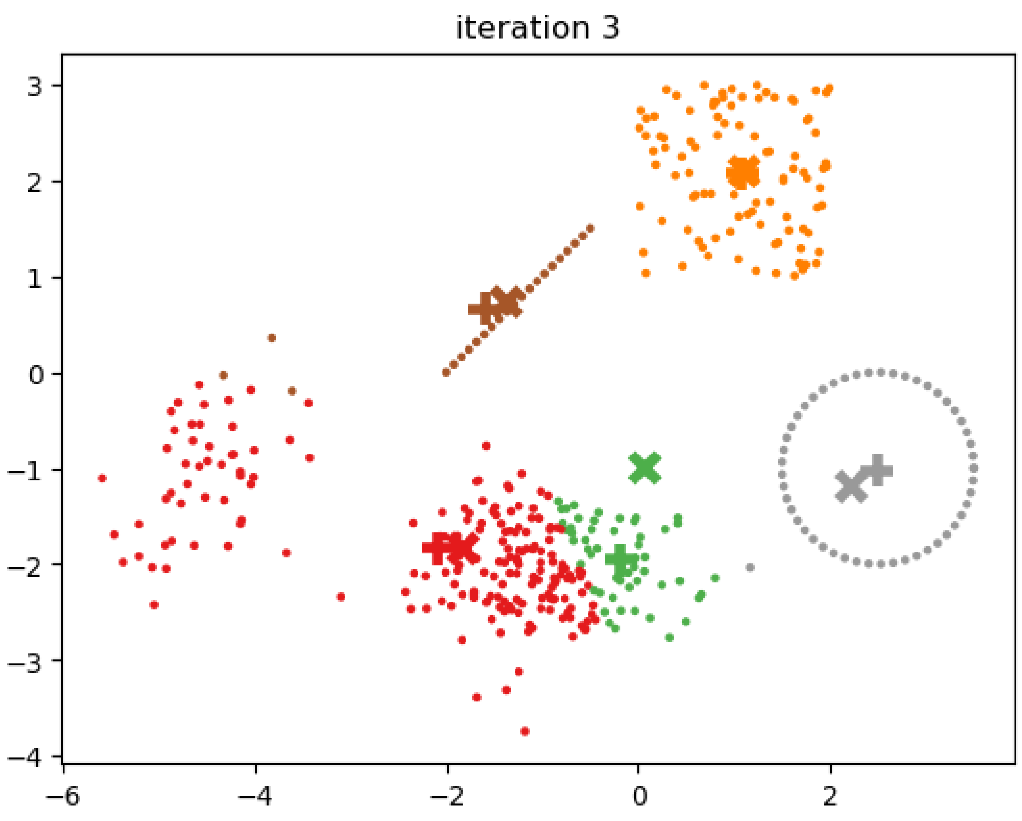
(continued from previous page)

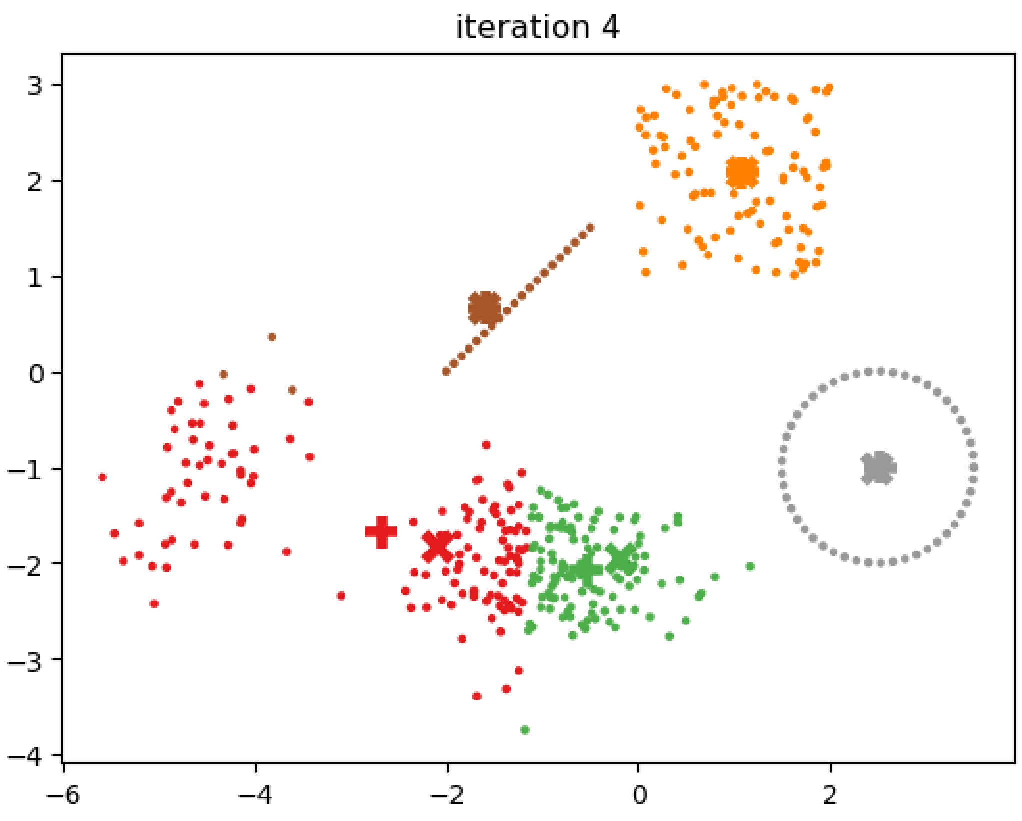
```
#init_centers = np.array([[-1, -1], [-1, 1], [1, -1]], dtype=float)
#k = 6
#init_centers = np.array([[-1, -1], [0, 0], [1, 1], [-1, 1], [1, -1], [0.5, 0.5]],
↪ dtype=float)
#k = 7
#init_centers = np.array([[-1, -1], [0, 0], [1, 1], [-1, 1], [1, -1], [0.5, 0.5],␣
↪[-0.5, -0.5]], dtype=float)

max_iter = 20

print('+ is current centroid')
print('x is previous centroid')

centers = init_centers
for i in range(0, max_iter):

    fig, ax = plt.subplots()

    # get clusters
    if i > 0:
        old_clusters = clusters
    dists = np.empty((n, k))
    for l in range(0, k):
        dists[:, l] = np.sum((X - centers[l, :]) ** 2, axis=1)
    clusters = dists.argmin(axis=1)

    # plot clusters and centers
    ax.scatter(X[:, 0], X[:, 1], s=5, c=clusters, cmap='Set1')
    ax.scatter(centers[:, 0], centers[:, 1], s=100, c=range(0, k), cmap='Set1',␣
↪marker='x', linewidth=4)

    # update centers
    old_centers = centers.copy()
    for l in range(0, k):
        if np.any(clusters == l):    # update only if cluster is not empty
            centers[l, :] = X[clusters == l, :].mean(axis=0)
        else:
            print('empty cluster')

    # stopping criterion
    if i > 0 and (old_clusters == clusters).all():
        print('stopping criterion satisfied after {} iterations'.format(i))
        break

    # plot new centers
    ax.scatter(centers[:, 0], centers[:, 1], s=150, c=range(0, k), cmap='Set1',␣
↪marker='+', linewidth=4)

    ax.set_title('iteration ' + str(i + 1))
    ax.axis('equal')

    plt.show()

else:
    print('max_iter reached')
```

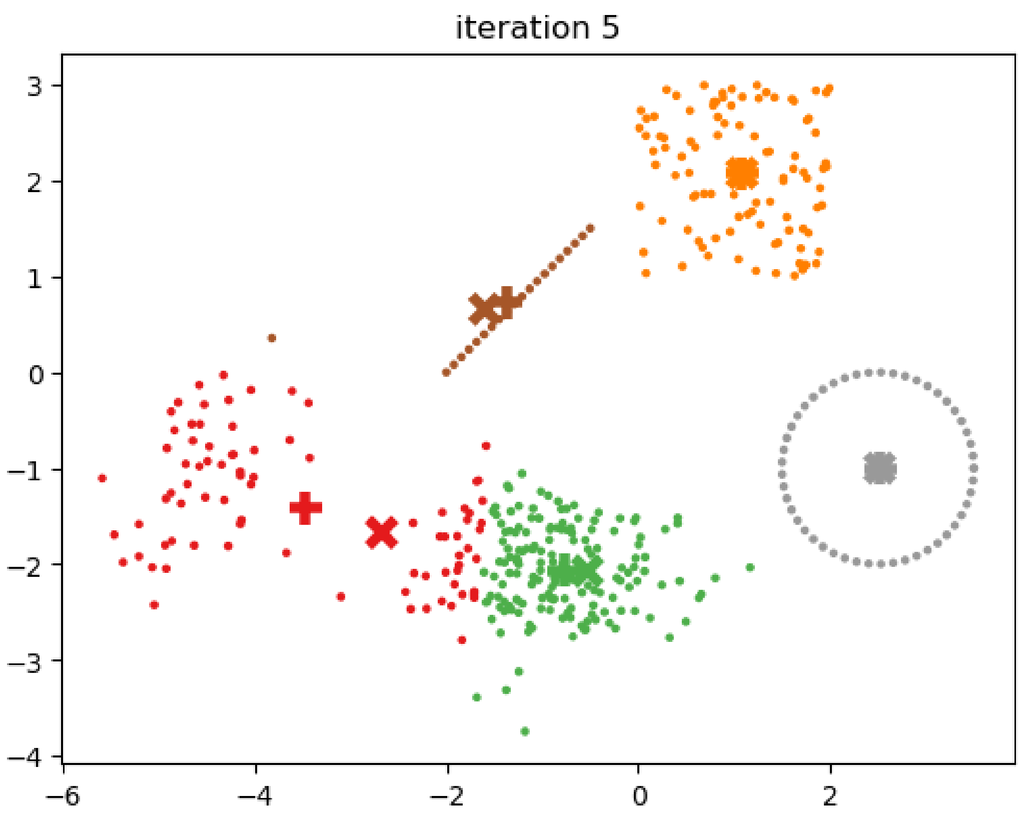
```
+ is current centroid
x is previous centroid
empty cluster
```

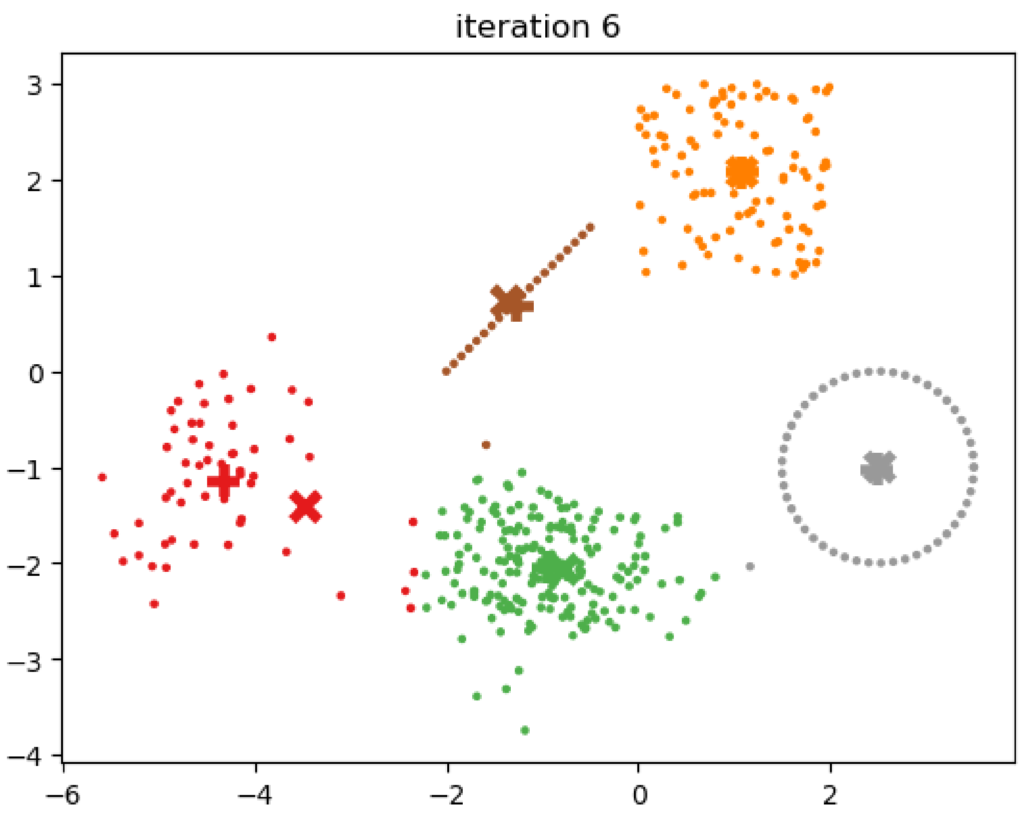

iteration 1
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2

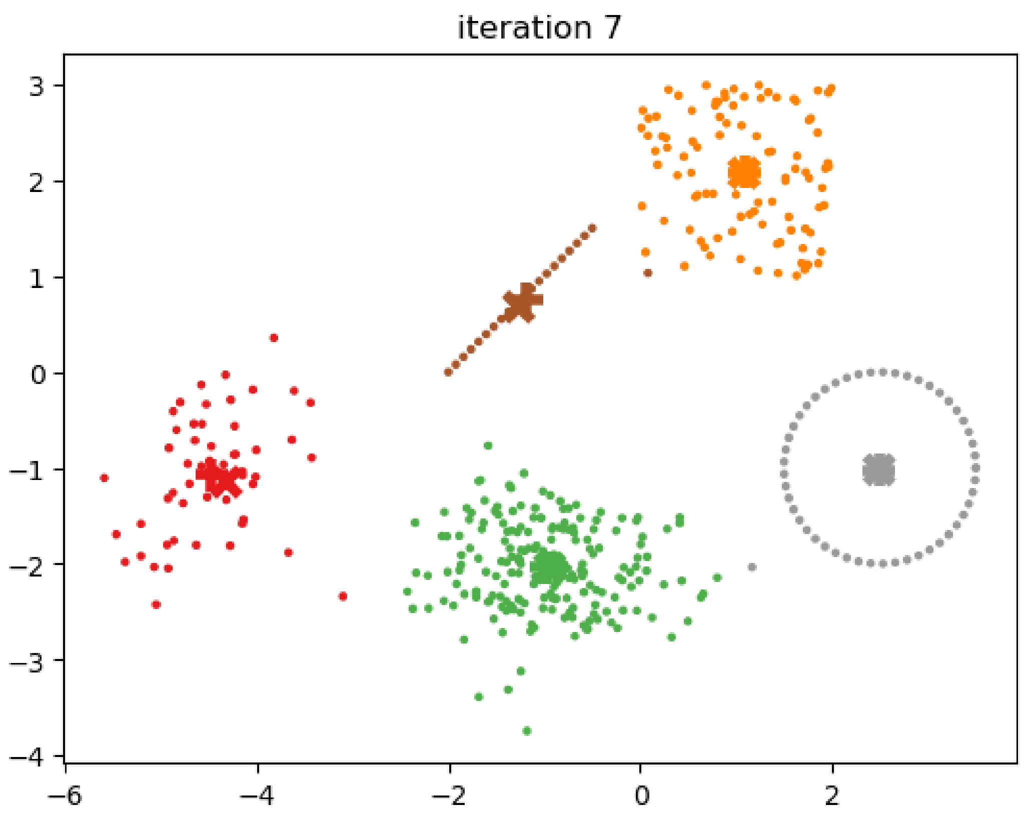

iteration 2
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2


iteration 3
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2

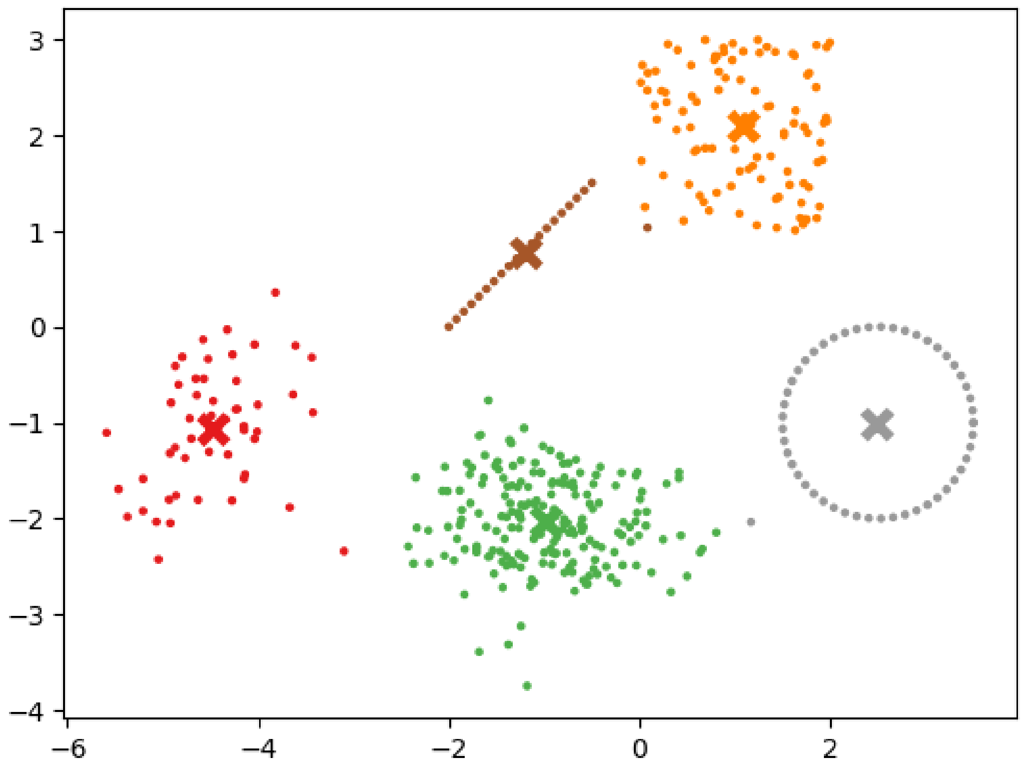

iteration 4
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2


iteration 5
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2


iteration 6
3
2
1
0
−1
−2
−3
−4
−6
−4
−2
0
2



```
stopping criterion satisfied after 7 iterations
```



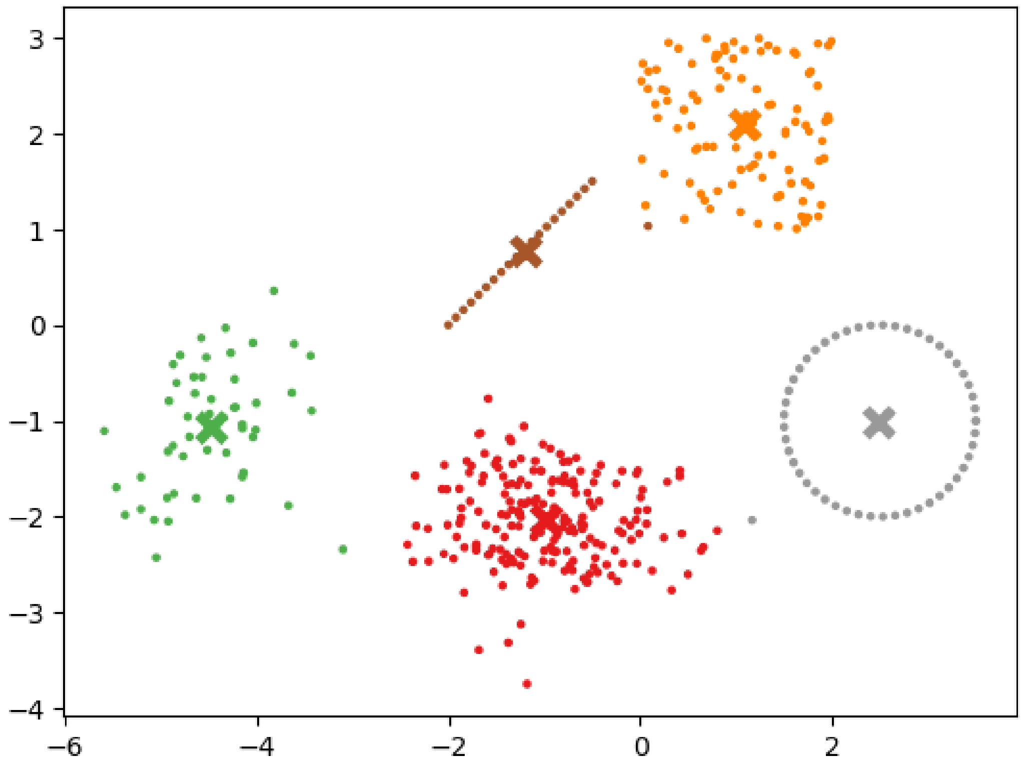
3.4 Implementation with Scikit-Learn

Scikit-Learn implements a `KMeans`[13] class in its `cluster` module. The `fit` method generates the clustering, resulting in a list of cluster centers. The `predict` method assigns cluster labels to samples. Note that `fit` already computes labels for training data. So we do not have to call `predict` on training data.

The `KMeans` constructor takes several arguments described in the documentation. Values for the `init` parameter will be described below. In case of random initialization `n_init` is the number of k-means runs. From multiple runs Scikit-Learn chooses the result with lowest inertia, that is, with smallest sum of squared distances of samples to their closest cluster center. Next to naive k-means Scikit-Learn supports the (often) more efficient Elkan method, which uses the triangle inequality to avoid unnecessary distance calculations.

```
import sklearn.cluster as cluster
```

```
k = 5
init_centers = np.array([[-1, -1], [0, 0], [1, 1], [-1, 1], [1, -1]], dtype=float)

km = cluster.KMeans(n_clusters=k, init=init_centers, n_init=1)
km.fit(X)
centers = km.cluster_centers_
clusters = km.labels_

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c=clusters, cmap='Set1')
ax.scatter(centers[:, 0], centers[:, 1], s=100, c=range(0, k), cmap='Set1',␣
↪marker='x', linewidth=4)
ax.axis('equal')
plt.show()
```



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

3.5 Random Initialization

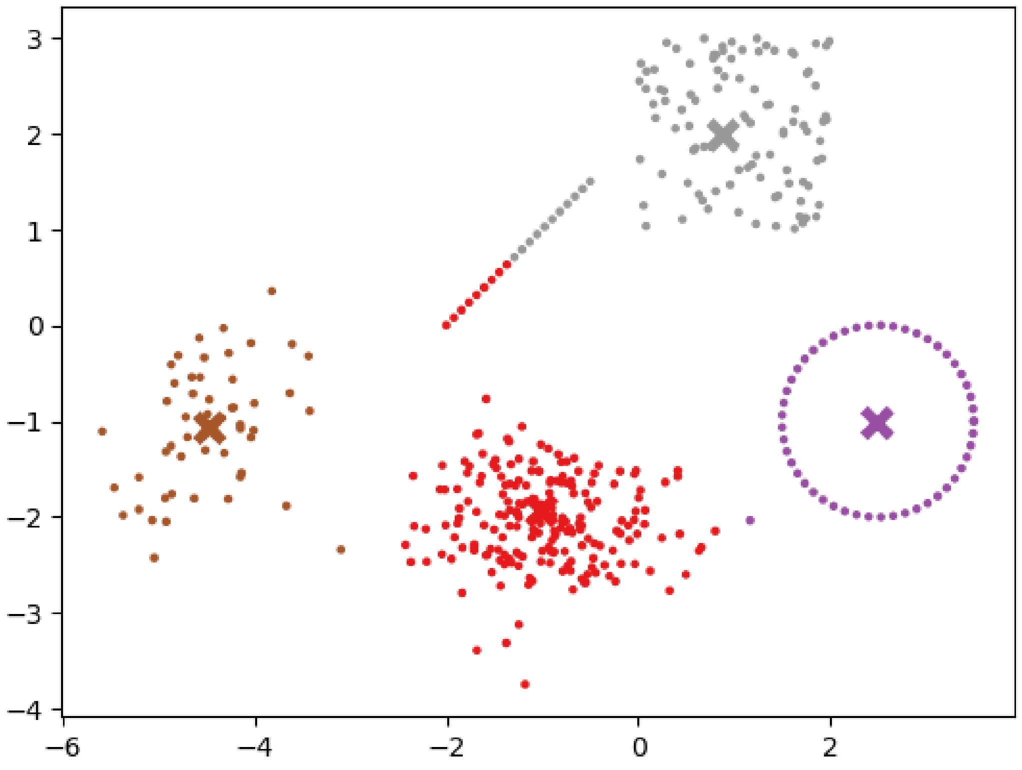
There exist (at least) two approaches for randomly choosing initial cluster centers:

- Choosing k random samples from the training data.
- Randomly assign samples to k clusters and take the centroids as initial centers.

The second approach results in closely spaced initial centers, whereas the first yields initial centers scattered over the data set. Scikit-Learn only implements the first approach, which can be activated with `init='random'`.

If we use random initialization the `n_init` parameter should be greater than one (defaults to 10). Scikit-Learn will run the algorithm `n_init` times and choose the best result (lowest inertia).

```
k = 4

km = cluster.KMeans(n_clusters=k, init='random', n_init=10)
km.fit(X)
centers = km.cluster_centers_
clusters = km.labels_

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c=clusters, cmap='Set1')
ax.scatter(centers[:, 0], centers[:, 1], s=100, c=range(0, k), cmap='Set1',␣
↪marker='x', linewidth=4)
ax.axis('equal')
plt.show()
```



3.6 k-Means++ for Initializing Centers

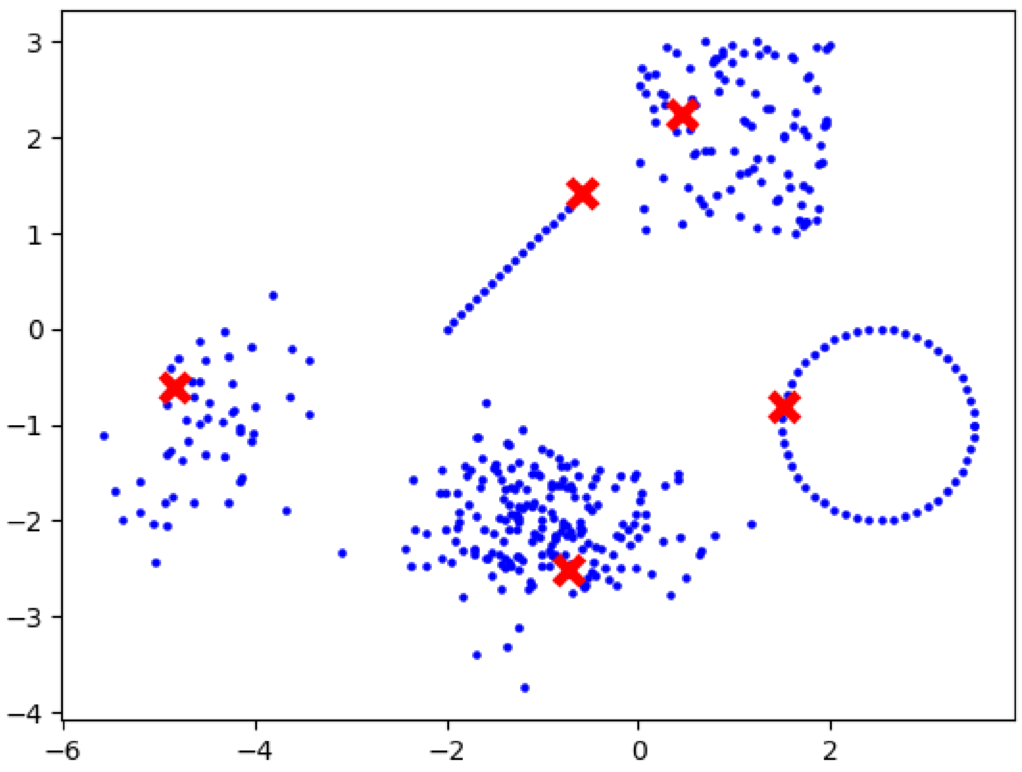
k-means++ is not an improved variant of k-means, but an algorithm for choosing good initial cluster centers for k-means. Here 'good' means fast convergence of k-means based on the initial centers.

k-means++ chooses all initial centers randomly from the training data set (like random initialization), but only the first center is chosen uniformly at random. After the first center has been chosen the following procedure is repeated until k centers have been found:

- For each sample (except the ones already chosen as center point) calculate the distance to the closest center.
- Choose a sample as next center point at random with probabilities proportional to the squared distances.

Samples far away from already existing center points are more likely to become the next center. Thus, initial centers chosen by k-means++ will be scattered over the whole data set and closely spaced centers are very unlikely.

```
k = 5

init_centers = np.empty((k, X.shape[1]))

# first center uniformly at random
init_centers[0, :] = X[rng.integers(0, X.shape[0]), :]

for l in range(1, k):

    # calculate distances to closest center (l centers already exist)
    dists = np.empty((n, l))
    for j in range(0, l):
        dists[:, j] = np.sum((X - init_centers[j, :]) ** 2, axis=1)
    min_dists = dists.min(axis=1)

    # next center with probability proportional to distance to closest center
    init_centers[l, :] = rng.choice(X, 1, p=min_dists/min_dists.sum())

# plot
fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c='b')
ax.scatter(init_centers[:, 0], init_centers[:, 1], s=100, c='r', marker='x',␣
↪linewidth=4)
ax.axis('equal')
plt.show()
```

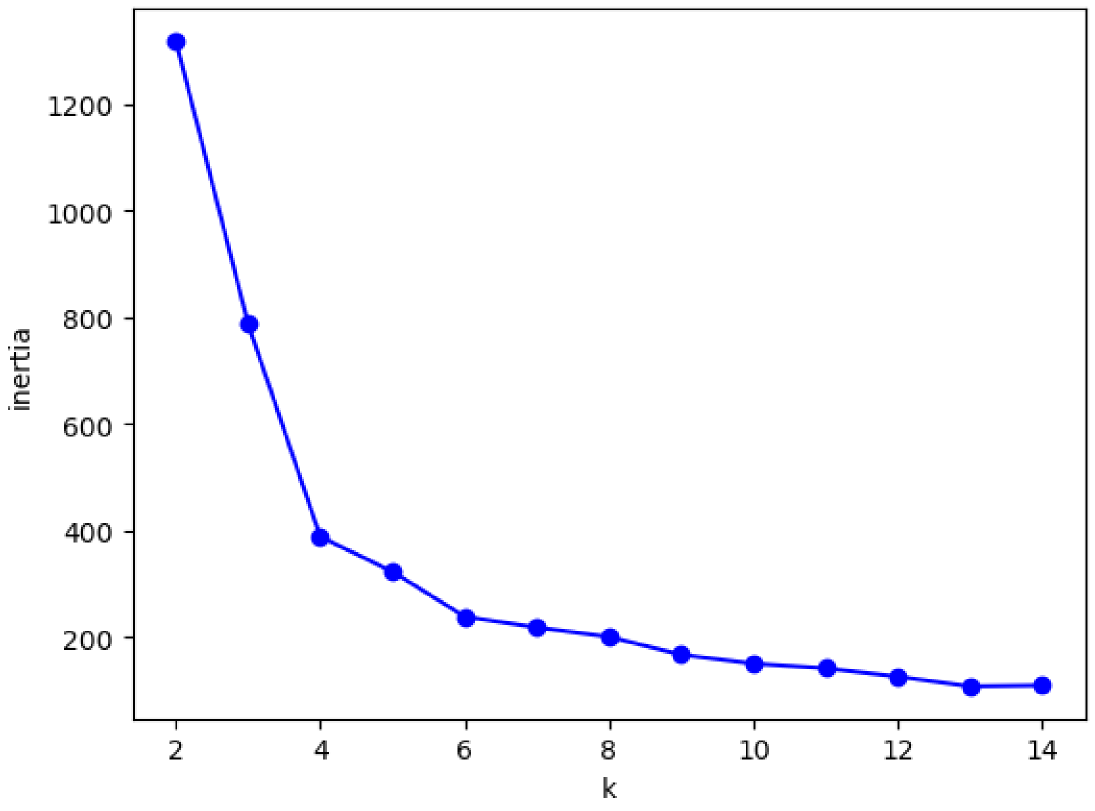


3.7 Choosing k

k-means aims to minimize the sum of distances of all samples to the closest cluster center. Thus, a suitable k should yield small objective function. Obviously $k = n$ with each sample being a cluster center would be the best choice, but this is not our intention when clustering data. Instead we want to have a sensible number of clusters.

Starting with $k = 2$ we may run k-means for an increasing sequence of values for k. The more clusters we allow, the smaller the objective in the above minimization problem. If k becomes larger than the number of clusters in the data, then the drop in the objective will be much smaller than for smaller k. If we plot the objective values against the number of clusters we should see an elbow-like curve. The k at the elbow should be chosen. This heuristic approach is known as *Elbow method*. Often it works quite well, but especially in case of clusters not well separated the elbow may be hard to identify. Another drawback is, that reliable automatic detection of the elbow's position is difficult.

```
ks = range(2, 15)

obj_values = []
for k in ks:
    km = cluster.KMeans(n_clusters=k, n_init='auto')
    km.fit(X)
    obj_values.append(km.inertia_)

fig, ax = plt.subplots()
ax.plot(ks, obj_values, '-ob')
ax.set_xlabel('k')
ax.set_ylabel('inertia')
plt.show()
```



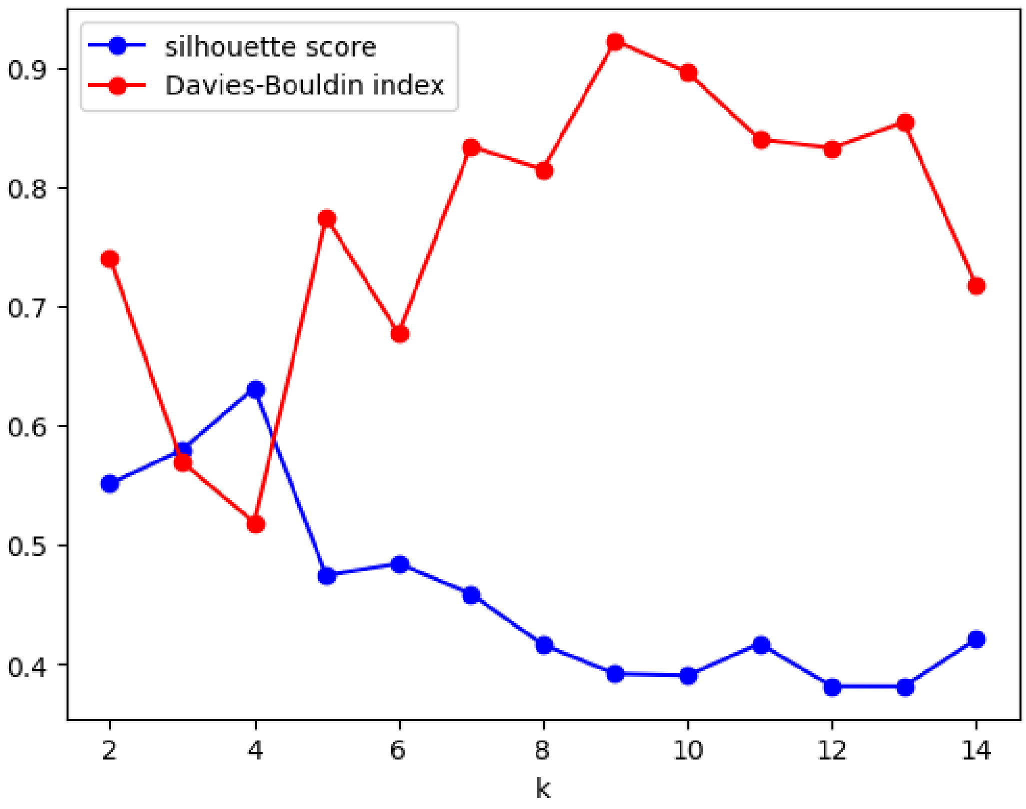
The elbow method here suggests $k = 4$.

An alternative to the elbow method is to consider quality measures for clusterings for different k. Scikit-Learn provides implementations for the silhouette score (`silhouette_score`[14]) and for the Davies-Bouldin index (`davies_bouldin_score`[15]) and for many other clustering metrics[16].

```
import sklearn.metrics as metrics
```

```
ks = range(2, 15)

sil = []
db = []
for k in ks:
    km = cluster.KMeans(n_clusters=k, n_init='auto')
    km.fit(X)
    sil.append(metrics.silhouette_score(X, km.labels_))
    db.append(metrics.davies_bouldin_score(X, km.labels_))

fig, ax = plt.subplots()
ax.plot(ks, sil, '-ob', label='silhouette score')
ax.plot(ks, db, '-or', label='Davies-Bouldin index')
ax.set_xlabel('k')
ax.legend()
plt.show()
```

[14] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html
[15] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.davies_bouldin_score.html
[16] https://scikit-learn.org/stable/modules/classes.html#clustering-metrics



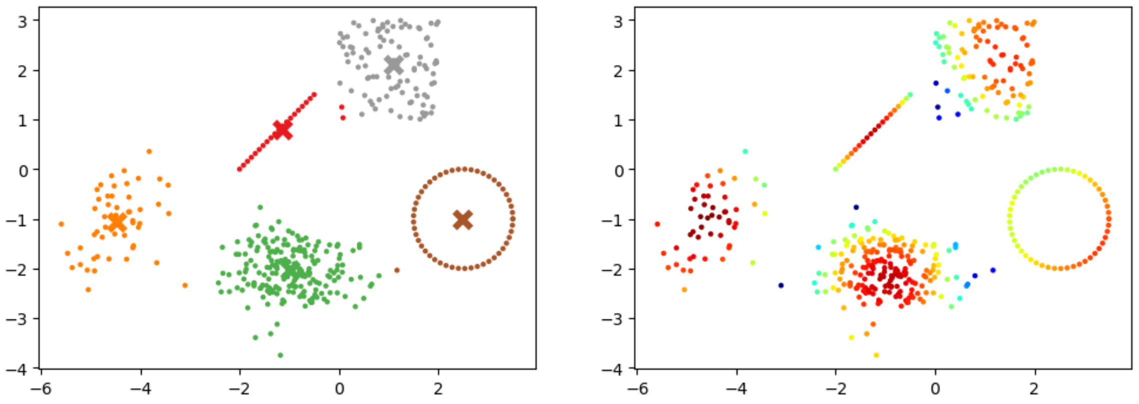
Both scores suggest $k = 4$.

3.8 Per Sample Silhouette Score

To judge on the quality of a clustering we may also look at each sample's silhouette score. This way we can identify missclusterings or samples with uncertain cluster assignment. Scikit-Learn provides `silhouette_samples`[17] for this purpose.

```
k = 5

km = cluster.KMeans(n_clusters=k, n_init='auto')
km.fit(X)
centers = km.cluster_centers_
clusters = km.labels_
sil = metrics.silhouette_samples(X, km.labels_)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.scatter(X[:, 0], X[:, 1], s=5, c=clusters, cmap='Set1')
ax1.scatter(centers[:, 0], centers[:, 1], s=100, c=range(0, k), cmap='Set1',␣
↪marker='x', linewidth=4)
ax1.axis('equal')
ax2.scatter(X[:, 0], X[:, 1], s=5, c=sil, cmap='jet')
ax2.axis('equal')
plt.show()
```

[17] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_samples.html



Red points (high silhouette score) clearly belong to their cluster, blue points (low silhouette score) also could belong to a different or to no cluster.

3.9 k-Means for Very Large Data Sets

k-means requires as many distance computations per cluster and per iteration as there are samples in the training data set. To reduce computation time for large data sets we may

- use only a subset of the training data in each iteration (subsets change from iteration to iteration) and
- update center points sample by sample instead of cluster by cluster.

Working with subsets obviously reduces the amount of distance calculations. Depending on the randomly chosen subsets, cluster centers may change very much from iteration to iteration. To prevent such hopping, updates are calculated sample by sample, resulting in slightly different update results. A side effect of the sample-by-sample approach is a more efficient implementation of the update step. Details are given in Web-Scale K-Means Clustering[18].

Scikit-Learn implements this approach as `MiniBatchKMeans`[19].

3.10 When to Use k-Means?

k-means is very fast (at least faster than most other clustering algorithms). But it has some drawbacks one has to be aware of.

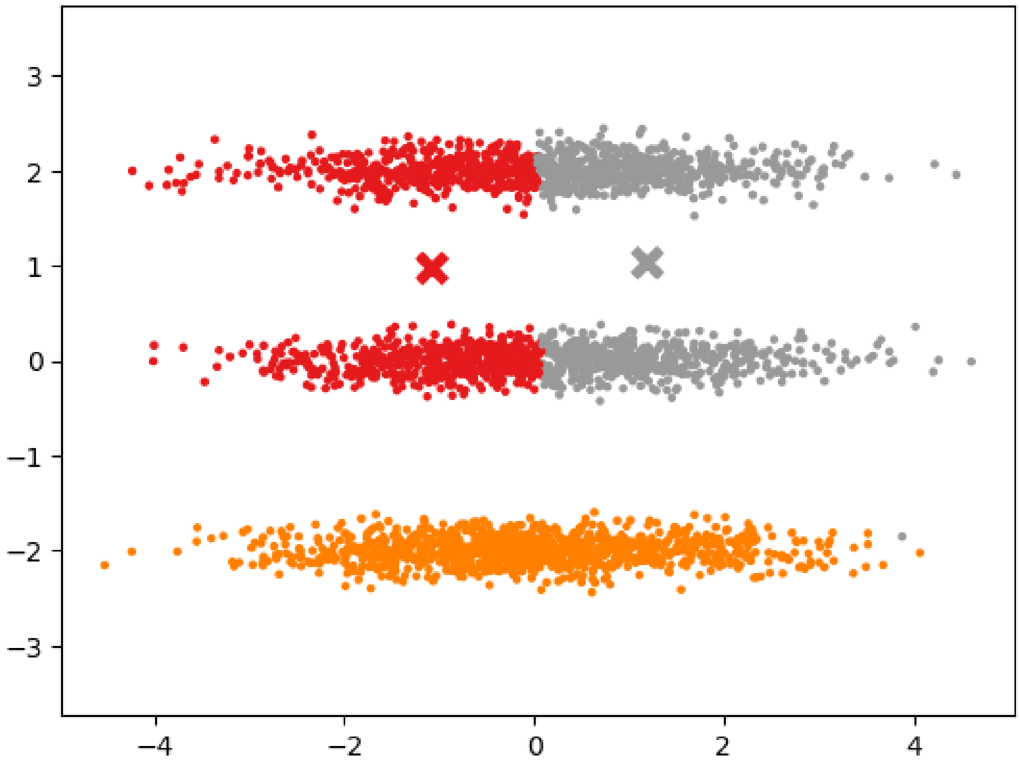
For very high dimensional data (thousands of features, images for instance) distances between samples do not carry useful information because all distances are almost identical (cf. discussion of curse of dimensionality in). Thus, k-means wont work. Resulting clusters will look like samples were assigned to clusters uniformly at random. Dimension reduction techniques (e.g. PCA) should be applied before trying k-means.

k-means prefers convex and sphere shaped clusters. Proper scaling of the data may improve results drastically, especially if features have very different numerical ranges. Alternatively the used distance measure has to be adapted to the data under consideration (weighted Euclidean distance, for instance).

```
n1, n2, n3 = 1000, 1000, 1000
n = n1 + n2 + n3

X1 = rng.multivariate_normal((0, -2), ((2, 0), (0, 0.02)), n1)
X2 = rng.multivariate_normal((0, 0), ((2, 0), (0, 0.02)), n2)
X3 = rng.multivariate_normal((0, 2), ((2, 0), (0, 0.02)), n3)

X = np.concatenate((X1, X2, X3))
```

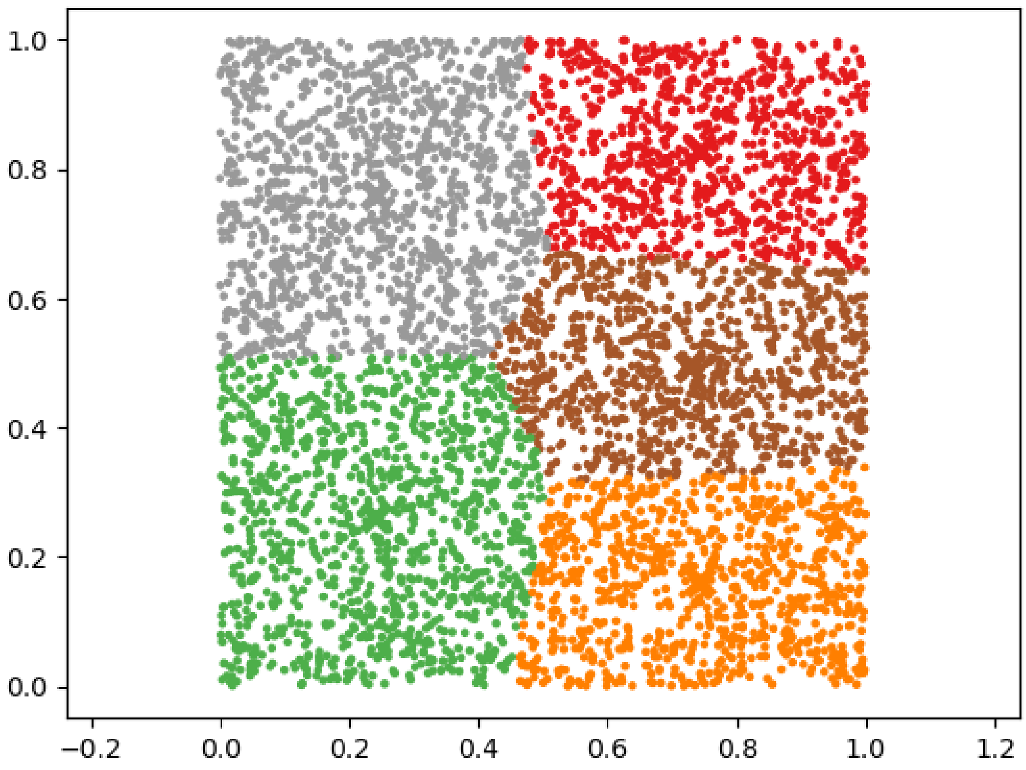
[18] https://web.archive.org/web/20230210073106/https://www.eecs.tufts.edu/~dsculley/papers/fastkmeans.pdf
[19] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html

```
k = 3

km = cluster.KMeans(n_clusters=k, n_init='auto')
km.fit(X)
centers = km.cluster_centers_
clusters = km.labels_

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c=clusters, cmap='Set1')
ax.scatter(centers[:, 0], centers[:, 1], s=100, c=range(0, k), cmap='Set1',␣
 ↪marker='x', linewidth=4)
ax.axis('equal')
plt.show()
```



Due to its distance-to-center based nature k-means prefers clusters with similar spatial extent.

```
n = 5000
X = rng.uniform((0, 0), (1, 1), (n, 2))

k = 5

km = cluster.KMeans(n_clusters=k, n_init='auto')
km.fit(X)
centers = km.cluster_centers_
clusters = km.labels_

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c=clusters, cmap='Set1')
ax.scatter(centers[:, 0], centers[:, 1], s=100, c=range(0, k), cmap='Set1',␣
 ↪marker='x', linewidth=4)
ax.axis('equal')
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2

CHAPTER

FOUR

HIERARCHICAL CLUSTERING

Hierarchical clustering algorithms not only yield one clustering but a series of clusterings. There are two approaches for generating a series of clusters:

- *coarse to fine* or *divisive* (the first clustering consists of only one cluster (the whole training data set) and the last contains as many clusters as there are samples),
- *fine to coarse* or *agglomerative* (the first clustering contains as many clusters as samples and the last clustering has only one large cluster).

In the first case the next clustering results from splitting a cluster in the previous clustering by some rule. In the second case two clusters are selected by some rule and than joined to form only one cluster. Both variants result in a nested sequence of clusters, usually (not always) with increasing intercluster dissimilarity from fine to coarse.

Hierarchical clustering methods do not imply canonical prediction routines (in contrast to k-means). They only assign labels to the training data. To add new data points to existing clusters kNN or some other supervised learning technique has to be used.

Related projects:

- *Chinese Celadons* (page 131)
  - *Hierarchical Clustering* (page 131)

4.1 Dendrograms

The aim of hierarchical clustering is not only to find a good clustering, but to better understand structures in the data set. A *dendrogram* is a graphical representation of a sequence of nested clusterings. On the one hand, a dendrogram shows relations between clusters of different clusterings. On the other hand, it provides information about dissimilarity of clusters within one clustering.

A dendrogram is a binary (that is, two children per parent node) tree. Each node represents a subset (cluster) of the data set. The whole data set is the root node and there are as many leaves as there are samples in the data set. Each intermediate node represents the union of its two children. The height of a node in the tree is related to the value of a dissimilarity measure between clusters which lead to the split or join operation.

Dendrograms can be used to determine the number of clusters in a data set. The wider the height gap between two nodes, the better the clustering (with respect to the chosen distance). Here 'better' means that the clustering corresponding to a wide gap is very stable with respect to the merging criterion, that is, contains no subclusters with low dissimilarity. See below for an example.

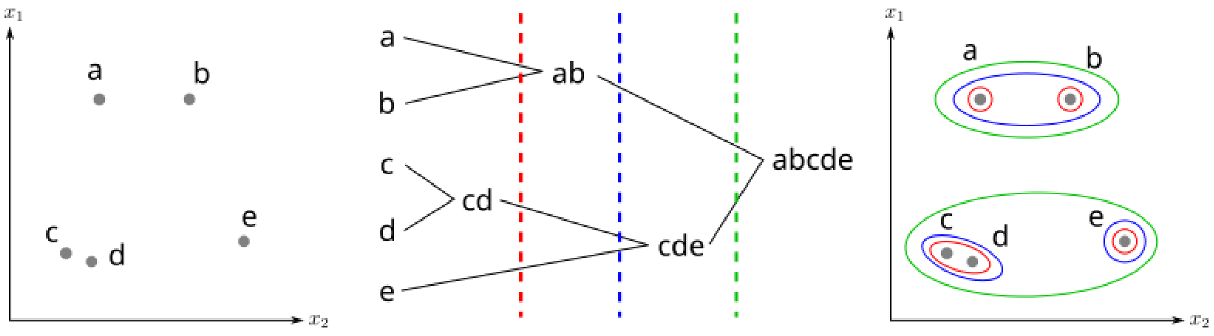


Fig. 4.1: A dendrogram and corresponding clusters.

4.2 Divisive Clustering

Divisive clustering is rarely used in practice, because good splitting stategies are computationally expensive. One possible approach is to find the cluster with highest intracluster dissimilarity and then use k-means with $k = 2$ for splitting the cluster. Depending on the chosen dissimilarity measure dissimilarity values at subsequent splits may be nondecreasing. Thus, height of nodes in the dendrogram cannot encode dissimilarity.

4.3 Agglomerative Clustering

For agglomerative clustering we have to choose a point-to-point distance d (Euclidean distance, for instance) and a distance D between two disjoint sets. Next to some relatively uncommon variants there are three major notions for distances between disjoint sets:

- minimum distance of points (*single linkage clustering*):

$$D(C,\tilde{C}) := \min_{x\in C,\tilde{x}\in\tilde{C}} d(x,\tilde{x}),$$

- maximum distance of points (*complete linkage clustering*):

$$D(C,\tilde{C}) := \max_{x\in C,\tilde{x}\in\tilde{C}} d(x,\tilde{x}),$$

- average distance of points (*average linkage clustering*):

$$D(C,\tilde{C}) := \frac{1}{|C|\,|\tilde{C}|}\sum_{x\in C}\sum_{\tilde{x}\in\tilde{C}} d(x,\tilde{x}).$$

Starting with the finest clustering the following steps are repeated until there is only one large cluster containing the whole data set:

- Calculate the distance D for each pair of clusters.
- Join the two clusters with smallest distance.

Clustering results for different distance measures may differ significantly. Single linkage clustering suffers from *chaining*. That is, several (for human eyes) clearly separated clusters are joined into one because they touch at one point. An advantage of single linkage clustering is that it finds clusters of arbitrary shape and does not prefer convex clusters. Complete linkage prefers compact sphere shaped clusters. Average linkage is a compromise between both extremes. Have a look at the plots of the linkage example in Scikit-Learn's documentaion[20].

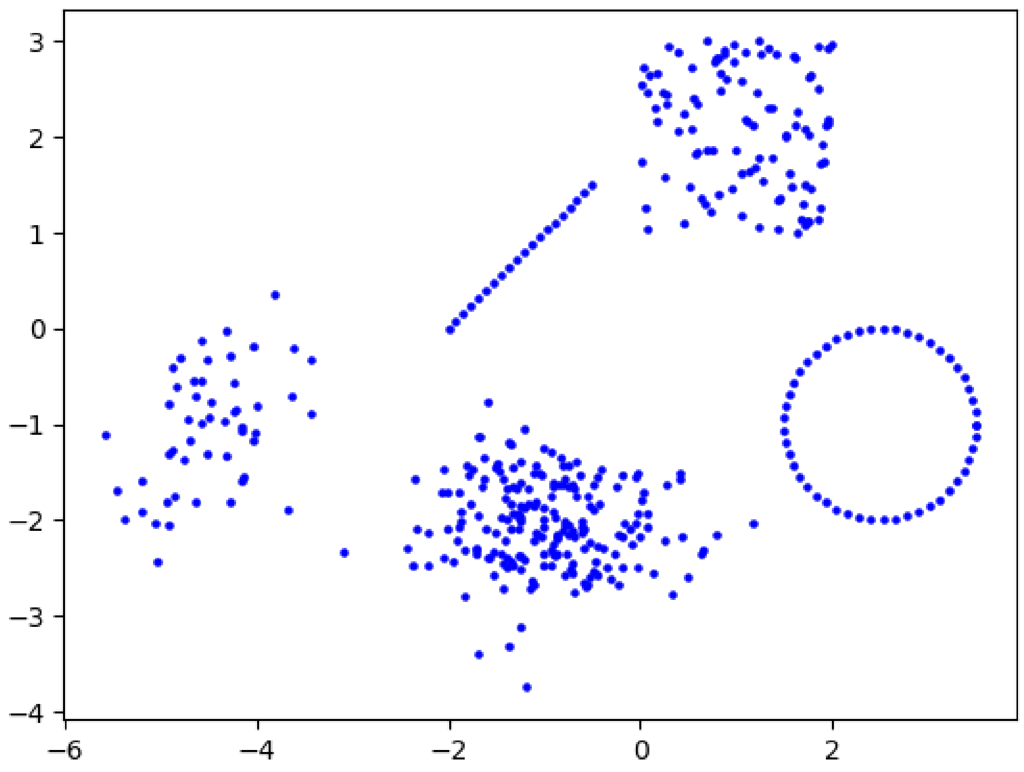
[20] https://scikit-learn.org/stable/auto_examples/cluster/plot_linkage_comparison.html

4.4 Hierarchical Clustering with Scikit-Learn

Scikit-Learn only supports agglomerative clustering via `AgglomerativeClustering`[21] class. We either have to provide the number of clusters `n_clusters` or the distance `distance_threshold` at which to cut the dendrogram.

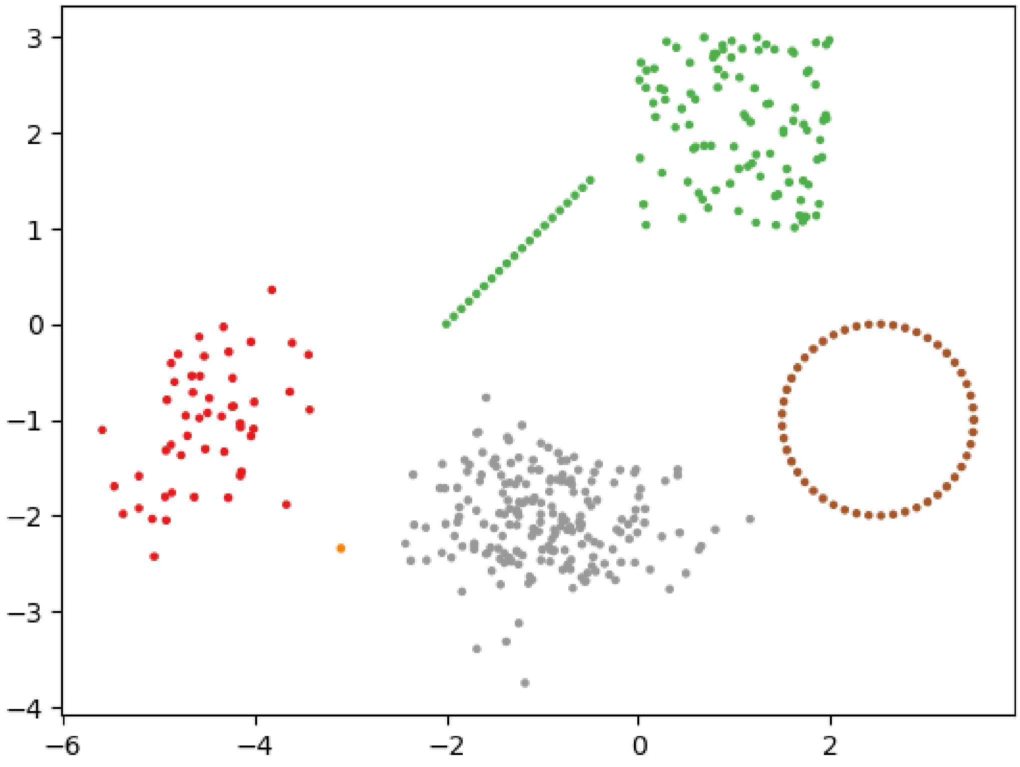
```
import numpy as np
import matplotlib.pyplot as plt
import sklearn.cluster as cluster

rng = np.random.default_rng(0)
```

```
# 5 clusters in 2 dimensions

n1, n2, n3, n4, n5 = 100, 200, 20, 50, 50
n = n1 + n2 + n3 + n4 + n5

X1 = rng.uniform((0, 1), (2, 3), (n1, 2))
X2 = rng.multivariate_normal((-1, -2), ((0.5, 0), (0, 0.2)), n2)
X3 = np.linspace(-1, 0.5, n3).reshape(-1, 1) * np.ones((1, 2)) + np.array([[-1,␣
↪1]])
X4 = rng.multivariate_normal((-4.5, -1), ((0.2, 0.1), (0.1, 0.4)), n4)
phi = np.linspace(0, 2 * np.pi, n5).reshape(-1, 1)
X5 = np.array([[2.5, -1]]) + np.concatenate((np.cos(phi), np.sin(phi)), axis=1)

X = np.concatenate((X1, X2, X3, X4, X5))

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c='b')
ax.axis('equal')
plt.show()
```



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

```
ac = cluster.AgglomerativeClustering(5, linkage='single')
ac.fit(X)

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c=ac.labels_, cmap='Set1')
ax.axis('equal')
plt.show()
```



Neither Scikit-Learn, nor Matplotlib, nor Seaborn support plotting dendrograms directly. Only SciPy[22] has a `dendrogram`[23] function for plotting. SciPy also has the `linkage`[24] function for agglomerative clustering. But we would like to use Scikit-Learn for clustering although SciPy shall plot the dendrogram. Thus, we have to adapt Scikit-Learn's output to the needs of SciPy.

```
import scipy.cluster.hierarchy as scipy_ch
```

```
def plot_dendrogram(ac, ax, max_nodes=10, color_threshold=0.5):

    n = ac.labels_.shape[0]     # number of samples

    data = np.empty((n - 1, 4))
    data[:, 0:2] = ac.children_
    data[:, 2] = ac.distances_

    # get number of samples per node
    for i in range(0, n - 1):     # visit each node
        c = 0

        # add samples from left child
        if ac.children_[i, 0] < n:     # child is leaf
```

(continues on next page)

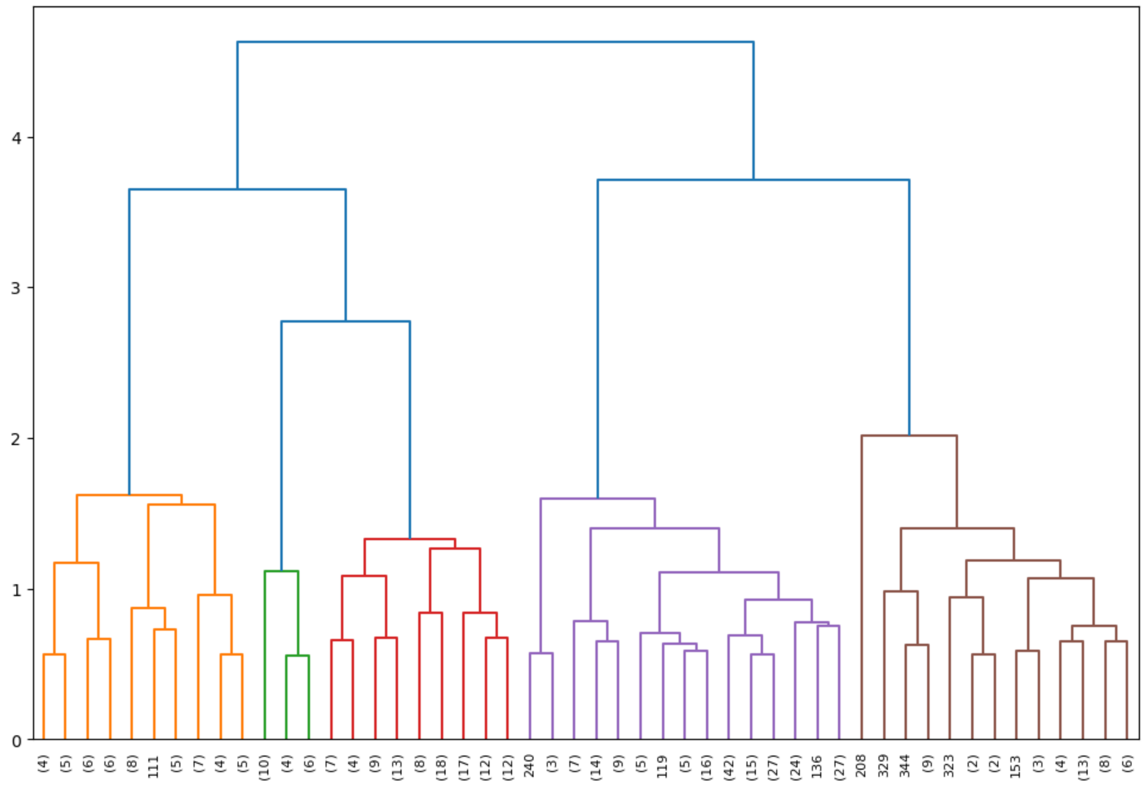
[22] https://docs.scipy.org/doc/scipy/index.html
[23] https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.dendrogram.html
[24] https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html

(continued from previous page)

```
            c = c + 1
        else:
            c = c + data[ac.children_[i, 0] - n, 3]

        # add samples from right child
        if ac.children_[i, 1] < n:     # child is leaf
            c = c + 1
        else:
            c = c + data[ac.children_[i, 1] - n, 3]
        data[i, 3] = c

    scipy_ch.dendrogram(data, ax=ax, truncate_mode='lastp', p=max_nodes, color_
↪threshold=color_threshold)
```

Note that the `n_clusters` parameter of `AgglomerativeClustering` does not matter, because we are interested in the dendrogram, not in a particular clustering.

```
ac = cluster.AgglomerativeClustering(linkage='average', compute_distances=True)
ac.fit(X)

fig, ax = plt.subplots(figsize=(12, 8))
plot_dendrogram(ac, ax, 50, 2.5)
plt.show()
```



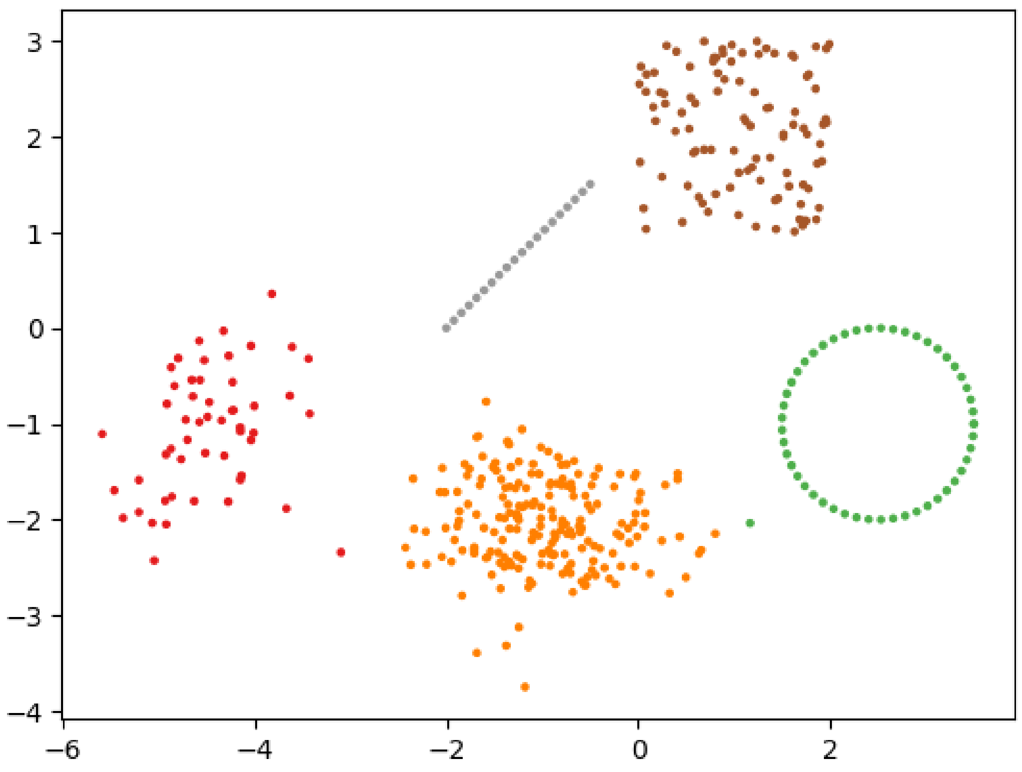
From the dendrogram we see that the number of clusters in the data is 2, 4 or 5.

```
ac = cluster.AgglomerativeClustering(n_clusters=5, linkage='average')
ac.fit(X)

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c=ac.labels_, cmap='Set1')
ax.axis('equal')
```

(continues on next page)

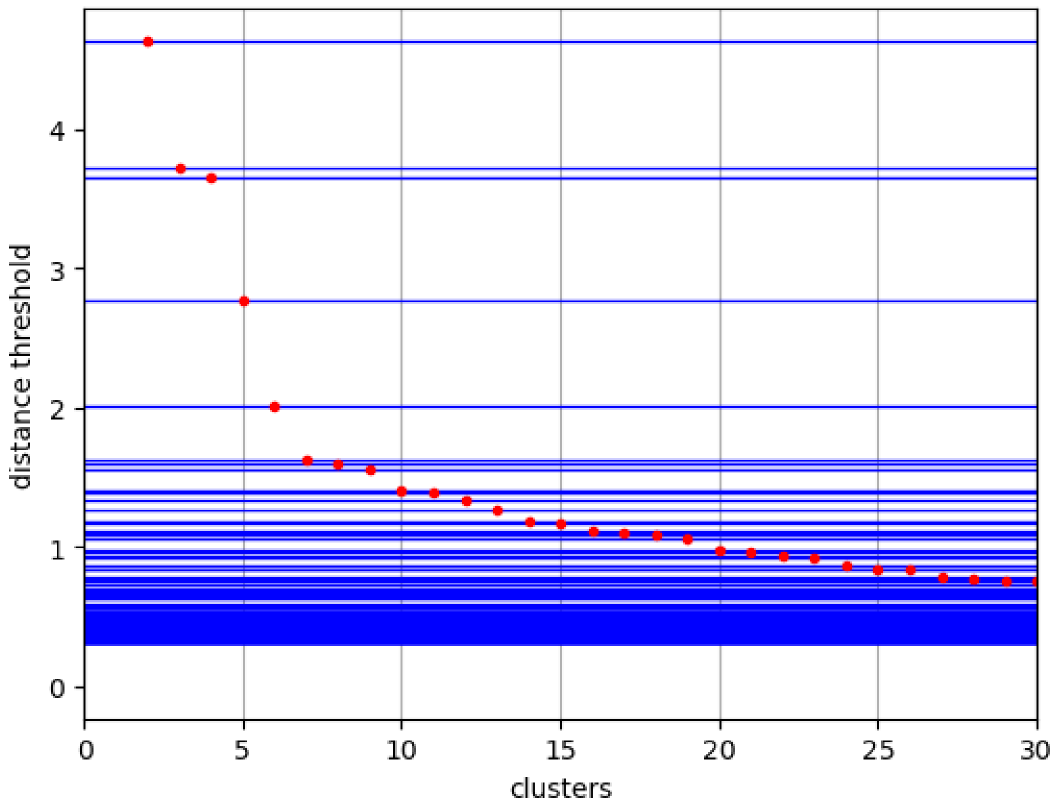
(continued from previous page)

```
plt.show()
```



If we only are interested in the number of clusters we may plot the number of clusters versus the distance threshold and look for wide vertical gaps.

```
ac = cluster.AgglomerativeClustering(linkage='average', compute_distances=True)
ac.fit(X)

fig, ax = plt.subplots()

dists = ac.distances_
for i in range(dists.size - 100, dists.size):
    ax.plot([0, dists.size], [dists[i], dists[i]], '-b', linewidth=1)
ax.plot(range(dists.size + 1, 1, -1), dists, 'or', markersize=3)

ax.set_xlim(0, 30)
ax.set_xlabel('clusters')
ax.set_ylabel('distance threshold')
ax.grid(axis='x')

plt.show()
```


distance threshold
clusters
0
1
2
3
4
0
5
10
15
20
25
30

CHAPTER

FIVE

DENSITY-BASED CLUSTERING

Density-based clustering aims at finding connected regions of closely spaced points in a data set. The basic idea is to mark *core points* (points with many surrounding points) and look for clusters in the set of core points. A distance measure is only used to determine a neighborhood for each point. Thus, the choice of a concrete distance measure is not as important as for centroid-based methods or hierarchical clustering.

There exist many ways to fill in the details of the approach. The most widely used density-based clustering algorithms are DBSCAN and OPTICS. Both only assign cluster labels to the training data but do not imply a canonical prediction routine. Both algorithms may yield clusters of arbitrary shape.

Another density-based clustering method, not discussed here, is mean shift[25].

Related projects:

5.1 DBSCAN

The DBSCAN (density-based spatial clustering of applications with noise) algorithm takes two parameters $\varepsilon > 0$ and $N \in \mathbb{N}$, which together specify what a *core point* is: a point x from the training data set is a core point if the ball of radius ε centered at x contains at least N points of the data set (including x). We may say that core points are points with dense neighborhood.

In the context of DBSCAN a cluster is a subset S of the data set with the following properties:

- S contains at least one core point $\bar{x}_0$.
- For each point x in S there is a finite sequence of core points $\bar{x}_1, \ldots, \bar{x}_k$ such that $\bar{x}_l$ belongs to the ε-neighborhood of $\bar{x}_{l-1}$ for $l = 1, \ldots, k$ and x belongs to the ε-neighborhood of $\bar{x}_k$.

Each cluster contains core points and may contain non-core points (points with few neighbors). Non-core points can be regarded as the cluster's edge. From the sequence property above one immediately sees that each core point of a cluster may take the role of $\bar{x}_0$.

There may exist points which do not belong to any cluster. Such points are regarded as noise or outliers.

DBSCAN starts with some point of the data set. If it's a non-core point, it is marked as visited and a new point is chosen. If it is a core point, a new cluster is started. All points from the ε-neighborhood are added to the cluster. Then all neighborhoods of core points in the starting point's neighborhood are added, and so on until there are no more reachable core points. Then a new unvisited point is chosen and the cluster discovery starts again from this new point (if it is a core point). If all points have been visited, the points not assigned to a cluster are interpreted as outliers.

[25] https://en.wikipedia.org/wiki/Mean-shift

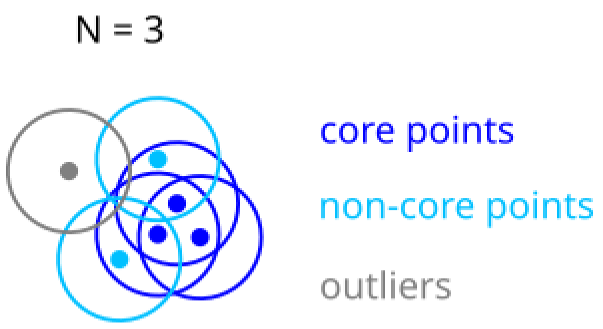


Fig. 5.1: DBSCAN classifies points as core points, non-core points, and outliers.

The computationally expensive part is to find all points in the ε-neighborhood of a given point. There exist efficient implementations for such *range queries* for data sets stored in well structured data bases.

A major advantage of DBSCAN is that we do not have to estimate the number of clusters in advance. Also results do not depend on (random) initializations. Although assignment of non-core points to clusters depends on the processing order of the points, clustering of core points is always the same.

Parameters ε and N may be hard to choose. But they have a clear interpretation. The smaller ε and the higher N the closer points have to be to form a cluster. Domain knowledge may help to choose these parameters. Another drawback is that there is only one set of parameters for all clusters. Consequently all clusters should show comparable density.

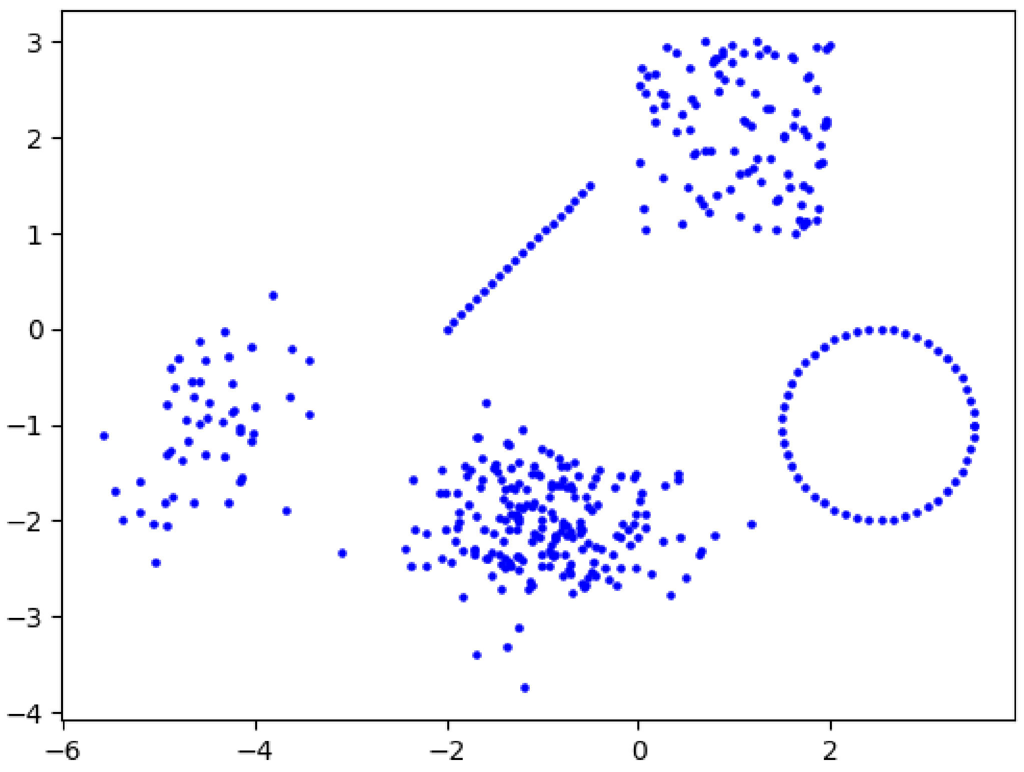
5.1.1 DBSCAN with Scikit-Learn

Scikit-Learn provides a DBSCAN[26] class in its `cluster` module. Relevant parameters are `eps` (our ε) and `min_samples` (our N).

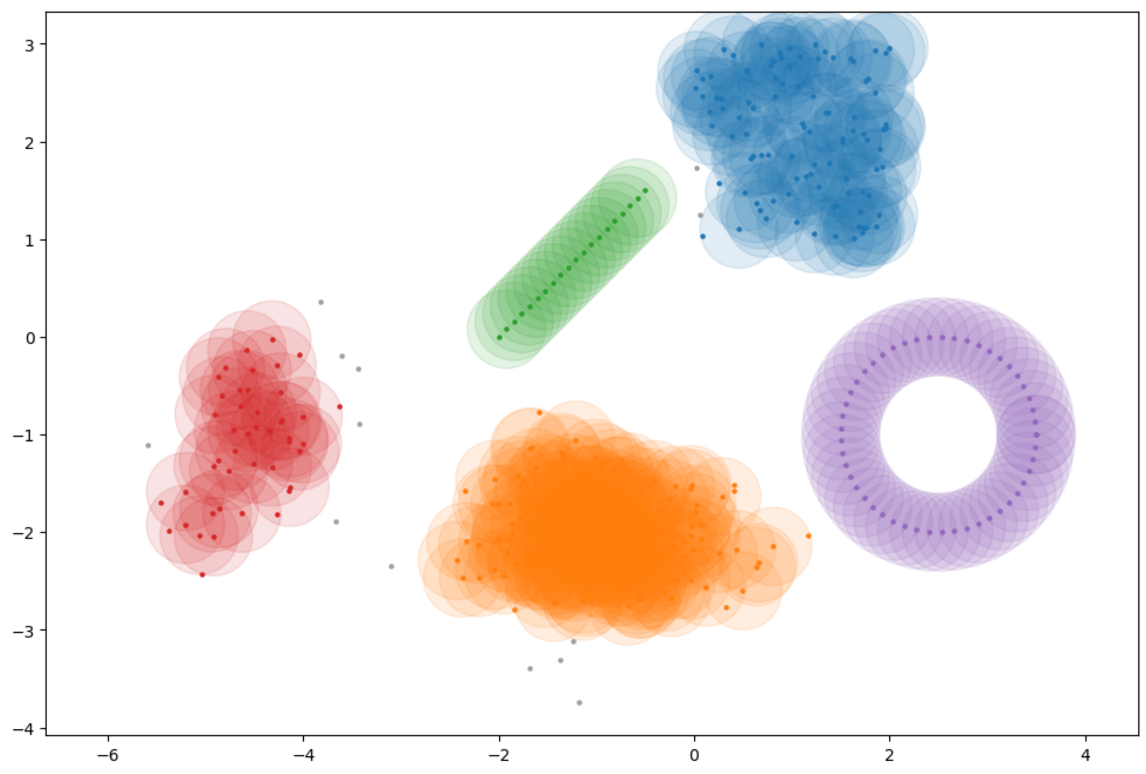
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import sklearn.cluster as cluster

rng = np.random.default_rng(0)
```

```
# 5 clusters in 2 dimensions

n1, n2, n3, n4, n5 = 100, 200, 20, 50, 50
n = n1 + n2 + n3 + n4 + n5

X1 = rng.uniform((0, 1), (2, 3), (n1, 2))
X2 = rng.multivariate_normal((-1, -2), ((0.5, 0), (0, 0.2)), n2)
X3 = np.linspace(-1, 0.5, n3).reshape(-1, 1) * np.ones((1, 2)) + np.array([[-1,␣
↪1]])
X4 = rng.multivariate_normal((-4.5, -1), ((0.2, 0.1), (0.1, 0.4)), n4)
phi = np.linspace(0, 2 * np.pi, n5).reshape(-1, 1)
X5 = np.array([[2.5, -1]]) + np.concatenate((np.cos(phi), np.sin(phi)), axis=1)

X = np.concatenate((X1, X2, X3, X4, X5))

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c='b')
ax.axis('equal')
plt.show()
```

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



```
eps = 0.4
N = 5

dbs = cluster.DBSCAN(eps=eps, min_samples=N)
dbs.fit(X)

colors = list(mpl.colors.TABLEAU_COLORS.values())

fig, ax = plt.subplots(figsize=(12, 8))

# plot neighborhoods of core points
for i in dbs.core_sample_indices_:
    color = colors[dbs.labels_[i] % len(colors)] + '20'
    ax.add_artist(mpl.patches.Circle(X[i, :], eps, color=color))

# plot points
for label in np.unique(dbs.labels_):
    if label == -1:
        color = '#A0A0A0'
    else:
        color = colors[label % len(colors)]
    mask = dbs.labels_ == label
    ax.scatter(X[mask, 0], X[mask, 1], c=color, s=5)

ax.axis('equal')
plt.show()
```



5.2 OPTICS

OPTICS (ordering points to identify the clustering structure) is an extension of DBSCAN, which automatically chooses ε and allows for cluster-dependent ε. Instead of ε OPTICS requires a parameter $\varepsilon_{\max}$ describing the maximum value to consider when choosing ε. This parameter has to be chosen large enough to not miss a cluster but there is no upper bound. The larger $\varepsilon_{\max}$ the more computation time is required while clustering results remain the same. Thus, choosing a good $\varepsilon_{\max}$ is only important for large scale data sets.

OPTICS differs from DBSCAN as follows:

- Neighborhood radius ε depends on the point x under consideration. It is chosen as small as possible but large enough to have N points (including x) in the $\varepsilon(x)$-neighborhood. In this sense every point is a core point, at least if $\varepsilon_{\max}$ is large enough (which we assume below).
- The direct output of the OPTICS algorithm is not a clustering. Instead, to each processed point a numerical value, the *reachability distance*, is assigned. These values imply a linear ordering of all processed points. From this ordering a clustering can be deduced by several different methods (see below).

5.2.1 Basic Algorithm

During runtime of OPTICS algorithm each point is in one of three states: untouched or discovered or processed. At the beginning all points are untouched, at the end all points will be processed (if $\varepsilon_{\max}$ is large enough).

OPTICS, like DBSCAN, starts at an arbitrary point x. For each point $\tilde{x}$ in the $\varepsilon_{\max}$-neighborhood of x then the reachability distance is computed as

$$\max\{\varepsilon(x), |x-\tilde{x}|\},$$

all points but x are appended to the list of discovered (but unprocessed) points and x is marked as processed. Now the following steps are repeated until no discovered but unprocessed points remain:

- The point with lowest reachability distance is chosen from the list of discovered points and marked as processed.
- For all unprocessed points in its $\varepsilon_{\max}$-neighborhood reachability distances with respect to the point are computed.

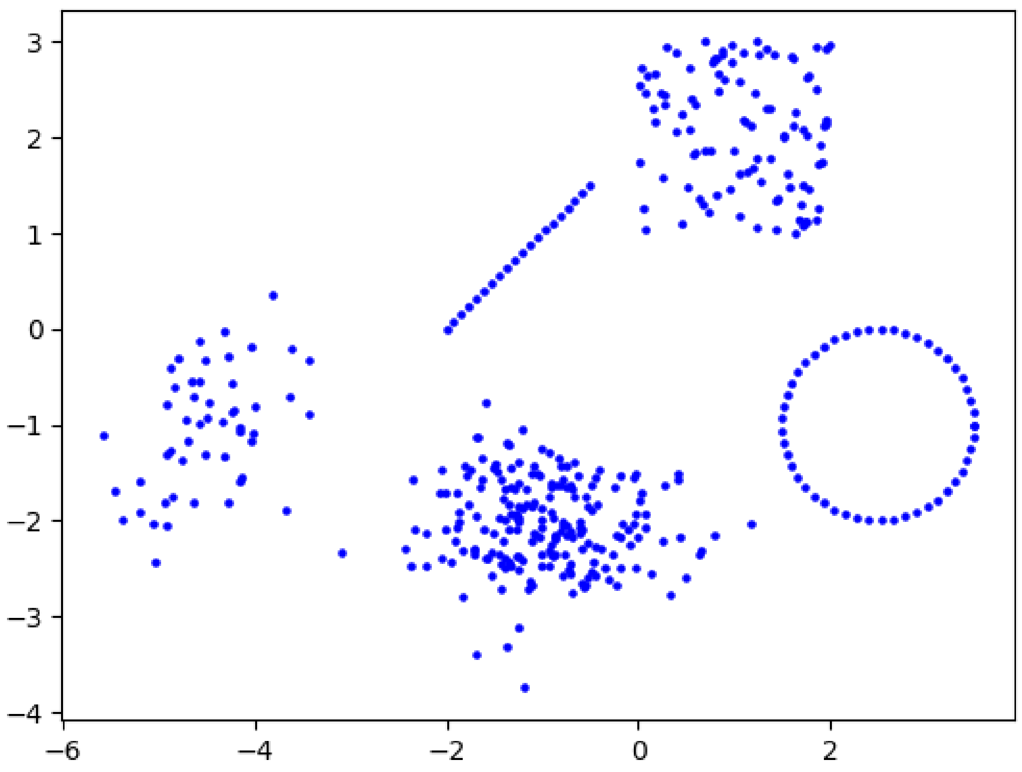
- If such an unprocessed neighboring point already has been discovered before, the smaller one of old and new reachability distance is chosen. If a neighboring point has been discovered for the first time, it is added to the list of discovered points.

5.2.2 Implementation from Scratch

Note that efficient implementation of OPTICS algorithm requires knowledge of advanced data structures. For understanding the basic principles here we only provide an inefficient implementation avoiding any use of advanced data structures. Further we set $\varepsilon_{\max} = \infty$, that is, instead of (large) neighborhoods we always discover the whole data set. Consequently, there will be no untouched points after processing the starting point.

```python
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(0)
```

```python
# 5 clusters in 2 dimensions

n1, n2, n3, n4, n5 = 100, 200, 20, 50, 50
n = n1 + n2 + n3 + n4 + n5

X1 = rng.uniform((0, 1), (2, 3), (n1, 2))
X2 = rng.multivariate_normal((-1, -2), ((0.5, 0), (0, 0.2)), n2)
X3 = np.linspace(-1, 0.5, n3).reshape(-1, 1) * np.ones((1, 2)) + np.array([[-1,␣
↪1]])
X4 = rng.multivariate_normal((-4.5, -1), ((0.2, 0.1), (0.1, 0.4)), n4)
phi = np.linspace(0, 2 * np.pi, n5).reshape(-1, 1)
X5 = np.array([[2.5, -1]]) + np.concatenate((np.cos(phi), np.sin(phi)), axis=1)

X = np.concatenate((X1, X2, X3, X4, X5))

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c='b')
ax.axis('equal')
plt.show()
```



```
N = 10
start_idx = 0

n = X.shape[0]

dists = np.empty(n)     # reachability distances
epss = np.empty(n)     # epsilons
processed = np.full(n, False)     # point already processed?
order = np.empty(n, dtype=int)     # indices in the order of processing (for␣
↪identifying clusters)
previous = np.empty(n, dtype=int)     # index of core point with shortest distance␣
↪(for visualization only)

# calculate pairwise distances
D = np.empty((n, n))
for l in range(0, n):
    D[l, l] = 0
    for ll in range(0, l):
        D[l, ll] = np.sqrt(np.sum((X[l, :] - X[ll, :]) ** 2))
        D[ll, l] = D[l, ll]

# calculate epsilon for each point
for l in range(0, n):
    epss[l] = np.sort(D[l, :])[N]

# calculate reachability distances with respect to starting point
x = X[start_idx, :]
for l in range(0, n):
    dists[l] = np.maximum(epss[start_idx], D[start_idx, l])
    previous[l] = start_idx
processed[start_idx] = True
order[0] = start_idx
previous[start_idx] = -1     # no predecessor
```

(continues on next page)

(continued from previous page)

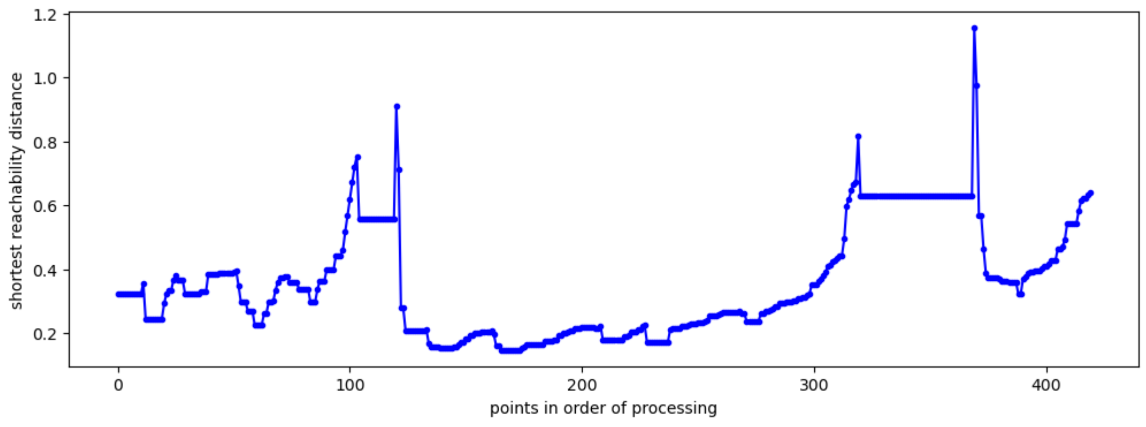
```python
# process all points
for i in range(1, n):

    # get next core point (the one closest to set of processed points)
    dists_sort = np.argsort(dists)
    idx_sorted = np.flatnonzero(processed[dists_sort] == False)[0]
    idx = dists_sort[idx_sorted]
    processed[idx] = True
    order[i] = idx

    # update reachability distances
    for l in np.flatnonzero(processed == False):
        new_dist = np.maximum(epss[idx], D[idx, l])
        if new_dist < dists[l]:
            dists[l] = new_dist
            previous[l] = idx
```

5.2.3 Identifying Clusters

When OPTICS algorithm has finished we have the processing order and shortest reachability distances (SRD) of all points. The SRD of a point tells us how far away the point was from the already processed points before it got processed itself. As long as SRDs remain small while processing point by point, new points are close to previous ones. If SRD suddenly increases, then there are no more close points left. Thus, all points of a cluster have been processed and the processing of a new cluster starts.

If we plot SRD versus position in the processed chain of points, we may identify valleys as clusters. To extract a clustering from OPTICS results we have to identify valleys in the plot. Points with high SRD (mountains between valleys) can be interpreted as outliers.

```python
fig, ax = plt.subplots(figsize=(12, 4))
ax.plot(dists[order] , '-ob', markersize=3)
ax.set_xlabel('points in order of processing')
ax.set_ylabel('shortest reachability distance')
plt.show()
```

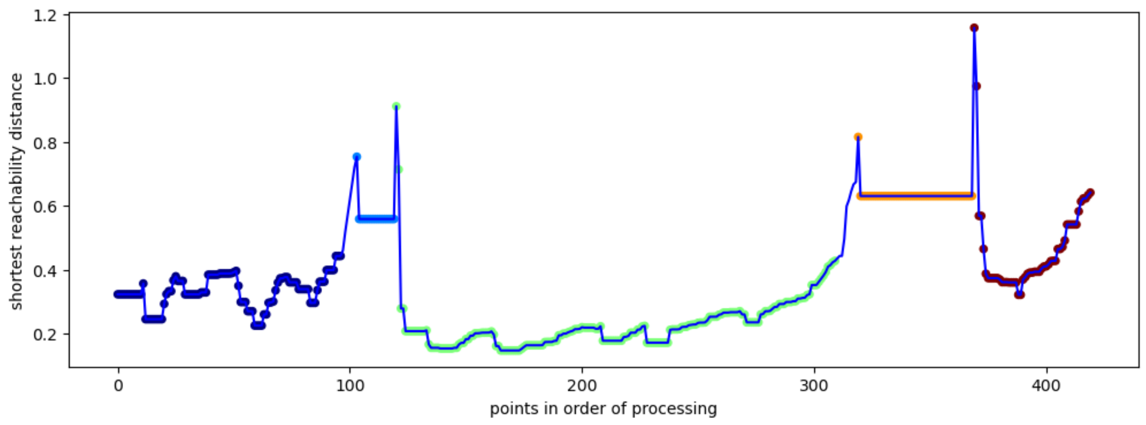


There exist several algorithms for finding valleys in the plot. One could look for steep decents or ascents or one could look for almost constant regions. Depending on the sensitivity of this valley search more of less clusters will be found. With different levels of sensitivity we would obtain a sequence of nested clusterings. In this sense, OPTICS algorithm is sometimes classified as hierarchical clustering method.

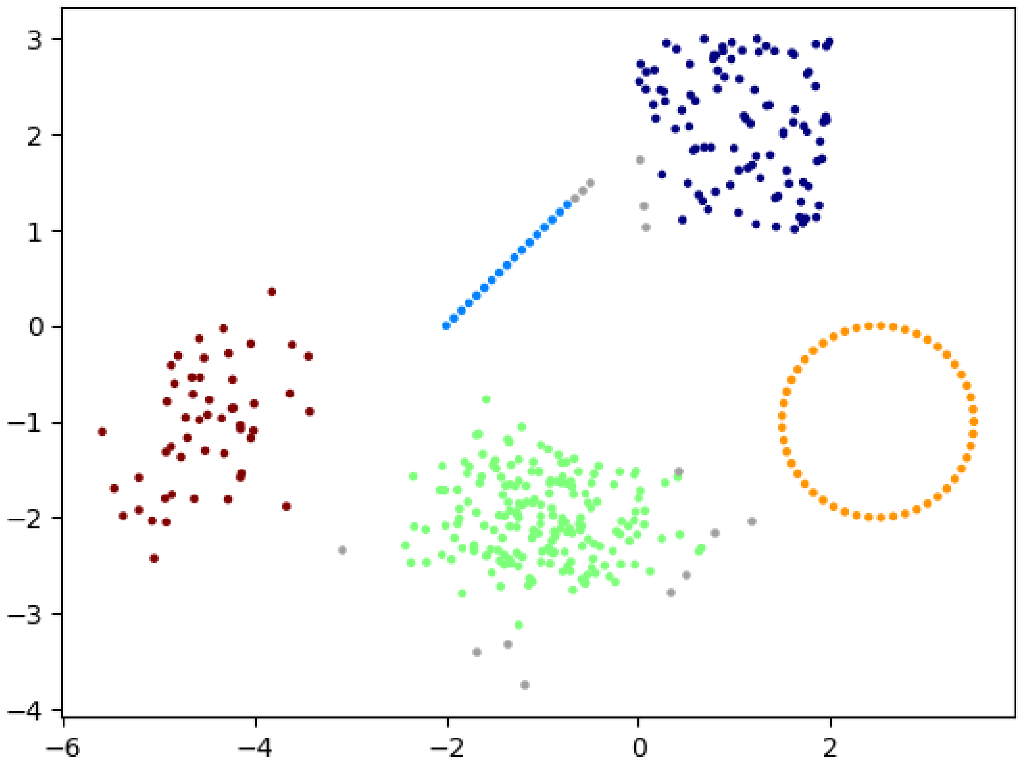
Here we do not go into the details of automatically identifying clusters.

```
labels = np.full(n, -1, dtype=int)

labels[order[0:97]] = 0
labels[order[103:120]] = 1
labels[order[120:310]] = 2
labels[order[319:369]] = 3
labels[order[369:]] = 4
```

```
mask = labels != -1

fig, ax = plt.subplots(figsize=(12, 4))
ax.plot(dists[order] , '-b', markersize=3)
ax.scatter(np.arange(0, n)[mask[order]], dists[order][mask[order]],
           s=20, c=labels[order][mask[order]], cmap='jet')
ax.set_xlabel('points in order of processing')
ax.set_ylabel('shortest reachability distance')
plt.show()
```



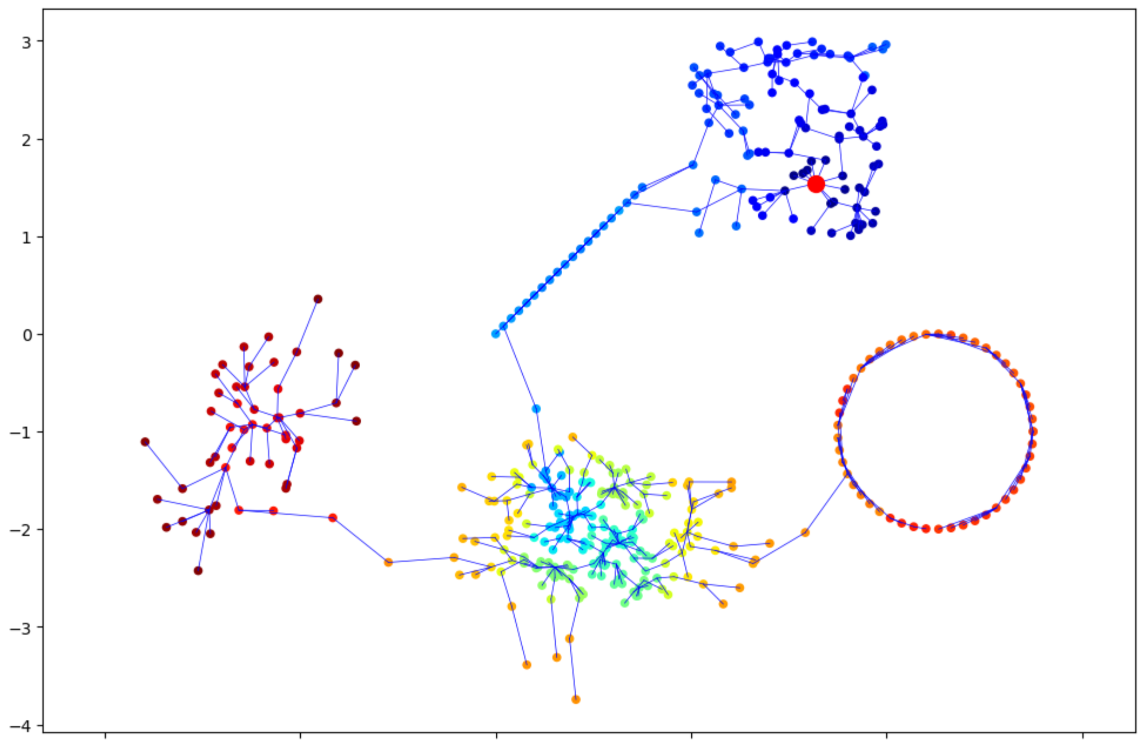
```
fig, ax = plt.subplots()
mask = labels != -1
ax.scatter(X[mask, 0], X[mask, 1], s=5, c=labels[mask], cmap='jet')
ax.scatter(X[mask == False, 0], X[mask == False, 1], s=5, c='#A0A0A0')
ax.axis('equal')
plt.show()
```



5.2.4 Visualizing Processing Order and Reachability

For getting some more insight into OPTICS algorithm we may visualize the processing order. On the one hand we could color each point by its position in the chain of processed points. On the other hand we could connect each point with its closest core point, that is, with the point used in the computation of the shortest reachability distance.

```
fig, ax = plt.subplots(figsize=(12, 8))

# connections
for l in range(0, n):
    if previous[l] != -1:
        ax.plot([X[previous[l], 0], X[l, 0]], [X[previous[l], 1], X[l, 1]], '-b',␣
↪markersize=3, linewidth=0.5)

# colored points
c = np.empty(n)
c[order] = np.arange(0, n)
ax.scatter(X[:, 0], X[:, 1], s=20, c=c, cmap='jet')

# starting point
ax.plot(X[start_idx, 0], X[start_idx, 1],'-or', markersize=10)

ax.axis('equal')
plt.show()
```



All connections together constitute a graph which is tree shaped and spans the data set, a so called *spanning tree*. This graph may be used for further investigation of the data set, for instance, to visualize high dimensional data sets in two dimensions without loosing too much structure.

Unfortunately there seems to be no Python package supporting tree visualization with prescribed edge lengths and having a simple API. Thus, we write our own tree visualization function.

```
def plot_tree(prevs, dists, core_dists, ax, root_x=0, root_y=0):
    ''' Plot tree with prescribed edge lengths.
    Only plots the edges, not the nodes. Node positions are
    returnd as NumPy array of shape (n_samples, 2)

    prevs ... 1d array with predecessor (parent) for each node (root has -1)
    dists ... distance to predecessor
    core_dists ... distance considered as close to the node (close leaves will
                   be visualized closer than prescribed by dists)
    '''

    # for each node create list of children
    children = [[] for i in range(0, n)]
    for i in range(0, n):
        if prevs[i] != -1:
            children[prevs[i]].append(i)
        else:
            root = i

    # randomly permutate children of each node (for better visual impression)
    for i in range(0, n):
        if len(children[i]) > 0:
            children[i] = list(rng.permuted(children[i]))

    # array for node positions
    pos = np.empty((n, 2))

    # place root on stack of nodes to process
```

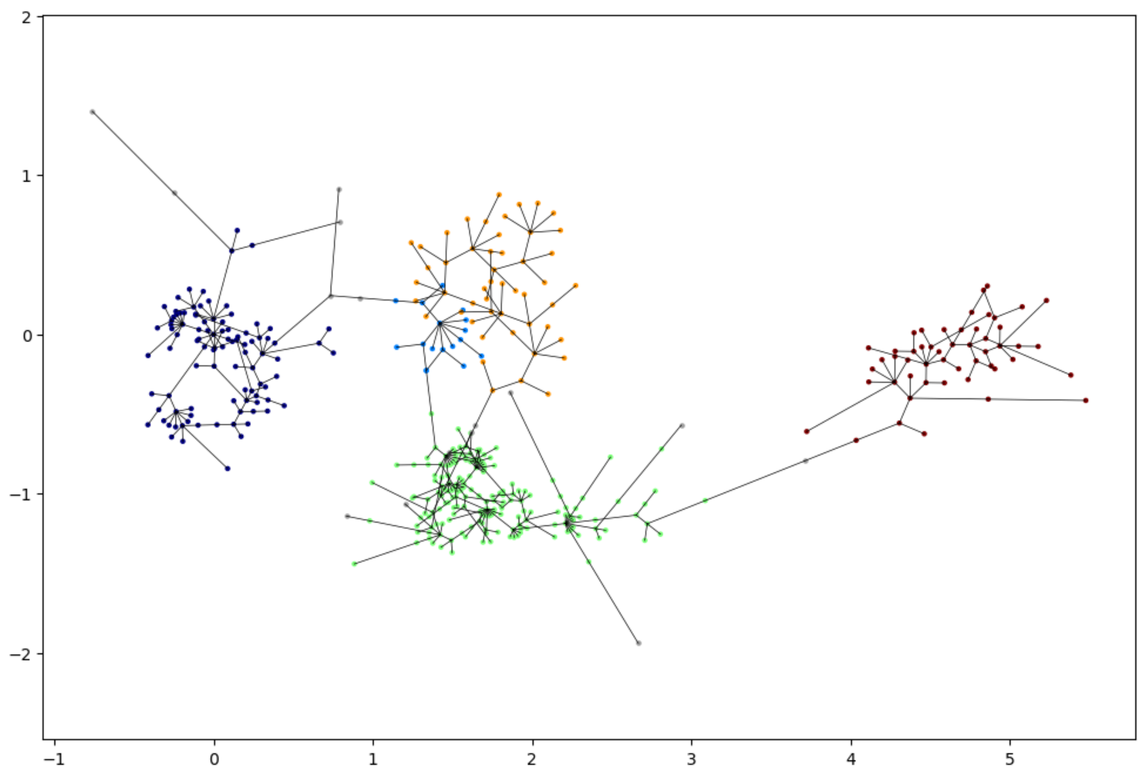
(continues on next page)

(continued from previous page)

```
    pos[root, 0] = 0
    pos[root, 1] = 0
    stack = [root]

    # process nodes until stack is empty
    while len(stack) > 0:
        node = stack.pop()
        for i, child in enumerate(children[node]):

            # radius (distance to parent node)
            r = dists[child]
            if r <= core_dists[node]:
                r = 0.3 * core_dists[node]

            # angle
            if node == root:
                # evenly distribute children around root
                angle = 2 * np.pi / len(children[node]) * (i + 0.5)
            else:
                # evenly distribute children in a 180° region around node,
                # rotate 180° region to look away from parent of current node
                parent_angle = np.arctan2(pos[node, 1] - pos[prevs[node], 1],␣
↪pos[node, 0] - pos[prevs[node], 0])
                angle = np.pi / len(children[node]) * (i + 0.5) - 0.5 * np.pi +␣
↪parent_angle

            # child position
            pos[child, 0] = pos[node, 0] + r * np.cos(angle)
            pos[child, 1] = pos[node, 1] + r * np.sin(angle)
            stack.append(child)

            # plot edge to child
            ax.plot([pos[node, 0], pos[child, 0]], [pos[node, 1], pos[child, 1]],
↪'-k', linewidth=0.5)

    return pos
```

At the moment we only have 2d data. Thus, direct visualization of clusters is possible. Nethertheless we should have a look at corresponding tree visualization to get an idea of how to interpret it.

```
fig, ax = plt.subplots(figsize=(12, 8))

pos = plot_tree(previous, dists, epss, ax)

mask = labels != -1
ax.scatter(pos[mask, 0], pos[mask, 1], s=5, c=labels[mask], cmap='jet')
ax.scatter(pos[mask == False, 0], pos[mask == False, 1], s=5, c='#A0A0A0')

ax.axis('equal')
plt.show()
```



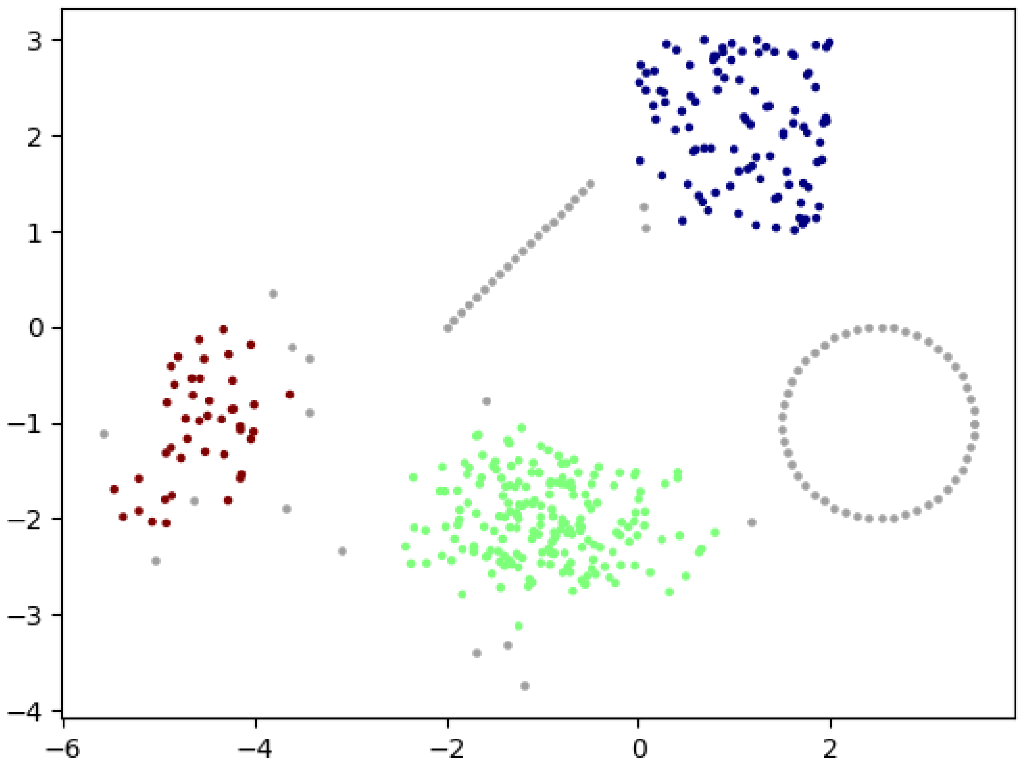
5.2.5 OPTICS Algorithm with Scikit-Learn

Scikit-Learn provides the `OPTICS`[27] class. The only relevant parameter is `min_samples` (our N). There are two cluster identification routines available. One based on DBSCAN (with sensitivity parameter `eps`) and another one with sensitivity parameter `xi`. Note, that automatic cluster identification often yields poor results.

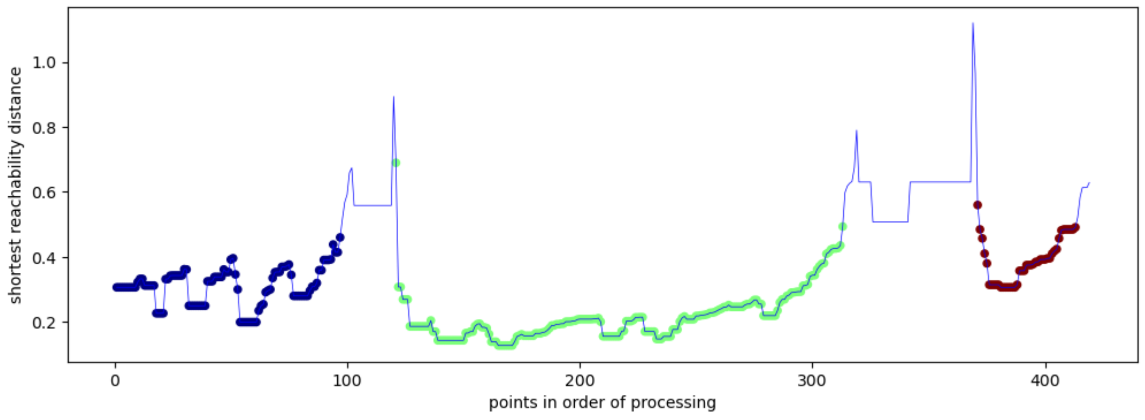
```
import sklearn.cluster as cluster
```

```
opt = cluster.OPTICS(min_samples=10, cluster_method='dbscan', eps=0.5)
opt.fit(X)

fig, ax = plt.subplots()
mask = opt.labels_ != -1
ax.scatter(X[mask, 0], X[mask, 1], s=5, c=opt.labels_[mask], cmap='jet')
ax.scatter(X[mask == False, 0], X[mask == False, 1], s=5, c='#A0A0A0')
ax.axis('equal')
plt.show()
```

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



```
fig, ax = plt.subplots(figsize=(12, 4))
ax.plot(opt.reachability_[opt.ordering_] , '-b', linewidth=0.5)
ax.scatter(np.arange(0, n)[mask[opt.ordering_]], opt.reachability_[opt.ordering_
↪][mask[opt.ordering_]], s=20, c=opt.labels_[opt.ordering_][mask[opt.ordering_]],
↪ cmap='jet')
ax.set_xlabel('points in order of processing')
ax.set_ylabel('shortest reachability distance')
plt.show()
```



CHAPTER

SIX

DISTRIBUTION-BASED CLUSTERING

In distribution-based clustering each cluster $1, \ldots, k$ is represented by a probability distribution. The underlying assumption is that there exist k probability distributions $p_1, \ldots, p_k$ from which each cluster's data points have been drawn. Given training data one aims to find the probability distribution for each cluster. Usually Gaussian distributions are used, but others may be appropriate as well.

The difficulty is to derive (for fixed k) the distribution parameters from the data set and, thus, cluster positions, sizes, shapes. Resulting model of the data set is denoted as a *mixture model* because the data set is represented as a mixture of points drawn from different distributions. In case of Gaussian distributions the searched for parameters are the k mean vectors and the k covariance matrices. Resulting model then is a *Gaussian mixture model*.

Mixture models are an example of *generative models*, because they allow to generate new samples with similar properties like samples in the training set. Here we do not obtain a model of the training set's clusters themselves, but a model to generate data sets similar to the training set. From this generative model we may extract information about clusters in the training data (distribution parameters), but partitioning of training samples into concrete clusters is an extra step based on information extracted from the model.

Related projects:

- *Generating Handwritten Digits* (page 119)

6.1 Expectation Maximization

On the one hand we want to find parameters of the k underlying distributions such that data fits well to the distributions. On the other hand we want to label training samples (assign them to clusters). The questions are how to formalize 'fits well' and how to derive the labels. A common approach is to choose parameters and labels such that the training set is the most probable data set within all data sets which can be drawn from distributions with the chosen parameters respecting the cluster assignments. This approach (and its algorithmic realization below) is known as *expectation maximization (EM)*.

6.1.1 Notation

The EM approach requires that we consider training samples $x_1, \ldots, x_n$ and labels as realizations of random variables $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$, respectively. The X_l are continuous random variables with values in $\mathbb{R}^m$. The Y_l are discrete random variable taking values in $\{1, \ldots, k\}$.

Clusters $1, \ldots, k$ are described by probability densities $p_i : \mathbb{R}^m \to [0, \infty)$, $i = 1, \ldots, k$ containing parameters $\vartheta_1, \ldots, \vartheta_k$, respectively. Here, ϑ_i is a vector if the ith density has more than one parameter. If we know the cluster of some sample l, that is, $Y_l = y_l$ for some $y_l \in \{1, \ldots, k\}$, then we know the distribution of the lth sample: it's p_{y_l}. In other words, the densities p_i are our model for describing generation of random samples given preset cluster assignments.

Note that for continuous random variables working with probability densities is much more comfortable than working with the probability measures directly, because $P(X = x) = 0$ for each x, whereas $p(x)$ carries the relevant information as long os p is continuous, which is usually taken for granted when working with densities without explicitly mentioning this important assumption.

For the Y_l we do not specify a model. The Y_l are discrete random variables taking k different values. Thus, the distribution of each Y_l can be described by k non-negative numbers:

$$q_{l,1} := P(Y_l = 1), \quad \ldots, \quad q_{l,k} := P(Y_l = k) \qquad \text{for } l = 1, \ldots, n.$$

The $q_{l,i}$ describe the generation of random cluster assignments for all samples. By definition they satisfy

$$\sum_{i=1}^{k} q_{l,i} = 1 \qquad \text{for } l = 1, \ldots, n.$$

6.1.2 Maximization Problem

The aim is to choose all parameters describing the generation of random samples (that is, the ϑ_i and the $q_{l,i}$) in such a way that the probability to observe the available training data becomes maximal with respect to all possible parameter values:

$$p(x_1, \ldots, x_n) \to \max_{\vartheta_1,\ldots,\vartheta_k,q_{1,1},\ldots,q_{n,k}}.$$

Here, p is the probability density function of $(X_1, \ldots, X_n)$.

The structure of p is quite involved and corresponding maximization problem hard to solve, analytically as well as numerically.

6.1.3 Idea of EM Algorithm

Solving the maximization problem with respect to all parameters is almost impossible. But if we fix the $q_{l,i}$ to certain special but still reasonable values then optimal $\vartheta_1, \ldots, \vartheta_k$ can be obtained in a straight-forward way. The other way round, if we fix $\vartheta_1, \ldots, \vartheta_k$, then calculating optimal $q_{l,i}$ becomes viable.

We may alternate both steps: Fix (initial, almost random) $q_{l,i}$ and get optimal ϑ_i. Then for those ϑ_i get optimal $q_{l,i}$ and so on until some stopping criterion is satisfied.

6.1.4 Optimal Labels

Given some fixed values for $\vartheta_1, \ldots, \vartheta_k$ we want so solve above maximization problem with respect to the $q_{l,i}$.

The law of total probability allows to rewrite p as

$$p(x_1, \ldots, x_n) = \sum_{y_1=1}^{k} \cdots \sum_{y_n=1}^{k} P(y_1, \ldots, y_n)\, p(x_1, \ldots, x_n \,|\, y_1, \ldots, y_n),$$

where $p(x_1, \ldots, x_n \,|\, y_1, \ldots, y_n)$ is the conditional probability density for $(X_1, \ldots, X_n)$ given conrete values $y_1, \ldots, y_n$ for $Y_1, \ldots, Y_n$.

Mathematical side note

The somewhat unusual mix of densities and probability measure P is mathematically correct. This can be seen by considering a subset A of $\mathbb{R}^m$ instead of only one element, allowing to write everything without densities:

$$P(A) = \sum \cdots \sum P(y_1, \ldots, y_n)\, P(A \,|\, y_1, \ldots, y_n).$$

Replacing P by integrals over corresponding densities we obtain

$$\begin{aligned}\int_A p(x_1, \ldots, x_n)\, \mathrm{d}(x_1, \ldots, x_n) &= \sum \cdots \sum P(y_1, \ldots, y_n) \int_A p(x_1, \ldots, x_n \,|\, y_1, \ldots, y_n)\, \mathrm{d}(x_1, \ldots, x_n) \\ &= \int_A \sum \cdots \sum P(y_1, \ldots, y_n)\, p(x_1, \ldots, x_n \,|\, y_1, \ldots, y_n)\, \mathrm{d}(x_1, \ldots, x_n).\end{aligned}$$

This integral equation holds for all sets A. Thus, integrants on both sides are equal.

Remember

$$P(y_1,\dots,y_n)=q_{1,y_1}\cdots q_{n,y_n}$$

by definition of the $q_{l,i}$. The conditional density may be rewritten in terms of conditional densities for each sample (we reuse p here although joint and per-sample densities are different functions):

$$p(x_1,\dots,x_n\,|\,y_1,\dots,y_n)=p(x_1\,|\,y_1,\dots,y_n)\cdots p(x_n\,|\,y_1,\dots,y_n).$$

The distribution of X_l only depends on Y_l, not on Y_λ for $\lambda\neq l$. Thus,

$$p(x_l\,|\,y_1,\dots,y_n)=p(x_l\,|\,y_l).$$

The value $p(x_l\,|\,y_l)$ is the probability of observing x_l if we know that x_l belongs to cluster y_l. With the cluster densities $p_1,\dots,p_k$ we thus have

$$p(x_l\,|\,y_1,\dots,y_n)=p_{y_l}(x_l)$$

The maximization problem to solve now reads

$$\sum_{y_1=1}^{k}\cdots\sum_{y_n=1}^{k}q_{1,y_1}\cdots q_{n,y_n}\,p_{y_1}(x_1)\cdots p_{y_n}(x_n)\to\max_{q_{1,1},\dots,q_{n,k}}.$$

From the properties of the $q_{l,i}$ (non-negative, sum over i is 1) we immediately see that $q_{1,y_1}\cdots q_{n,y_n}\geq 0$ for each $(y_1,\dots,y_n)$ and

$$\sum_{y_1=1}^{k}\cdots\sum_{y_n=1}^{k}q_{1,y_1}\cdots q_{n,y_n}=1.$$

Thus, our maximization problem has the structure

$$\sum_j a_j\,z_j\to\max_{a_j},\qquad a_j\geq 0,\qquad \sum_j a_j=1.$$

The solution of such problems is known (and easily verified) to be $a=(0,\dots,0,1,0,\dots,0)$ with the 1 at the position of the maximal z_j.

In our setting: Let $y_1^*,\dots,y_n^*$ be the set of labels maximizing $p_{y_1}(x_1)\cdots p_{y_n}(x_n)$. Then $q_{1,y_1^*}\cdots q_{n,y_n^*}$ has to be 1 (which is only possible if all q_{l,y_l^*} equal 1) and all other $q_{l,i}$ have to be 0. In other words, the optimal distribution of $(Y_1,\dots,Y_n)$ is deterministic with the whole probability mass on $(Y_1,\dots,Y_n)=(y_1^*,\dots,y_n^*)$. Note that $p_{y_1}(x_1)\cdots p_{y_n}(x_n)$ becomes maximal if each factor becomes maximal. Thus,

$$y_l^*=\operatorname{argmax}_i p_i(x_l)\qquad\text{for } l=1,\dots,n.$$

6.1.5 Optimal Per-Cluster Distributions

Given fixed $q_{l,i}$ we want to solve the above maximization problem with respect to the parameters $\vartheta_1,\dots,\vartheta_k$ of the per-cluster probability densities $p_1,\dots,p_k$. We do not solve the maximization problem for arbitraty $q_{l,i}$ but for $q_{l,i}$ with the structure obtained as optimal solution in the previous subsection. That is, the $q_{l,i}$ describe a deterministic probability distribution with all mass on some fixed $(y_1,\dots,y_n)$.

Given the concrete set of labels $y_1,\dots,y_n$ the probability to observe our training samples $x_1,\dots,x_n$ (that is, the objective of the maximization problem) is

$$p(x_1,\dots,x_n)=\prod_{l=1}^{n}p_{y_l}(x_l)=\prod_{i=1}^{k}\prod_{l:y_l=i}p_i(x_l).$$

Each factor $\prod_{l:y_l=i}p_i(x_l)$ only depends on ϑ_i and not on ϑ_j for $j\neq i$. Thus, we may maximize factors independently.

Finding ϑ_i which maximizes $\prod_{l:y_l=i}p_i(x_l)$ is a standard task in maximum likelihood estimation[28]. Depending on the structure of p_i there exist explicit solutions (see below for Gaussion densities).

[28] https://en.wikipedia.org/wiki/Maximum_likelihood_estimation

6.2 The EM Algorithm

The procedure derived above can be summarized as follows:

1. Randomly choose initial labels $y_1, \ldots, y_n$.
2. Calculate optimal $\vartheta_1, \ldots, \vartheta_k$ based on the training samples in each cluster.
3. Update labels by assigning each x_l the cluster i with the highest $p_i(x_l)$.
4. Go to 2 if stopping criterion not satisfied.

Common stopping criteria are to stop if the y_l or the ϑ_i do not change anymore.

The process is known to converge to a stationary point (gradient is zero) of the original maximization problem. Thus, it may get stuck in local maxima or even in saddle points.

6.3 Gaussian Mixtures

If $p_1, \ldots, p_k$ are Gaussian probability densities, p_i has the mean vector and the covariance matrix as parameters. The products

$$\prod_{l:y_l=i} p_i(x_l), \qquad i = 1, \ldots, k$$

in the above derivation are maximized with respect to the distribution parameters, if we choose mean and covariance of

$$\{x_l : y_l = i\} \qquad \text{(cluster } i\text{)}$$

as parameters.

EM with Gaussian distributions is a very fast algorithm. The two alternating steps boil down to:

- calculate probabilities $p_{l,i} := p_i(x_l)$ for all pairs of samples and clusters,
- for each l set $y_l := \operatorname{argmax}_{i=1,\ldots,k} p_{l,i}$,
- calculate empirical mean vectors and empirical covariance matrices for each cluster.

6.4 Relation to k-Means

The idea of alternating two steps, each solving the problem partially, is very similar to k-means:

- EM: for fixed labels get optimal parameters.k-means: for fixed labels get centroid of each cluster.
- EM: for fixed parameters get optimal labels.k-means: for fixed centroids get optimal labels.

One easily verifies that the training of Gaussian mixture models with identity covariance matrices is equivalent to k-means clustering.

6.5 Choosing k

Like for k-means the number of clusters has to be known in advance. The larger k, the better the data can be represented by the model, but interpretation of clusters becomes more difficult (overfitting).

Silhouette score and Davies-Bouldin index may help choosing k. For elbow method one can use the optimal value of the objective function from the above maximization problem. Another approach is known as *Bayesian information criterion*. Here one chooses k to minimize

$$t \log n - \log p(x_1, \ldots, x_n).$$

The second summand is the negative logarithm of the optimal value of the original maximization problem and the first summand is a penalty containing the number t of model parameters. Obviously, t is a multiple of k if all k distributions are of equal type (e.g., Gaussian). The more clusters the model has, the higher t and the larger the penalty. Minimizing the sum yields a value for k which is a compromise between number of clusters and fitting error. For more detailed motivation of BIC see Wikipedia on BIC[29].

6.6 Soft Clustering

EM allows for soft clustering, that is, each sample we may assign probabilities to belong to the different clusters. The $p_i(x_l)$ can be interpreted as score for x_l to belong to cluster i. Probability-like scores can be obtained the usual way as

$$\frac{p_i(x_l)}{p_1(x_l)+\cdots+p_k(x_l)}.$$

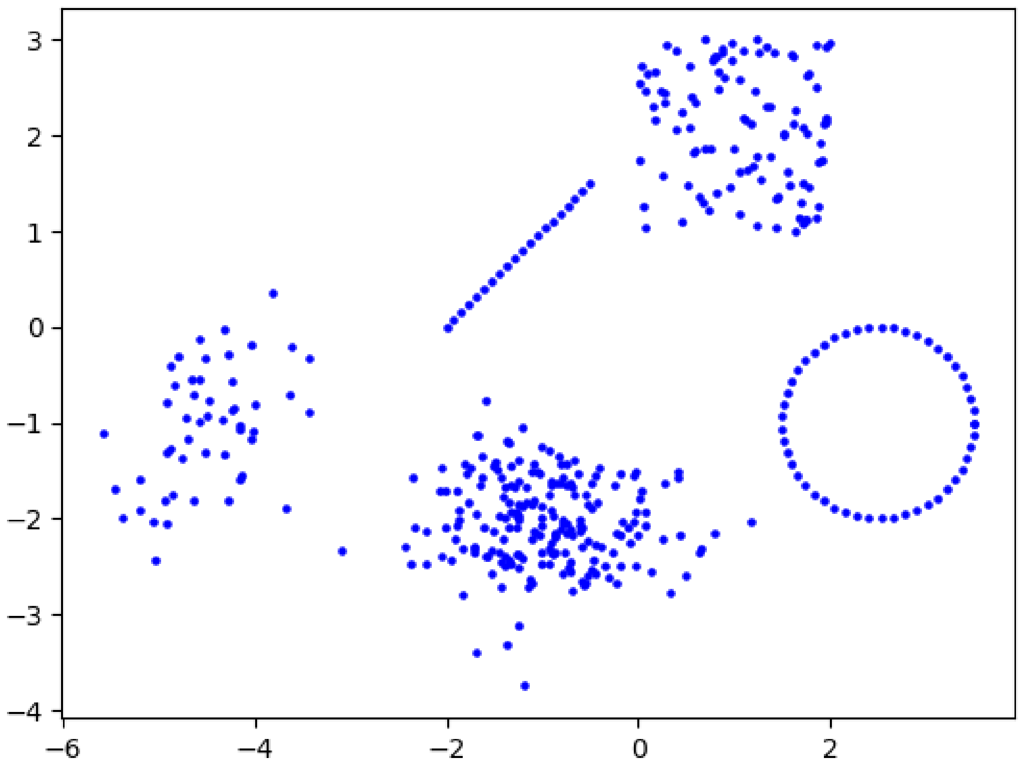
6.7 Gaussian Mixture Models with Scikit-Learn

Scikit-Learn provides the GaussianMixture[30] class in its `mixture` module. The only relevant parameter is `n_components` (our k).

```python
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import sklearn.mixture as mixture
import sklearn.metrics as metrics

rng = np.random.default_rng(0)
```

```python
# 5 clusters in 2 dimensions

n1, n2, n3, n4, n5 = 100, 200, 20, 50, 50
n = n1 + n2 + n3 + n4 + n5

X1 = rng.uniform((0, 1), (2, 3), (n1, 2))
X2 = rng.multivariate_normal((-1, -2), ((0.5, 0), (0, 0.2)), n2)
X3 = np.linspace(-1, 0.5, n3).reshape(-1, 1) * np.ones((1, 2)) + np.array([[-1,␣
↪1]])
X4 = rng.multivariate_normal((-4.5, -1), ((0.2, 0.1), (0.1, 0.4)), n4)
phi = np.linspace(0, 2 * np.pi, n5).reshape(-1, 1)
X5 = np.array([[2.5, -1]]) + np.concatenate((np.cos(phi), np.sin(phi)), axis=1)

X = np.concatenate((X1, X2, X3, X4, X5))

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], s=5, c='b')
ax.axis('equal')
plt.show()
```

[29] https://en.wikipedia.org/wiki/Bayesian_information_criterion
[30] https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html



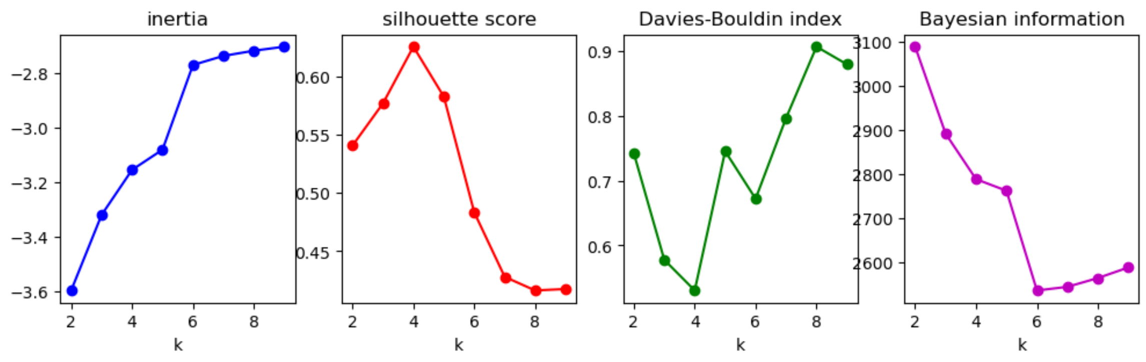
```
ks = range(2, 10)

obj = []
sil = []
db = []
bic = []
for k in ks:

    print(k)

    gm = mixture.GaussianMixture(n_components=k)
    gm.fit(X)

    labels = gm.predict(X)
    obj.append(gm.lower_bound_)
    sil.append(metrics.silhouette_score(X, labels))
    db.append(metrics.davies_bouldin_score(X, labels))
    bic.append(gm.bic(X))

obj = np.array(obj)
sil = np.array(sil)
db = np.array(db)
bic = np.array(bic)

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(12, 3))
ax1.plot(ks, obj, '-ob')
ax2.plot(ks, sil, '-or')
ax3.plot(ks, db, '-og')
ax4.plot(ks, bic, '-om')
ax1.set_xlabel('k')
ax2.set_xlabel('k')
ax3.set_xlabel('k')
ax4.set_xlabel('k')
ax1.set_title('inertia')
```

(continues on next page)

(continued from previous page)

```
ax2.set_title('silhouette score')
ax3.set_title('Davies-Bouldin index')
ax4.set_title('Bayesian information')
plt.show()
```

```
2
3
4
5
6
7
8
9
```



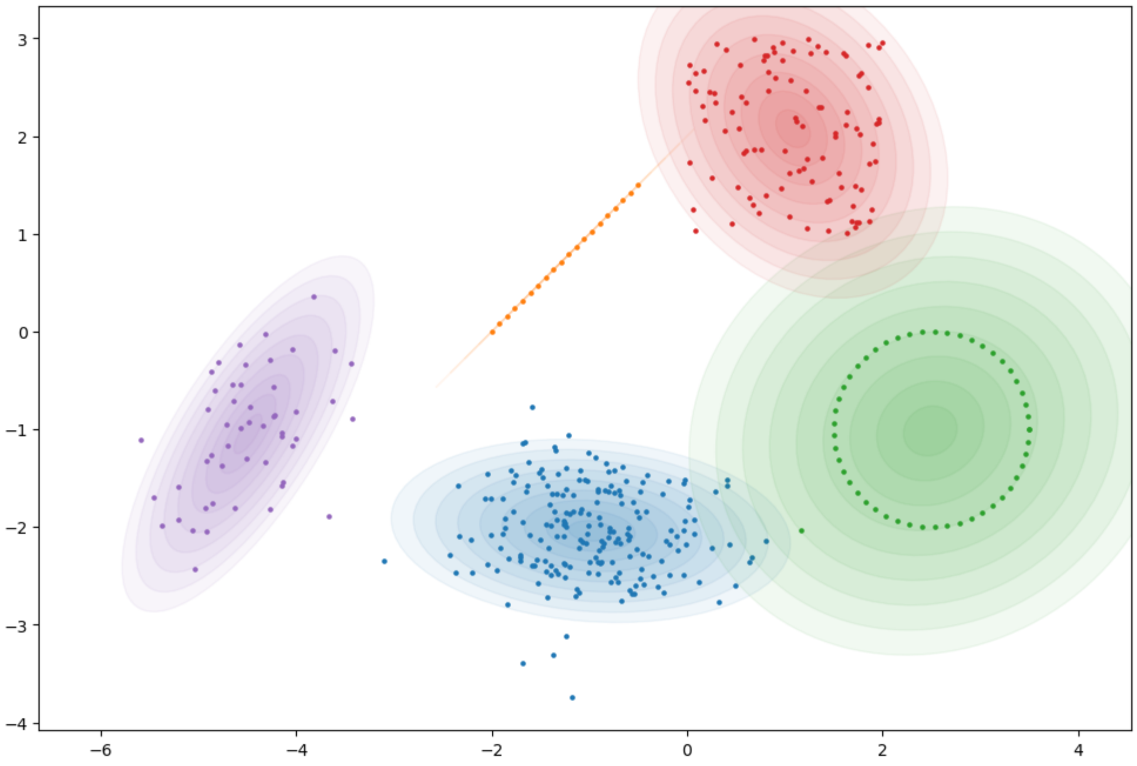
```
import matplotlib as mpl
```

```
k = 5

gm = mixture.GaussianMixture(k)
gm.fit(X)

colors = list(mpl.colors.TABLEAU_COLORS.values())

fig, ax = plt.subplots(figsize=(12, 8))

for label in range(0, k):
    mean = gm.means_[label, :]
    cov = gm.covariances_[label, :, :]

    # do some calculations for drawing an ellipse
    evals, evecs = np.linalg.eigh(cov)
    if evecs[0, 0] == 0:
        angle = 0.5 * np.pi
    else:
        angle = np.arctan(evecs[1, 0] / evecs[0, 0])

    # draw ellipses (level sets of Gaussian density)
    for fac in range(1, 10):
        ell = mpl.patches.Ellipse(mean, fac * evals[0], fac * evals[1],␣
↪angle=angle / np.pi * 180,
                                  color=colors[label] + '10')
        ax.add_artist(ell)

    # plot cluster
    labels = gm.predict(X)
```

(continues on next page)

(continued from previous page)

```
    mask = labels == label
    ax.scatter(X[mask, 0], X[mask, 1], s=5, c=colors[label])

ax.axis('equal')
plt.show()
```



CHAPTER

SEVEN

AUTOENCODERS

An autoencoder is a special ANN mainly used for dimensionality reduction and anomaly detection. In principle, autoencoders can be used for generating new samples similar to training samples (generative model), but there exist much better generative models nowadays. We introduce autoencoders here, because they are a relatively simple ancestor of variational autoencoders, one of two very prominent generative machine learning techniques (the other prominent technique are generative adversarial networks).

Related projects:

- *MNIST Character Recognition* (page 117)
  - *Autoencoder for QMNIST* (page 120)

7.1 ANN Structure

Like PCA, autoencoders aim at reducing feature space dimension without loosing relevant information. PCA follows a clear mathematical reasoning to reduce feature space dimension. In particular, for PCA the transform between original and reduced feature space is linear (multiplication by a matrix). Autoencoders use ANNs for transforming features and also for the inverse transform. Thus, transformation may be highly nonlinear and, thus, much more flexible.

Encoder ANN and decoder ANN are trained simultaneously by joining them to form one large ANN. The input neurons of the decoder are connected to the output neurons of the encoder. Training data consists of (unlabeled) samples which are used both as inputs and as outputs of the ANN. So the ANN learns to reproduce its inputs!

The output of the encoder sometimes is denoted as *code*. If the number of encoder output neurons (dimension of reduced feature space or code space) is smaller than the original feature space dimension, then the ANN is forced to drop information not relevant for reproducing inputs.

There is nothing special about training an autoencoder. Usually mean squared error is used as loss. Regularization may be used to enforce certain special properties of the codes. An example are sparse codes (resulting from ℓ^1-penalty), which may be used together with high dimensional code spaces to identify most relevant original features.

7.2 Implementing an Autoencoder

Scikit-Learn does not offer autoencoders (at the moment), so we have to use Keras and create an ANN with special autoencoder structure manually.

We first create a simple toy data set for demonstration.

```python
import numpy as np
import matplotlib.pyplot as plt

rng = np.random.default_rng(0)
```

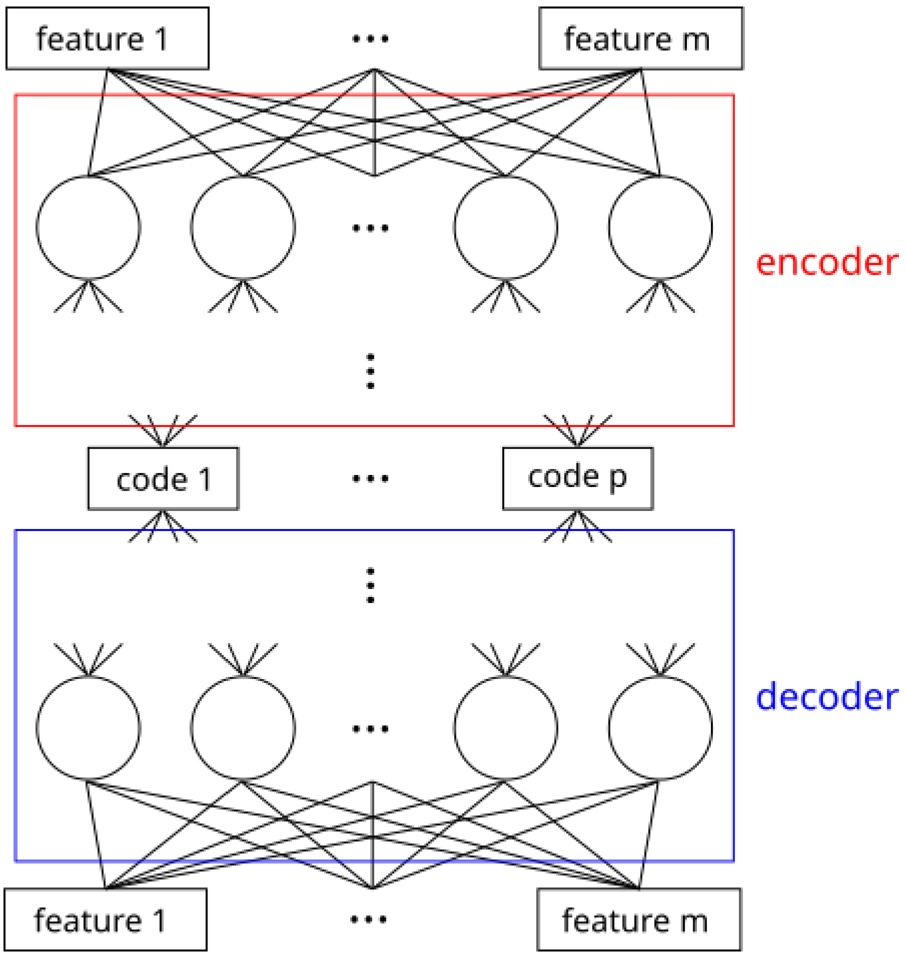
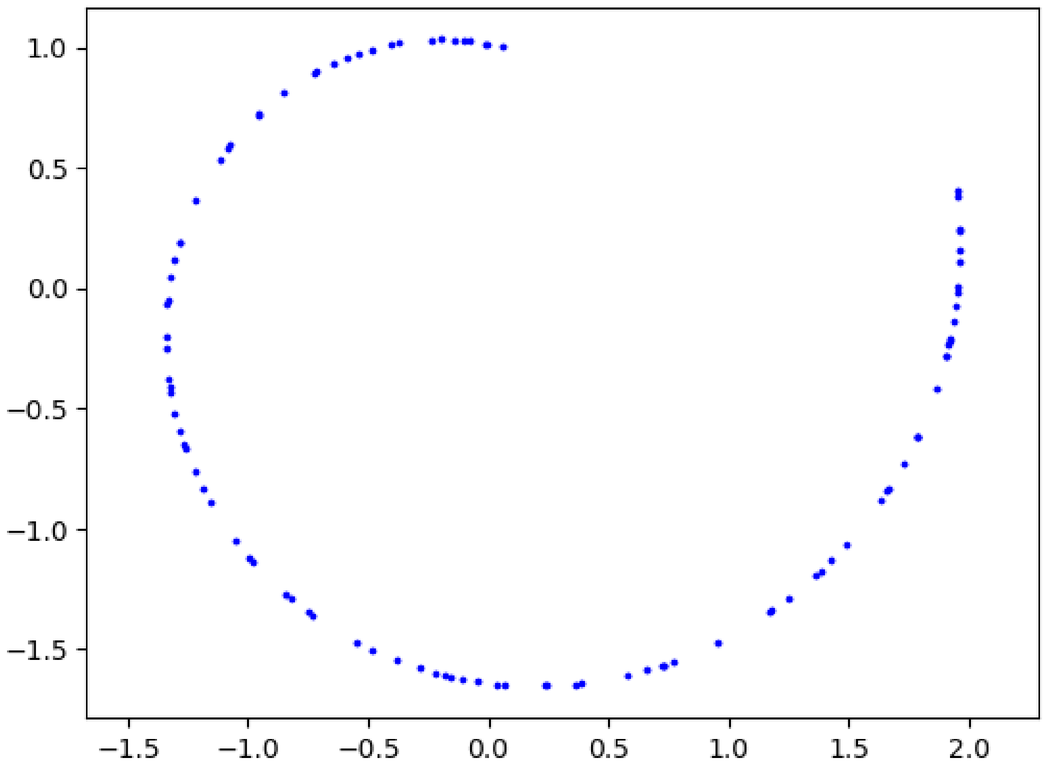


Fig. 7.1: Autoencoders are ANNs with as many outputs as inputs.

```
n = 100

t = rng.uniform(0, 1, n)
X = np.empty((n, 2))
X[:, 0] = (t + 1) * np.cos(5 * t + 1.5)
X[:, 1] = (t + 1) * np.sin(5 * t + 1.5)

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], c='b', s=3)
ax.axis('equal')
plt.show()
```



Original data space has two dimensions here. Human eye immediately sees that the data set lives on a one dimensional manifold. Thus, we could unwind the curve to map the data set into a one dimensional space without loss of information. So code space should have one dimension and the encoder ANN has to learn the unwinding operation. We give it a try with two small layers. Using only one layer would yield a more or less linear transform, which obviously cannot unwind the curve.

```
# disable GPU if TensorFlow with GPU causes problems
import os
os.environ["CUDA_VISIBLE_DEVICES"] = "-1"

import tensorflow.keras as keras
```

```
2023-10-13 05:39:50.963324: I tensorflow/core/platform/cpu_feature_guard.
 ↪cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network␣
 ↪Library (oneDNN) to use the following CPU instructions in performance-
 ↪critical operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
 ↪compiler flags.
```

```
encoder = keras.Sequential(name='encoder')
encoder.add(keras.Input(shape=(2,)))
encoder.add(keras.layers.Dense(10, activation='relu', name='enc1'))
encoder.add(keras.layers.Dense(10, activation='relu', name='enc2'))
encoder.add(keras.layers.Dense(1, activation='linear', name='code'))

encoder.summary()
```

```
Model: "encoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 enc1 (Dense)                (None, 10)                30        
                                                                 
 enc2 (Dense)                (None, 10)                110       
                                                                 
 code (Dense)                (None, 1)                 11        
                                                                 
=================================================================
Total params: 151
Trainable params: 151
Non-trainable params: 0
_________________________________________________________________
```

```
2023-10-13 05:39:52.663649: E tensorflow/compiler/xla/stream_executor/cuda/cuda_
↪driver.cc:267] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable␣
↪device is detected
2023-10-13 05:39:52.663688: I tensorflow/compiler/xla/stream_executor/cuda/cuda_
↪diagnostics.cc:169] retrieving CUDA diagnostic information for host: WHZ-46349
2023-10-13 05:39:52.663696: I tensorflow/compiler/xla/stream_executor/cuda/cuda_
↪diagnostics.cc:176] hostname: WHZ-46349
2023-10-13 05:39:52.663862: I tensorflow/compiler/xla/stream_executor/cuda/cuda_
↪diagnostics.cc:200] libcuda reported version is: 525.125.6
2023-10-13 05:39:52.663889: I tensorflow/compiler/xla/stream_executor/cuda/cuda_
↪diagnostics.cc:204] kernel reported version is: 525.125.6
2023-10-13 05:39:52.663895: I tensorflow/compiler/xla/stream_executor/cuda/cuda_
↪diagnostics.cc:310] kernel version seems to match DSO: 525.125.6
2023-10-13 05:39:52.664376: I tensorflow/core/platform/cpu_feature_guard.
↪cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network␣
↪Library (oneDNN) to use the following CPU instructions in performance-
↪critical operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
↪compiler flags.
```

The decoder ANN looks the same because it as to learn the inverse of the encoder. So complexity of the mappings to learn by encoder and decoder is comparable.

```
decoder = keras.Sequential(name='decoder')
decoder.add(keras.Input(shape=(1,)))
decoder.add(keras.layers.Dense(10, activation='relu', name='dec1'))
decoder.add(keras.layers.Dense(10, activation='relu', name='dec2'))
decoder.add(keras.layers.Dense(2, activation='linear', name='reconstruction'))

decoder.summary()
```

```
Model: "decoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
```

(continues on next page)

(continued from previous page)

```
 dec1 (Dense)                (None, 10)                20

 dec2 (Dense)                (None, 10)                110

 reconstruction (Dense)      (None, 2)                 22

=================================================================
Total params: 152
Trainable params: 152
Non-trainable params: 0
_________________________________________________________________
```

Now we join encoder and decoder. Keras models can be used as layers of a new model. So we only have to create a sequential model with encoder and decoder as layers. Important: all three models use the same underlying computation graph.

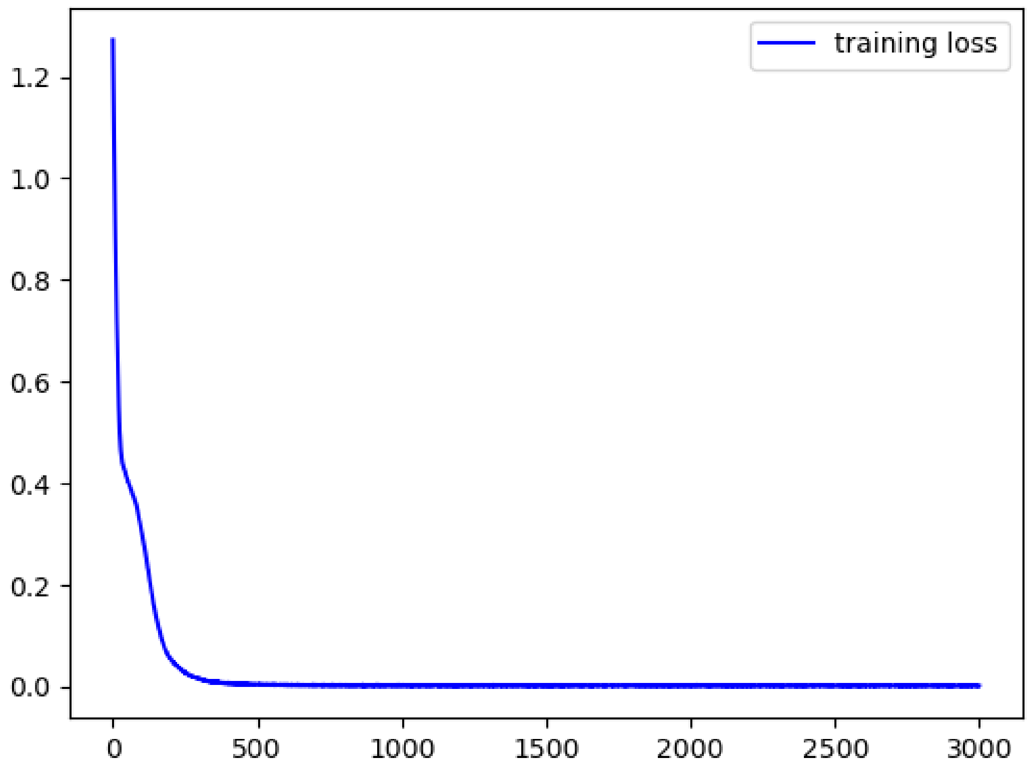
```
autoencoder = keras.Sequential(name='autoencoder')
autoencoder.add(keras.Input(shape=(2,)))
autoencoder.add(encoder)
autoencoder.add(decoder)

autoencoder.summary()
```

```
Model: "autoencoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder (Sequential)        (None, 1)                 151

 decoder (Sequential)        (None, 2)                 152

=================================================================
Total params: 303
Trainable params: 303
Non-trainable params: 0
_________________________________________________________________
```

We see that the number of weights equals the sum of the weights of encoder and decoder. The autoencoder model shares weights with encoder and decoder models. If we train the autoencoder the weights of encoder and decoder get modified, too (because there is only one computation graph).

```
autoencoder.compile(loss='mean_squared_error')

loss = []
```

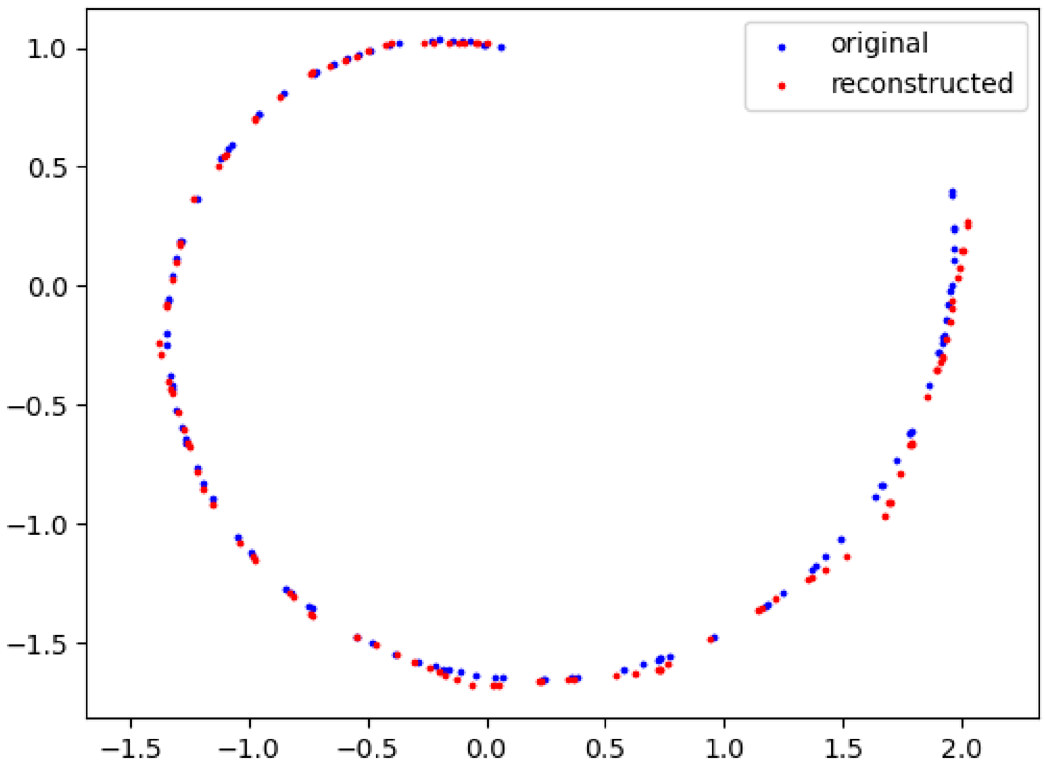
```
history = autoencoder.fit(X, X, epochs=1000, verbose=0)

loss.extend(history.history['loss'])

fig, ax = plt.subplots()
ax.plot(loss, '-b', label='training loss')
ax.legend()
plt.show()
```



To see whether training has been successful we may calculate root mean squared error and plot original and reconstructed data.

```
X_pred = autoencoder.predict(X)

print('RMSE:', np.sqrt(((X - X_pred) ** 2).sum() / n))

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], c='b', s=3, label='original')
ax.scatter(X_pred[:, 0], X_pred[:, 1], c='r', s=3, label='reconstructed')
ax.axis('equal')
ax.legend()
plt.show()
```

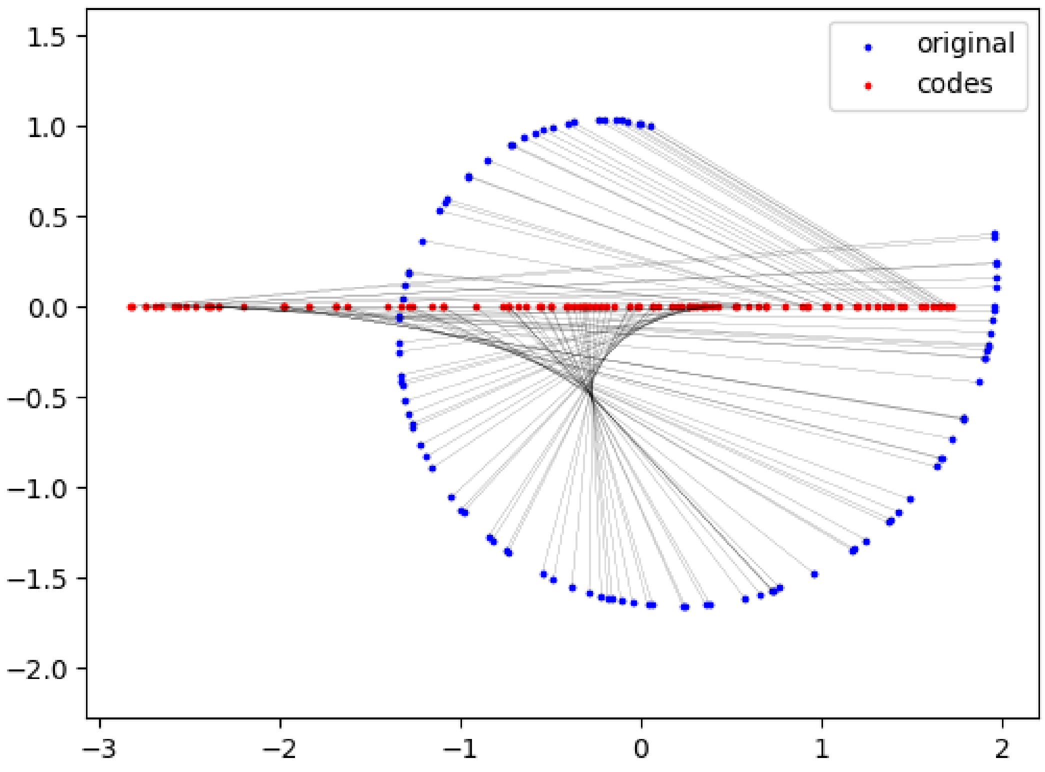
```
4/4 [==============================] - 0s 1ms/step
RMSE: 0.048846450064516586
```



Codes live in one dimension. To visualize the mapping of original data to codes we embed the one dimensional code space into the original data space and connect each sample with its code by a line.

```
C = encoder.predict(X)

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], c='b', s=3, label='original')
ax.scatter(C, np.zeros_like(C), c='r', s=3, label='codes')
for l in range(0, n):
    ax.plot([X[l, 0], C[l, 0]], [X[l, 1], 0], 'k', linewidth=0.1)
ax.axis('equal')
ax.legend()
plt.show()
```

```
4/4 [==============================] - 0s 1ms/step
```



From the visualization we see that the autoencoder preserves order. This is not (!) a general property of autoencoders, but here it works.

7.3 Comparison to PCA

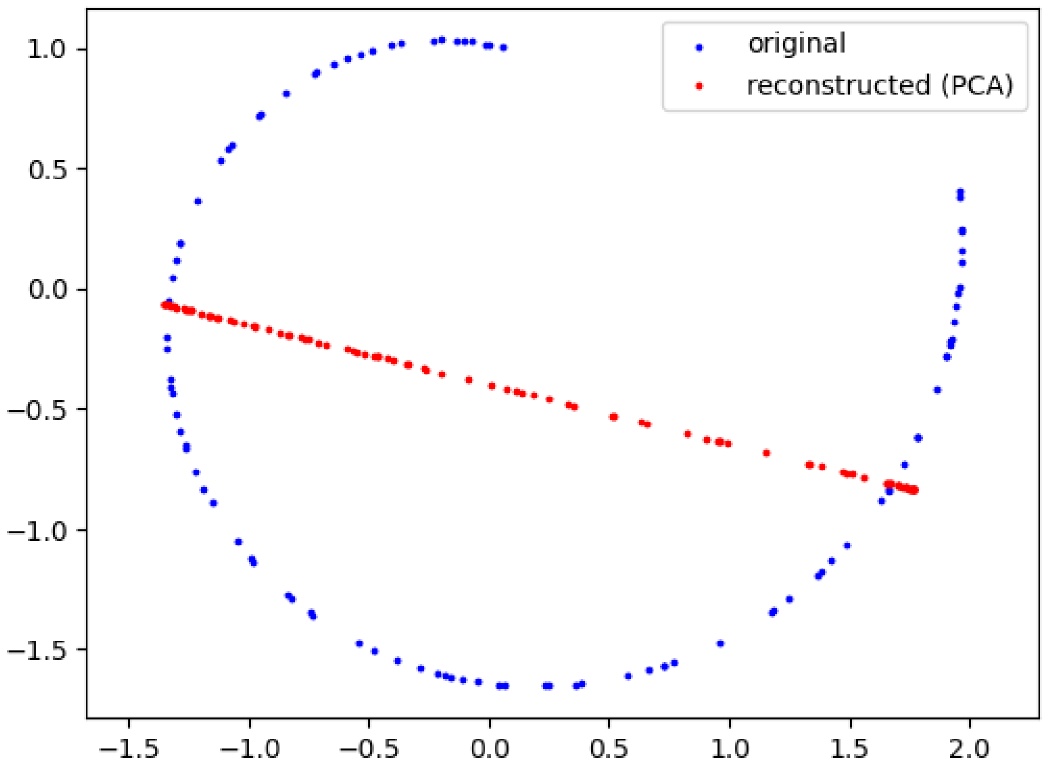
For comparison we use PCA to reduce feature space dimesion.

```python
import sklearn.decomposition as decomposition
```

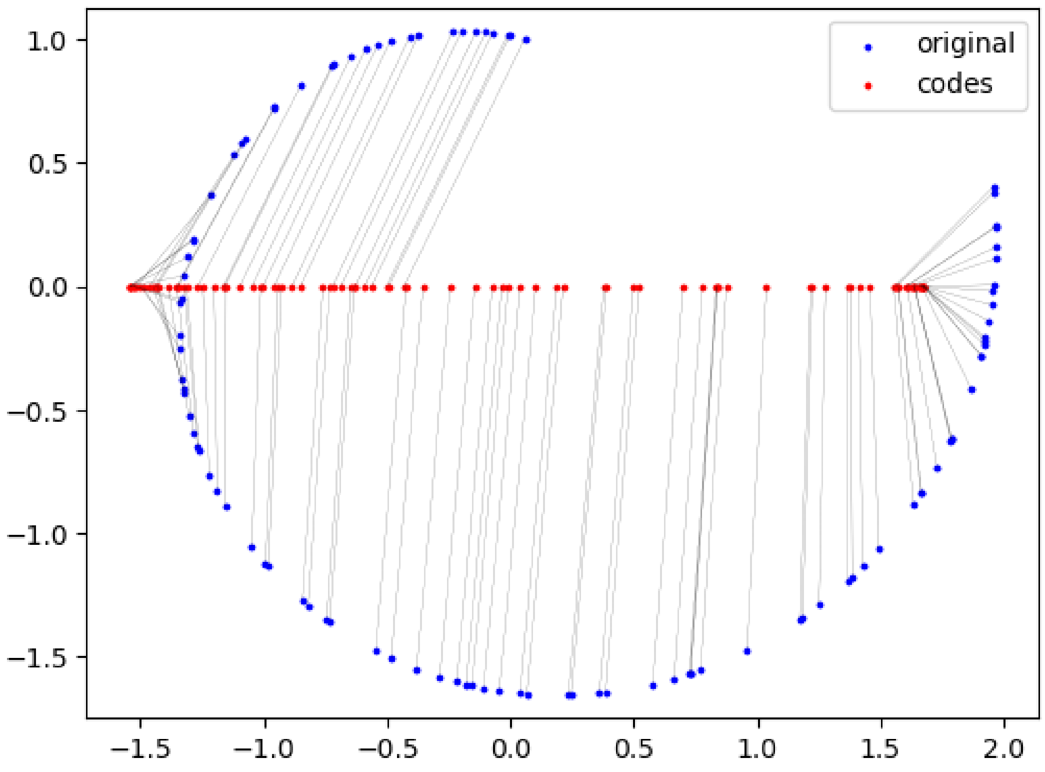
```python
pca = decomposition.PCA(1)
C_pca = pca.fit_transform(X)     # encoder
X_pca = pca.inverse_transform(C_pca)     # decoder
```

```python
print('RMSE:', np.sqrt(((X - X_pca) ** 2).sum() / n))

fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], c='b', s=3, label='original')
ax.scatter(X_pca[:, 0], X_pca[:, 1], c='r', s=3, label='reconstructed (PCA)')
ax.axis('equal')
ax.legend()
plt.show()
```

```
RMSE: 0.9049436506773132
```



```
fig, ax = plt.subplots()
ax.scatter(X[:, 0], X[:, 1], c='b', s=3, label='original')
ax.scatter(C_pca, np.zeros_like(C_pca), c='r', s=3, label='codes')
for l in range(0, n):
    ax.plot([X[l, 0], C_pca[l, 0]], [X[l, 1], 0], 'k', linewidth=0.1)
ax.axis('equal')
ax.legend()
plt.show()
```



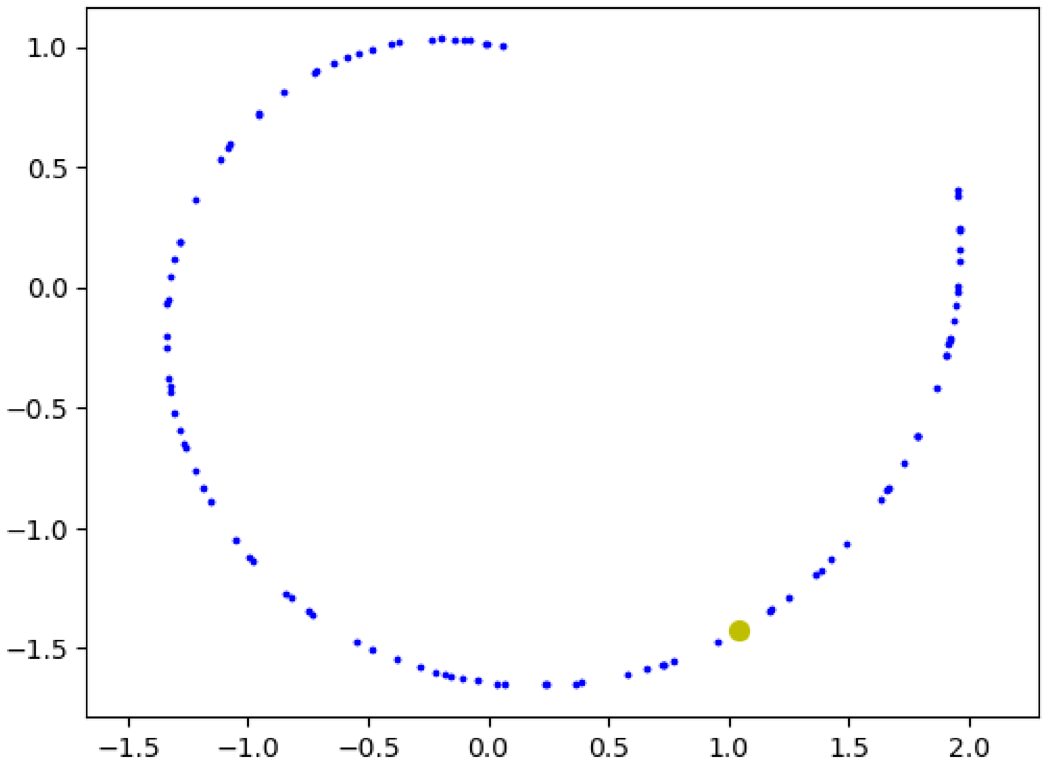
PCA maps distant points to similar codes because PCA maps original data to codes by orthogonal projection.

7.4 Generating New Samples

Autoencoders can be used as generative models. If we feed some input (code) into the decoder, it will yield some output. At least if the input is in the range of codes generated from training data output should be close to original data, that is, similar to training samples.

```
C_new = np.array([[-1]])
#C_new = np.linspace(-10, 10, 100).reshape(-1, 1)

X_new = decoder.predict(C_new)

fig, ax = plt.subplots()
ax.scatter(X_new[:, 0], X_new[:, 1], c='y', s=50)
ax.scatter(X[:, 0], X[:, 1], c='b', s=3)
ax.axis('equal')
plt.show()
```

```
1/1 [==============================] - 0s 20ms/step
```

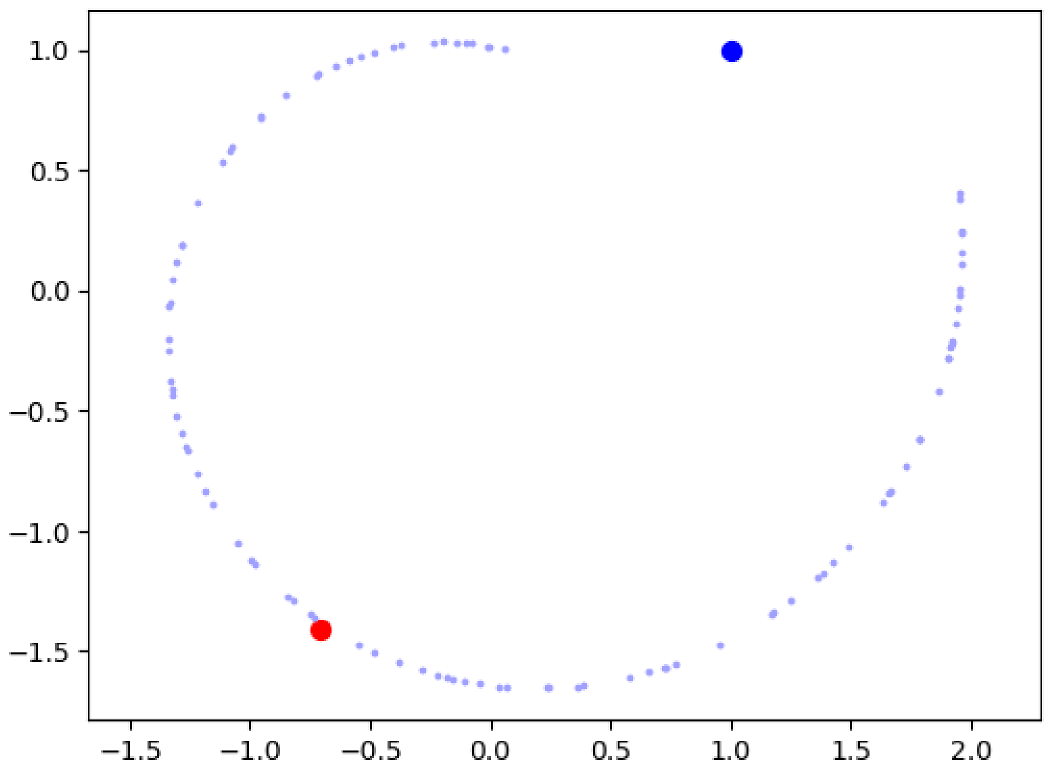


7.5 Anomaly Detection

Autoencoders typically yield good reconstructions only on data similar to training data. If we feed an autoencoder with samples very different from training data reconstructions won't look like inputs. This behavior can be used for anomaly detection. If an autoencoder is trained on 'positive' data only (legitimate credit card transactions for instance) it is likely to yield bad reconstructions on 'negative' samples (fraudulent credit card transactions).

```
X_bad = np.array([[1, 1]])
#X_bad = np.array([[-1, -1]])

X_bad_pred = autoencoder.predict(X_bad)

print('RMSE:', np.sqrt(((X_bad - X_bad_pred) ** 2).sum() / n))

fig, ax = plt.subplots()
ax.scatter(X_bad[:, 0], X_bad[:, 1], c='b', s=50)
ax.scatter(X_bad_pred[:, 0], X_bad_pred[:, 1], c='r', s=50)
ax.scatter(X[:, 0], X[:, 1], c='#a0a0ff', s=3)
ax.axis('equal')
plt.show()
```

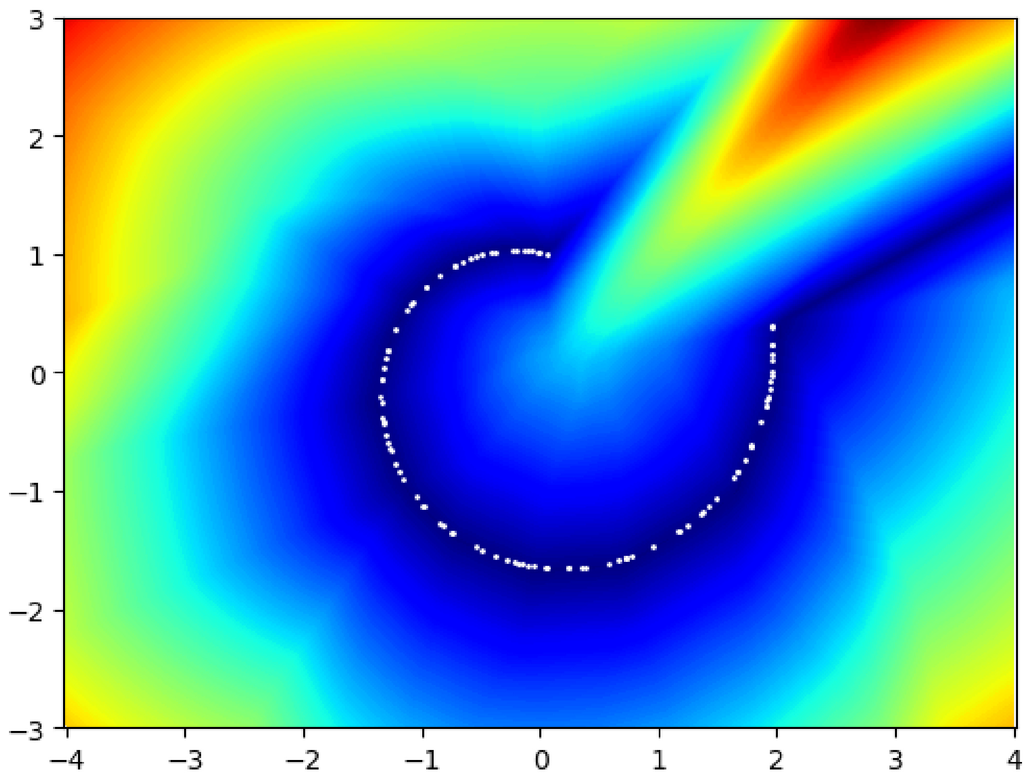
```
1/1 [==============================] - 0s 20ms/step
RMSE: 0.29520192221997077
```



Calculating RMSE between original and reconstructed samples for a grid of samples we get a visual impression of the autoencoder mapping.

```
n_grid = 200

u, v = np.meshgrid(np.linspace(-4, 4, n_grid), np.linspace(-3, 3, n_grid))
X_grid = np.concatenate((u.reshape(-1, 1), v.reshape(-1, 1)), axis=1)

X_grid_pred = autoencoder.predict(X_grid)

errors = np.sqrt(((X_grid - X_grid_pred) ** 2).sum(axis=1) / n)

fig, ax = plt.subplots()
ax.contourf(u, v, errors.reshape(n_grid, n_grid), cmap='jet', levels=200)
ax.scatter(X[:, 0], X[:, 1], c='w', s=1)
ax.axis('equal')
plt.show()
```

```
1250/1250 [==============================] - 1s 817us/step
```



7.6 Convolutional Autoencoders

For image processing task CNNs (convolutional ANNs) are known to perform much better than fully connected CNNs. Using a CNN as encoder is straight forward, but how to 'invert' a CNN for decoding? The decoder has to upsample the code until the original image size is reached. For this upsampling process we have to options:

- simple upsampling (e.g. duplicating each pixel),
- transposed convolution, also known as backward convolution.

A transposed convolution layer has inverse structure of a convolution layer (inputs and outputs switched). Like a usual convolutional layer, it is defined by a stack of filters (shared weights) and a stride value (step size for moving the filter to the next position). Note that a transposed convolution layer only inverts the structure (connections between neurons), not the calculation. So chaining convolution and transposed convolution is not the identity, but only the chain's input shape equals its output shape.

From the computation scheme we see that transposed convolution can be regarded as usual convolution with zero padded inputs (add zeros on left and right side).

While stride is applied to inputs for usual convolution, for transposed convolution stride is applied to the outputs.

A decoder symmetric to a CNN encoder can be constructed by inverting the encoder's layer stack and replacing pooling by upsampling and convolutions by transposed convolutions. Corresponding Keras layers are `UpSampling2D`[31] and `Conv2DTranspose`[32]. 1d and 2d variants exist as well, but `Conv1DTranspose` not before TensorFlow 2.3.

[31] https://keras.io/api/layers/reshaping_layers/up_sampling2d
[32] https://keras.io/api/layers/convolution_layers/convolution2d_transpose

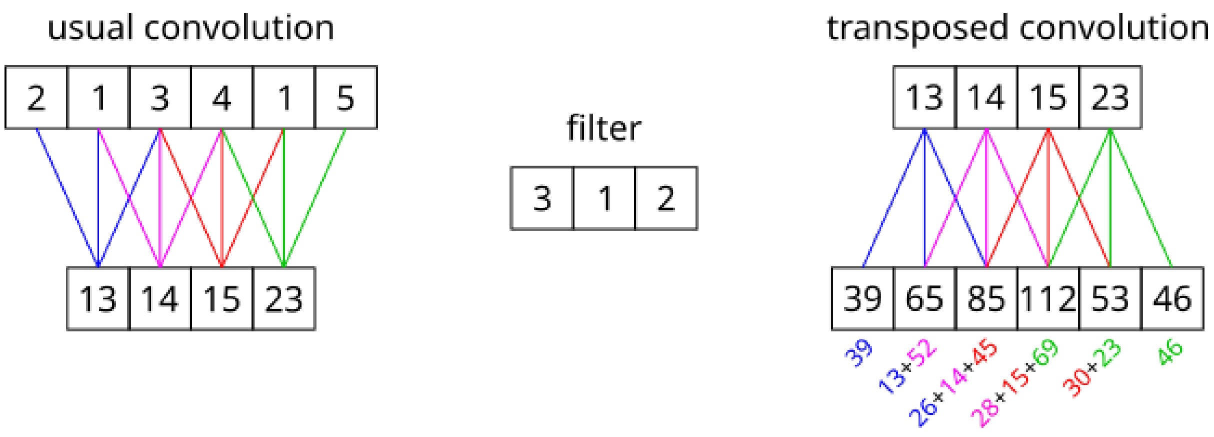


Fig. 7.2: Transposed convolutions do NOT invert convolutions but corresponding shapes only.

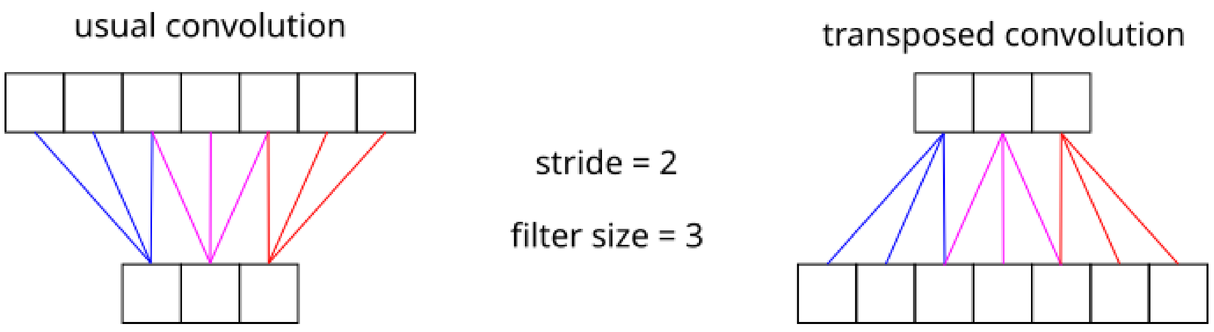


Fig. 7.3: Striding for transposed convolutions is realized such that resulting that input and output shapes match ouput and input shapes of corresponding usual convolution.

CHAPTER

EIGHT

NONLINEAR DIMENSIONALITY REDUCTION OVERVIEW

Up to now we only considered PCA and autoencoders for dimensionality reduction. PCA is a linear technique, that is, it applies a linear transform (matrix multiplication) to the data. Autoencoders are nonlinear and, thus, more flexible.

There exist many other nonlinear techniques for dimensionality reduction. Dimensionality reduction is very important for visualizing high dimensional data. Some of them turned out to be more or less equivalent, some are different realizations of the same idea. The following scheme provides an overview:

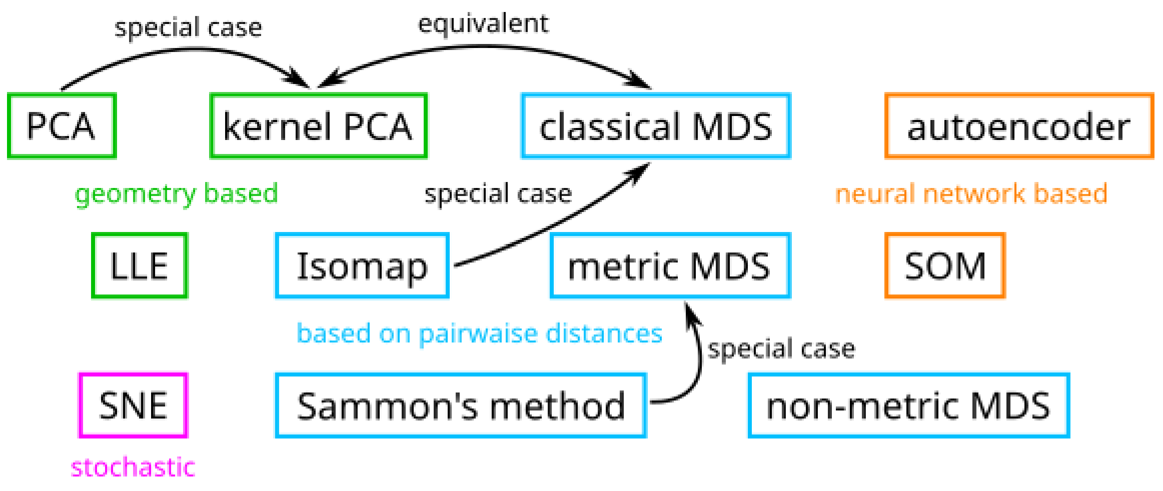


Fig. 8.1: Nonlinear dimensionality reduction is a wide field, but several methods turn out to be equivalent after careful inspection.

8.1 Toy Example 'Omega'

The first toy example for testing nonlinear dimensionality reduction is an Ω-shaped two dimensional manifold in $\mathbb{R}^3$. Data points lie (up to some noise) on this nonlinear manifold.

To each generated sample we assign a different color. Thus, after embedding the manifold into 2d space we can reconstruct from where in 3d space the sample came.

```
import numpy as np
import plotly.graph_objects as go

rng = np.random.default_rng()
```

```
n1 = 75     # number of grid points in first dimension
n2 = 40     # number of gird points in second dimension
noise = 0.005     # noise level for moving samples away from the manifold

```

(continues on next page)

(continued from previous page)

```
# parameter space
S, T = np.meshgrid(np.linspace(0, 1, n1), np.linspace(0, 1, n2))
S = S.reshape(-1)
T = T.reshape(-1)

# noise in parameter space to destroy rigid grid structure
S = S + rng.normal(0, 1 / (2 * n1), S.size)
T = T + rng.normal(0, 1 / (2 * n2), T.size)

# cut-off to keep parameters in [0, 1]
S = np.clip(S, 0, 1)
T = np.clip(T, 0, 1)

# samples in 3d
x = S + 0.15 * np.sin(4 * np.pi * S)
y = T
z = 5 * np.maximum(0, -np.abs(S - 0.5) + 0.5) ** 1 + 1 * T ** 2

# colors
red = np.sin(4 * np.pi * S)
green = np.sin(2 * np.pi * T)
blue = np.sin(2 * np.pi * (S + T))
red = (255 * (red - red.min()) / (red - red.min()).max()).astype(int)
green = (255 * (green - green.min()) / (green - green.min()).max()).astype(int)
blue = (255 * (blue - blue.min()) / (blue - blue.min()).max()).astype(int)

# some noise
x = x + rng.normal(0, noise, x.size)
y = y + rng.normal(0, noise, y.size)
z = z + rng.normal(0, noise, z.size)

# plot
fig = go.Figure(layout_width=800, layout_height=600, layout_scene_aspectmode='cube
 ↪')
fig.add_trace(go.Scatter3d(
    x=x, y=y, z=z,
    mode='markers',
    marker={'size': 2, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
 ↪green, blue)]},
    hoverinfo = 'none'
))
fig.show()
```

```
<IPython.core.display.HTML object>
```

```
np.savez('omega.npz', x=x, y=y, z=z, red=red, green=green, blue=blue)
```

8.2 Toy Example 'Sphere'

The next toy example is a sphere shaped 2d manifold in 3d space. This manifold cannot be mapped into 2d space without cuts or overlaps.

```
n = 40     # number of stacked circles
noise = 0.05     # noise level for moving samples away from the manifold

# phi is latitude angle
```

(continues on next page)

(continued from previous page)

```
# theta is longitude angle
# number of longitudes depends on latitude (more points per latitude on equator␣
↪than near poles)
x = []
y = []
z = []
red = []
green = []
blue = []
for phi in np.linspace(0, np.pi, n + 2)[1:-1]:
    m = int(2 * n * np.abs(np.sin(phi)))
    for i in range(0, m):
        phi_noisy = phi + rng.normal(0, np.pi / (2 * n))
        r = np.sin(phi_noisy)
        theta = i * 2 * np.pi / m + rng.normal(0, np.pi / m)
        x.append(r * np.cos(theta))
        y.append(r * np.sin(theta))
        z.append(np.cos(phi_noisy))
        red.append(np.sin(2 * phi_noisy))
        green.append(np.sin(2 * theta))
        blue.append(np.sin(2 * (phi_noisy + theta)))

x = np.array(x)
y = np.array(y)
z = np.array(z)

red = np.array(red)
green = np.array(green)
blue = np.array(blue)
red = (255 * (red - red.min()) / (red - red.min()).max()).astype(int)
green = (255 * (green - green.min()) / (green - green.min()).max()).astype(int)
blue = (255 * (blue - blue.min()) / (blue - blue.min()).max()).astype(int)

# some noise
x = x + rng.normal(0, noise, x.size)
y = y + rng.normal(0, noise, y.size)
z = z + rng.normal(0, noise, z.size)

# plot
fig = go.Figure(layout_width=800, layout_height=600, layout_scene_aspectmode='cube
↪')
fig.add_trace(go.Scatter3d(
    x=x, y=y, z=z,
    mode='markers',
    marker={'size': 2, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
    hoverinfo = 'none'
))
fig.show()
```

```
<IPython.core.display.HTML object>
```

```
np.savez('sphere.npz', x=x, y=y, z=z, red=red, green=green, blue=blue)
```

8.3 Toy Example 'Cube'

The next example is a filled 3d cube, which cannot be mapped into two dimensions without destroying its structure.

```
n = 15    # number of grid points per axis

x, y, z = np.meshgrid(np.linspace(0, 1, n), np.linspace(0, 1, n), np.linspace(0,␣
↪1, n))
x = x.reshape(-1)
y = y.reshape(-1)
z = z.reshape(-1)

# some noise
x = x + rng.normal(0, 1 / (2 * n), x.size)
y = y + rng.normal(0, 1 / (2 * n), y.size)
z = z + rng.normal(0, 1 / (2 * n), z.size)

# colors
red = np.cos(2 * np.pi * x)
green = np.cos(2 * np.pi * y)
blue = np.cos(2 * np.pi * z)
red = (255 * (red - red.min()) / (red - red.min()).max()).astype(int)
green = (255 * (green - green.min()) / (green - green.min()).max()).astype(int)
blue = (255 * (blue - blue.min()) / (blue - blue.min()).max()).astype(int)

# plot
fig = go.Figure(layout_width=800, layout_height=600, layout_scene_aspectmode='cube
↪')
fig.add_trace(go.Scatter3d(
    x=x, y=y, z=z,
    mode='markers',
    marker={'size': 2, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
    hoverinfo = 'none'
))
fig.show()
```

```
<IPython.core.display.HTML object>
```

```
np.savez('cube.npz', x=x, y=y, z=z, red=red, green=green, blue=blue)
```

8.4 Toy Example 'Clouds'

The next example data set consists of four 2d point clouds in 3d space. With this data set we can investigate the behavior of nonlinear dimensionality reduction techniques for nonconnected data sets.

```
X1 = rng.multivariate_normal((-1, -1, -1), ((0.1, 0.1, 0.01), (0.1, 0.2, 0.1), (0.
↪01, 0.1, 0.1)), 300)
X2 = rng.multivariate_normal((1, 1, 1), ((0.1, 0.09, 0.01), (0.09, 0.2, 0.08), (0.
↪01, 0.08, 0.05)), 300)
X3 = rng.multivariate_normal((-1, 1, -1), ((0.2, 0.1, 0.1), (0.1, 0.4, 0.1), (0.1,
↪ 0.1, 0.08)), 500)
X4 = rng.multivariate_normal((1, 1, -1), ((0.1, 0.1, 0.01), (0.1, 0.2, 0.1), (0.
↪01, 0.1, 0.1)), 300)

X1 = (1 + X1 / np.abs(X1).max()) / 2
X2 = (1 + X2 / np.abs(X2).max()) / 2
```

(continues on next page)

(continued from previous page)

```
X3 = (1 + X3 / np.abs(X3).max()) / 2
X4 = (1 + X4 / np.abs(X4).max()) / 2

X = np.concatenate((X1, X2, X3, X4))
x = X[:, 0]
y = X[:, 1]
z = X[:, 2]

dists1 = np.sum(np.abs(X1 - X1.mean(axis=0)) ** 0.8, axis=1)
dists2 = np.sum(np.abs(X2 - X2.mean(axis=0)) ** 0.8, axis=1)
dists3 = np.sum(np.abs(X3 - X3.mean(axis=0)) ** 0.8, axis=1)
dists4 = np.sum(np.abs(X4 - X4.mean(axis=0)) ** 0.8, axis=1)

red1 = 1 - dists1 / dists1.max()
green1 = np.ones(X1.shape[0])
blue1 = np.zeros(X1.shape[0])

red2 = np.zeros(X2.shape[0])
green2 = 1 - dists2 / dists2.max()
blue2 = np.ones(X2.shape[0])

red3 = np.ones(X3.shape[0])
green3 = np.zeros(X3.shape[0])
blue3 = 1 - dists3 / dists3.max()

red4 = np.ones(X4.shape[0])
green4 = 1 - dists4 / dists4.max()
blue4 = np.zeros(X4.shape[0])

red = np.concatenate((red1.reshape(-1, 1), red2.reshape(-1, 1), red3.reshape(-1,␣
↪1), red4.reshape(-1, 1)), axis=0).reshape(-1)
green = np.concatenate((green1.reshape(-1, 1), green2.reshape(-1, 1), green3.
↪reshape(-1, 1), green4.reshape(-1, 1)), axis=0).reshape(-1)
blue = np.concatenate((blue1.reshape(-1, 1), blue2.reshape(-1, 1), blue3.reshape(-
↪1, 1), blue4.reshape(-1, 1)), axis=0).reshape(-1)
red = (255 * (red - red.min()) / (red - red.min()).max()).astype(int)
green = (255 * (green - green.min()) / (green - green.min()).max()).astype(int)
blue = (255 * (blue - blue.min()) / (blue - blue.min()).max()).astype(int)

# some noise
x = x + rng.normal(0, noise, x.size)
y = y + rng.normal(0, noise, y.size)
z = z + rng.normal(0, noise, z.size)

# plot
fig = go.Figure(layout_width=800, layout_height=600, layout_scene_aspectmode='cube
↪')
fig.add_trace(go.Scatter3d(
    x=x, y=y, z=z,
    mode='markers',
    marker={'size': 2, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
    hoverinfo = 'none'
))
fig.show()
```

```
<IPython.core.display.HTML object>
```

```
np.savez('clouds.npz', x=x, y=y, z=z, red=red, green=green, blue=blue)
```

8.5 PCA for Toy Examples

To compare results from nonlinear dimensionality reduction to the linear standard technique PCA we plot 2d PCA projections for all toy examples.

```
import sklearn.decomposition as decomposition
from sklearn.preprocessing import StandardScaler
from plotly.subplots import make_subplots
```

```
data_files = ['omega.npz', 'sphere.npz', 'cube.npz', 'clouds.npz']

for file in data_files:

    loaded = np.load(file)
    x = loaded['x']
    y = loaded['y']
    z = loaded['z']
    red = loaded['red']
    green = loaded['green']
    blue = loaded['blue']

    pca = decomposition.PCA(2)
    U = pca.fit_transform(StandardScaler().fit_transform(np.stack((x, y, z),␣
↪axis=1)))

    fig = make_subplots(rows=1, cols=2, specs=[[{'type': 'scatter3d'}, {'type':
↪'xy'}]])
    fig.update_layout(width=1000, height=600, scene_aspectmode='cube')
    fig.add_trace(go.Scatter3d(
        x=x, y=y, z=z,
        mode='markers',
        marker={'size': 1.5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,
↪ green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=1)
    fig.add_trace(go.Scatter(
        x=U[:, 0], y=U[:, 1],
        mode='markers',
        marker={'size': 5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=2)
    fig.show()
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

CHAPTER

NINE

KERNEL PCA

Principal component analysis (PCA) contructs a new coordinate system in the data space such that variance is maximal along the first axis of this new coordinate system. Variance along the other axes decreases with increasing axis index. Coordinates of the data set with respect to the new coordinate system have zero covariance.

Such data adapted coordinate system at hand data dimensionality can be reduced by dropping axes of low variance or, in other words, by projecting onto the subspace spanned by the first few axes.

While classical PCA is a linear technique (data transformation by matrix multiplication) kernel PCA extends the basic idea to nonlinear transforms by exploiting the kernel trick. We already met the kernel trick when discussing support vector machines. In theory data is (nonlinearly) embedded into a higher dimensional space and PCA together with corresponding dimensionality reduction is performed there. But in practice, all computations are carried out in the original space by replacing inner products in high dimensions by special kernel functions.

In the following we always count eigenvalues with their multiplicity. So each real symmetric matrix has as many eigenvalues as it has rows and columns.

9.1 Recap of PCA

Denote by $X \in \mathbb{R}^{n\times m}$ the matrix containing all samples $x_1, \ldots, x_n$ as rows. Here m is the dimension of the data space. We assume that data is centered, that is,

$$\sum_{l=1}^{n} x_l = 0.$$

The (up to a factor $\frac{1}{n-1}$) covariance matrix $X^\mathrm{T} X \in \mathbb{R}^{m\times m}$ is symmetric and positive semi-definite. Thus, there exist m nonnegative numbers

$$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_m \geq 0 \qquad \text{(eigenvalues)}$$

and m mutually orthogonal normalized vectors $a_1, \ldots, a_m$ (eigenvectors) such that

$$X^\mathrm{T} X\, a_k = \lambda_k\, a_k \qquad \text{for } k = 1, \ldots, m.$$

The vectors a_k form the coordinate axes of the PCA coordinate system and the values λ_k are the variances (up to a factor $\frac{1}{n-1}$) of the data set along those axes. Denoting by $A \in \mathbb{R}^{m\times m}$ the orthogonal matrix of eigenvectors (as columns) the coordinate matrix $\tilde{X}$ with respect to the new coordinate system is given by

$$\tilde{X} = X\, A.$$

Equivalently, for each sample x_l and its transformed variant $\tilde{x}_l$ we have the relations

$$\tilde{x}_l = \begin{bmatrix} {x_l}^\mathrm{T} a_1 \\ \vdots \\ {x_l}^\mathrm{T} a_m \end{bmatrix} \qquad \text{and} \qquad x_l = \tilde{x}_l^{(1)}\, a_1 + \cdots + \tilde{x}_l^{(m)}\, a_m.$$

Dimensionality reduction now boils down to replacing vectors a_k corresponding to small λ_k by zero vcetors. The transform from x_l to $\tilde{x}_l$ and back to x_l then does not yield x_l again, but its projection onto the subspace spanned by the remaining a_k.

9.2 PCA with Kernel Trick

Kernel PCA appeared first in the 1996 report Nonlinear Component Analysis as a Kernel Eigenvalue Problem[33]. Introducing a (nonlinear) mapping $\Phi : \mathbb{R}^m \to \mathbb{R}^q$ from the m dimensional data space into a much higher dimensional embedding space of dimension q with $q \geq m$ the authors propose to perform PCA based dimensionality reduction for $\{\Phi(x_1), \ldots, \Phi(x_n)\}$, the embedded data set. They have shown that PCA transformed data (in the q dimensional space) can be computed without touching q dimensional vectors and without expensive computations in q dimensions.

9.2.1 The Kernel PCA Theorem

Given the embedding function Φ as above we define the kernel matrix $K \in \mathbb{R}^{n\times n}$ by

$$K_{l,\lambda} := \Phi(x_l)^{\mathrm{T}}\,\Phi(x_\lambda).$$

By $U \in \mathbb{R}^{n\times q}$ we denote the matrix containing $\Phi(x_1), \ldots, \Phi(x_n)$ as rows (by analogy with X for usual PCA in data space). Then one can show (and we will do this below) that

- **the eigenvalues of $U^{\mathrm{T}}\,U$ and K coincide,**
- **the eigenvectors of K contain the coordinates of $\Phi(x_1), \ldots, \Phi(x_n)$ with respect to the PCA coordinate system in the q dimensional space.**

In other words, we only have to compute eigenvalues and eigenvectors of the kernel matrix K when performing PCA for the embedded data. Size of K depends on number of samples, which might be large, but often much smaller than q. Remember: for polynomial kernel of degree 2 and 1000 dimensional data (images!) we would have $q = 501501$ (see chapter on SVMs).

The first statement above requires some clarifying remarks.

- The (up to a factor) covariance matrix $U^{\mathrm{T}}\,U$ is of size $q \times q$. Thus, PCA yields q eigenvalues and q eigenvectors. But the embedded data lives in a subspace of dimension at most n. Thus, there are at most n nonzero eigenvalues for $U^{\mathrm{T}}\,U$.
- Writing $K = U\,U^{\mathrm{T}}$ we see that K is a symmetric matrix. Thus, the number of nonzero eigenvalues equals the rank of K (standard math result). Because $U\,U^{\mathrm{T}}$ and $U^{\mathrm{T}}\,U$ have identical rank (another standard math result), K and $U^{\mathrm{T}}\,U$ have the same number of nonzero eigenvalues.
- The maximum number of nonzero eigenvalues of K and $U^{\mathrm{T}}\,U$, and thus also the maximum number of non-trivial features after dimensionality reduction via kernel PCA, is $\min\{n, q\}$.

The second statement above on the eigenvectors of K requires some clarification, too. In addition, we should formulate it more precisely.

- The new data adapted coordinate system in q dimensions has q axes.
- Embedded data $\{\Phi(x_1), \ldots, \Phi(x_n)\}$ uses not more than the first $\min\{n, q\}$ axes. Coordinates for remaining axes are zero.
- In context of dimensionality reduction we are interested in the coordinates with respect to the first p axes only. Here p is small, in particular $p \leq \min\{n, q\}$.
- If $B \in \mathbb{R}^{q\times q}$ contains the eigenvectors of $U^{\mathrm{T}}\,U$ (in columns), then by analogy to usual PCA the coordinates with respect to the new coordinate system are given by

$$\tilde{U} := U\,B.$$

- From the previous two items we see that the desired result from dimensionality reduction via kernel PCA is contained in the first p columns of $\tilde{U}$.
- The kernel PCA theorem states that the columns of $\tilde{U}$ are the eigenvectors of K.

Before we come to the proof, we have to think about centering embedded data. Remember: PCA is sensible for centered data only.

[33] https://www.face-rec.org/algorithms/Kernel/kernelPCA_scholkopf.pdf

9.2.2 Centering Embedded Data

Before applying PCA we have to center the data. Given the embedding function Φ we have to apply PCA to data

$$\Phi(x_1) - \frac{1}{n}\sum_{l=1}^{n}\Phi(x_l) \quad , \quad \dots \quad , \quad \Phi(x_n) - \frac{1}{n}\sum_{l=1}^{n}\Phi(x_l).$$

The new embedding function

$$\Phi_{\text{center}}(x) := \Phi(x) - \frac{1}{n}\sum_{l=1}^{n}\Phi(x_l)$$

automatically yields centered embedded data.

We may rewrite this as

$$\Phi_{\text{center}}(x) := \Phi(x) - U^{\mathrm{T}}\begin{bmatrix}\frac{1}{n}\\ \vdots \\ \frac{1}{n}\end{bmatrix}$$

and introducing the matrix

$$M := \begin{bmatrix}\frac{1}{n} & \dots & \frac{1}{n}\\ \vdots & & \vdots \\ \frac{1}{n} & \dots & \frac{1}{n}\end{bmatrix} \in \mathbb{R}^{n\times n}$$

we obtain

$$U_{\text{center}}{}^{\mathrm{T}} = U^{\mathrm{T}} - U^{\mathrm{T}} M = U^{\mathrm{T}}(I - M).$$

The kernel corresponding to Φ_{center} then is

$$K_{\text{center}} = U_{\text{center}}\, U_{\text{center}}{}^{\mathrm{T}} = (I - M)\, U\, U^{\mathrm{T}}(I - M) = (I - M)\, K\, (I - M).$$

Centering the embedded data reduces to a simple transform of the kernel. If the kernel K is positive semi-definite (which is a standard assumption), then the new kernel K_{center} is positive semi-definite, too. So centering the embedded data set does not imply any troubles. The kernel trick still works.

Passing remark: The matrix $I - M$ is known as *centering matrix*. Multiplying a Matrix K from the right by $I - M$ centers its columns (sum of all columns is zero vector). Multiplication from the left centers its rows (sum of all rows is zero vector). Applying both multiplications yields the *double centered* version of K. The mean over all entries of a double centered matrix is zero, too.

9.2.3 Proof of Kernel PCA Theorem

We assume $n \leq q$ (which is the typical situation in practice) to avoid repreated use of $\min\{n, q\}$ in the formulas. But the proof also works for $n > q$ if most n are replaced by $\min\{n, q\}$. Further we assume that K has rank n, that is, all eigenvalues are nonzero. So we may write n instead of rank K to simplify formulas. All arguments also work for rank below n if considerations are restricted to nonzero eigenvalues of K.

Denoting the nonzero eigenvalues of the kernel matrix K by $\mu_1, \dots, \mu_n$ and corresponding eigenvectors by $\omega_1, \dots, \omega_n \in \mathbb{R}^n$ we define vectors $b_1, \dots, b_n \in \mathbb{R}^q$ by

$$b_i := \frac{1}{\mu_i}\sum_{l=1}^{n}\omega_i^{(l)}\,\Phi(x_l), \qquad i = 1, \dots, n.$$

To proof the theorem it suffices to show that

- $b_1, \dots, b_n$ are eigenvectors of $U^{\mathrm{T}} U$ and $\mu_1, \dots, \mu_n$ are corresponding eigenvalues,
- $\omega_1, \dots, \omega_n$ contain the transformed coordinates, more precisely,

$$\omega_i = U\, b_i, \qquad i = 1, \dots, n.$$

For the first statement we have to show $U^{\mathrm{T}}\,U\,b_i=\mu_i\,b_i$. This follows from the definition of b_i, some simple manipulations and the fact that ω_i is an eigenvector of K to eigenvalue μ_i:

$$\begin{aligned}
U^{\mathrm{T}}\,U\,b_i &= \frac{1}{\mu_i}\sum_{l=1}^{n}\omega_i^{(l)}\,U^{\mathrm{T}}\,U\,\Phi(x_l)=\frac{1}{\mu_i}\sum_{l=1}^{n}\omega_i^{(l)}\,U^{\mathrm{T}}\begin{bmatrix}\Phi(x_1)^{\mathrm{T}}\,\Phi(x_l)\\ \vdots\\ \Phi(x_n)^{\mathrm{T}}\,\Phi(x_l)\end{bmatrix}\\
&= \frac{1}{\mu_i}\sum_{l=1}^{n}\omega_i^{(l)}\sum_{\lambda=1}^{n}\underbrace{\Phi(x_\lambda)^{\mathrm{T}}\,\Phi(x_l)}_{\in\mathbb{R}}\,\Phi(x_\lambda)\\
&= \frac{1}{\mu_i}\sum_{l=1}^{n}\sum_{\lambda=1}^{n}\omega_i^{(l)}\,\Phi(x_\lambda)^{\mathrm{T}}\,\Phi(x_l)\,\Phi(x_\lambda)\\
&= \frac{1}{\mu_i}\sum_{\lambda=1}^{n}\left(\sum_{l=1}^{n}\omega_i^{(l)}\,\Phi(x_\lambda)^{\mathrm{T}}\,\Phi(x_l)\right)\Phi(x_\lambda)=\frac{1}{\mu_i}\sum_{\lambda=1}^{n}\left(\sum_{l=1}^{n}K_{\lambda,l}\,\omega_i^{(l)}\right)\Phi(x_\lambda)\\
&= \frac{1}{\mu_i}\sum_{\lambda=1}^{n}[K\,\omega_i]_\lambda\,\Phi(x_\lambda)\\
&= \frac{1}{\mu_i}\sum_{\lambda=1}^{n}\mu_i\,\omega_i^{(\lambda)}\,\Phi(x_\lambda)=\mu_i\,b_i.
\end{aligned}$$

The second statement is a direct consequence of the definition of b_i:

$$U\,b_i=\frac{1}{\mu_i}\sum_{l=1}^{n}\omega_i^{(l)}\,U\,\Phi(x_l)=\frac{1}{\mu_i}\sum_{l=1}^{n}\omega_i^{(l)}\,K_{\bullet,l}=\frac{1}{\mu_i}\,K\,\omega_i=\omega_i.$$

9.3 Kernel PCA with Scikit-Learn

Scikit-Learn provides the `KernelPCA`[34] class in its `decomposition` module.

```python
import numpy as np
import sklearn.decomposition as decomposition
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

```python
data_files = ['omega.npz', 'sphere.npz', 'cube.npz', 'clouds.npz']

for file in data_files:

    loaded = np.load(file)
    x = loaded['x']
    y = loaded['y']
    z = loaded['z']
    red = loaded['red']
    green = loaded['green']
    blue = loaded['blue']

    kpca = decomposition.KernelPCA(2, kernel='rbf', gamma=5)
    U = kpca.fit_transform(np.stack((x, y, z), axis=1))

    fig = make_subplots(rows=1, cols=2, specs=[[{'type': 'scatter3d'}, {'type':
↪'xy'}]])
    fig.update_layout(width=1000, height=600, scene_aspectmode='cube')
    fig.add_trace(go.Scatter3d(
        x=x, y=y, z=z,
        mode='markers',
```

(continues on next page)

[34] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.KernelPCA.html

(continued from previous page)

```
        marker={'size': 1.5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,
↪ green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=1)
    fig.add_trace(go.Scatter(
        x=U[:, 0], y=U[:, 1],
        mode='markers',
        marker={'size': 5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=2)
    fig.show()
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

9.4 Why Kernel PCA?

Kernel PCA only yields useful results if the embedding map Φ somehow unroles the nonlinear structure of the data set such that in higher dimensions the data set is almost linear. Kernel PCA is rarely used in practise, because one does not know how to choose the kernel. But we will see in subsequent chapters that other nonlinear dimensionality reduction techniques turn out to be equivalent to kernel PCA with specific data set adapted kernel.

CHAPTER

TEN

MULTIDIMENSIONAL SCALING

We already considered techniques for dimensionality reduction in the context of feature reduction. There the aim was to construct more expressive features and to remove unnecessary ones. Now the focus is on visualization, that is, reduction of data sets to 2 or 3 dimensions. This way me may get a better understanding of a data set and we also may check results obtained from clustering methods.

We already met two techniques for dimensionality reduction:

- (Kernel) principal component analysis (PCA), which simply projects a data set orthogonally onto a lower dimensional linear manifold,
- Autoencoders, which try to find two mappings between the high and a low dimensional space such that their composition resamples the data set in the high dimensional space as good as possible.

PCA is a linear dimensionality reduction technique, because it performs a linear transform (multiplication by matrix). Kernel PCA and autoencoders are nonlinear dimensionality reduction techniques, because the mapping from high to low dimensions cannot be expressed by matrix multiplication.

Multidimensional scaling (MDS) refers to a third fundamental concept for dimensionality reduction. Here we try to find a set of points in low dimensions such that pairwise distances are as close as possible to pairwise distances in the high dimensional space. MDS comes in several variants differing in the choice of the distance measure and also in the weighting of pairwise distances.

Related projects:

- *Color Perception* (page 135)
- *Forest Fires* (page 139)

10.1 Abstract Mathematical Formulation

To make things precise, denote by $(x_1,\ldots,x_n)$ the data set in the high dimensional space $\mathbb{R}^m$ and by $(u_1,\ldots,u_n)$ a set of points in a lower dimensional space $\mathbb{R}^p$. Note that both sets are of equal size n. Further let d_m and d_p be similarity measures in high and low dimensions, respectively. By $w_{l,\lambda}\in(0,\infty)$, $l,\lambda=1,\ldots,n$ we denote some weights. Then MDS aims at solving

$$\sum_{l=1}^{n}\sum_{\lambda=1}^{n} w_{l,\lambda}\left(d_m(x_l,x_\lambda)-d_p(u_l,u_\lambda)\right)^2\to\min_{u_1,\ldots,u_n}.$$

Squaring is somewhat arbitrary here. We could apply any positive and increasing function to the difference of pairwise distances. But in each conrete MDS variant the square is the most fortunate choice, because it ensures differentiability and simplifies computation of gradients.

MDS solely relies on similarities $d_m(x_l,x_\lambda)$ between samples and does not touch samples directly. This observation allows for applications with arbitrary types of data (text data, for instance) as long as there is some notion of similarity. In addition, the similarity measure in high dimensions is not required to be some precise mathematical construct. Human scoring is possible, too, for instance.

10.2 Metric MDS and Sammon's Method

In metric MDS the low dimensional similarity measure is the squared Euclidean distance

$$d_p(u_l, u_\lambda) = |u_l - u_\lambda|^2.$$

So metric MDS tries to preserve pairwise Euclidean distances (if distances in high dimensions are Euclidean, too).

The minimization problem has to be solved numerically by gradient descent or Newton-type methods (iterative methods using second order derivatives). Results may be inaccurate due to local minima or saddle points. Starting guess may be determined by PCA or chosen at random.

Without weights (all weights set to 1) large distances will dominate the objective, because an error in distance fitting of 10 per cent has more influence on the objective for large distances than for small distances. Thus, without weights the local structure of the data set is not well reconstructed in low dimensions.

Usually one is more interested in the local structure than in the global one. For instance, the shape of a cluster or the boundary region between closely spaced clusters are more interesting than the correct distance between clusters far apart from each other.

Weighting by inverse distances solves this issue and puts emphasis on local structures. Typically, weights are

$$w_{l,\lambda} = \frac{1}{d_m(x_l, x_\lambda)}.$$

This weighted variant of metric MDS is known as *Sammon's method*.

```
import numpy as np
import sklearn.manifold as manifold
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

```
data_files = ['omega.npz', 'sphere.npz', 'cube.npz', 'clouds.npz']

for file in data_files:

    loaded = np.load(file)
    x = loaded['x']
    y = loaded['y']
    z = loaded['z']
    red = loaded['red']
    green = loaded['green']
    blue = loaded['blue']

    mds = manifold.MDS(2, normalized_stress='auto')
    U = mds.fit_transform(np.stack((x, y, z), axis=1))

    fig = make_subplots(rows=1, cols=2, specs=[[{'type': 'scatter3d'}, {'type':
↪'xy'}]])
    fig.update_layout(width=1000, height=600, scene_aspectmode='cube')
    fig.add_trace(go.Scatter3d(
        x=x, y=y, z=z,
        mode='markers',
        marker={'size': 1.5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,
↪ green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=1)
    fig.add_trace(go.Scatter(
        x=U[:, 0], y=U[:, 1],
        mode='markers',
        marker={'size': 5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
```

(continues on next page)

(continued from previous page)

```
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=2)
    fig.show()
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

10.3 Classical MDS

Like metric MDS classical MDS aims at fitting Euclidean distances, but avoids iterative minimization. We will see that applying a linear transform to the Euclidean distance matrices in both low and high dimensions before fitting allows for analytical minimization. Here, by Euclidean distance matrix we denote a matrix containing in row i and column j the squared Euclidean distance between sample i and sample j.

We will discuss several aspects step by step:

- the transform applied to Euclidean distances to simplify minimization,
- reconstruction of (the lower dimensional) data set from transformed distances
- formulation and solution of the minimization problem,
- modifications to allow for non-Euclidean distances in high dimensions,
- relations to PCA and kernel PCA.

10.3.1 From Distances to Inner Products

Let $z_1,\ldots,z_n\in\mathbb{R}^q$ be a collection of points in q dimensions, think of $q=m$ with $z_l=x_l$ or $q=p$ with $z_l=u_l$. Corresponding Euclidean distance matrix $D\in\mathbb{R}^{n\times n}$ is given by

$$D_{l,\lambda}:=|z_l-z_\lambda|^2$$

and the inner product matrix $S\in\mathbb{R}^n$ of the centered data set $z_1-\bar{z},\ldots,z_n-\bar{z}$ with $\bar{z}:=\frac{1}{n}\sum_{l=1}^n z_l$ is defined by

$$S_{l,\lambda}:=(z_l-\bar{z})^{\mathrm{T}}\,(z_\lambda-\bar{z}).$$

With

$$M:=\begin{bmatrix}\frac{1}{n}&\cdots&\frac{1}{n}\\ \vdots&&\vdots\\ \frac{1}{n}&\cdots&\frac{1}{n}\end{bmatrix}\in\mathbb{R}^{n\times n}$$

we now show

$$S=-\frac{1}{2}\,(I-M)\,D\,(I-M),$$

that is, double centering the Euclidean distance matrix yields the inner product matrix of the centered data set (up to some constant factor).

From the definition of matrix multiplication and then from

$$|z_l-z_\lambda|^2=|z_l|^2+|z_\lambda|^2-2\,z_l{}^{\mathrm{T}}\,z_\lambda$$

we see

$$\begin{aligned}
[(I-M)\,D\,(I-M)]_{l,\lambda} &= D_{l,\lambda} - [M\,D]_{l,\lambda} - [D\,M]_{l,\lambda} + [M\,D\,M]_{l,\lambda}\\
&= D_{l,\lambda} - \frac{1}{n}\sum_{i=1}^{n} D_{i,\lambda} - \frac{1}{n}\sum_{j=1}^{n} D_{l,j} + \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} D_{i,j}\\
&= |z_l|^2 + |z_\lambda|^2 - 2\,{z_l}^{\mathrm{T}}\,z_\lambda - \left(|z_\lambda|^2 + \frac{1}{n}\sum_{i=1}^{n}(|z_i|^2 - 2\,{z_i}^{\mathrm{T}}\,z_\lambda)\right)\\
&\quad - \left(|z_l|^2 + \frac{1}{n}\sum_{j=1}^{n}(|z_j|^2 - 2\,{z_l}^{\mathrm{T}}\,z_j)\right) + \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}(|z_i|^2 + |z_j|^2 - 2\,{z_i}^{\mathrm{T}}\,z_j).
\end{aligned}$$

Summands $|z_l|^2$ and $|z_\lambda|^2$ each appear twice with different sign, thus vanish. The double sum can be simplified to

$$\begin{aligned}
\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}(|z_i|^2 + |z_j|^2 - 2\,{z_i}^{\mathrm{T}}\,z_j) &= \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}|z_i|^2 + \frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}|z_j|^2 - \frac{2}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n}{z_i}^{\mathrm{T}}\,z_j\\
&= \frac{1}{n}\sum_{i=1}^{n}|z_i|^2 + \frac{1}{n}\sum_{j=1}^{n}|z_j|^2 - 2\left(\frac{1}{n}\sum_{i=1}^{n}z_i\right)^{\mathrm{T}}\left(\frac{1}{n}\sum_{j=1}^{n}z_j\right),
\end{aligned}$$

so that the sums over $|z_i|^2$ and $|z_j|^2$ cancel out, too. Thus, we have

$$\begin{aligned}
[(I-M)\,D\,(I-M)]_{l,\lambda} &= -2\,{z_l}^{\mathrm{T}}\,z_\lambda + \frac{2}{n}\sum_{i=1}^{n}{z_i}^{\mathrm{T}}\,z_\lambda + \frac{2}{n}\sum_{j=1}^{n}{z_l}^{\mathrm{T}}\,z_j - 2\,\bar{z}^{\mathrm{T}}\,\bar{z}\\
&= -2\,{z_l}^{\mathrm{T}}\,z_\lambda + 2\,\bar{z}^{\mathrm{T}}\,z_\lambda + 2\,{z_l}^{\mathrm{T}}\,\bar{z} - 2\,\bar{z}^{\mathrm{T}}\,\bar{z}\\
&= -2\,(z_l - \bar{z})^{\mathrm{T}}\,(z_\lambda - \bar{z}) = -2\,S_{l,\lambda}.
\end{aligned}$$

10.3.2 From Inner Products to Data

From an inner product matrix S we want to reconstruct original data $z_1, \dots, z_n$. As we will see, the mean $\bar{z}$ cannot be reconstructed from S. Choosing an arbitrary $\bar{z}$ amounts in a translation of the whole data compared to the original one. In addition there will be some freedom in rotating and mirrowing the reconstructed data set.

The data set can be reconstructed from the eigendecomposition of S. The matrix S is symmetric and positive semi-definite. So there exist n nonnegative numbers $\lambda_1 \geq \dots \geq \lambda_n \geq 0$ (eigenvalues of S) and n vectors $a_1, \dots, a_n \in \mathbb{R}^n$ (eigenvectors of S) with

$$S = A\,\Lambda\,A^{\mathrm{T}},$$

where A contains the eigenvectors as columns and Λ is the diagonal matrix of eigenvalues.

If $Z \in \mathbb{R}^{n\times q}$ contains $z_1, \dots, z_n$ as rows, then we have

$$S = ((I-M)\,Z)\,((I-M)\,Z)^{\mathrm{T}} = (I-M)\,Z\,Z^{\mathrm{T}}\,(I-M).$$

The rank of Z is at most q and, thus, the rank of S is at most q, too. So all but the first q eigenvalues of S are zero. Here we assume $q \leq n$. The case $q > n$ is not of interest to us, because for MDS we will have $q = p$ (dimension of low dimensional space, in practice 2 or 3).

With an arbitrary orthonormal matrix $R \in \mathbb{R}^{q\times q}$ (rotation and/or mirrowing) we set

$$\tilde{z}_l := R\begin{bmatrix}\sqrt{\lambda_1}\,a_1^{(l)}\\ \vdots \\ \sqrt{\lambda_q}\,a_q^{(l)}\end{bmatrix} \qquad \text{for } l = 1, \dots, n.$$

Denoting the diagonal matrix of square roots of the eigenvalues by $\lambda^{\frac{1}{2}}$ we see

$$\tilde{Z} = A\,\Lambda^{\frac{1}{2}}\,R^{\mathrm{T}}$$

and

$$\tilde{Z}\,\tilde{Z}^{\mathrm{T}} = A\,\Lambda^{\frac{1}{2}}\,R^{\mathrm{T}}\,R\,\Lambda^{\frac{1}{2}}\,A^{\mathrm{T}} = A\,\Lambda\,A^{\mathrm{T}} = S.$$

That is, S is the inner product matrix of $\tilde{z}_1, \ldots, \tilde{z}_n$.

Now from

$$|\tilde{z}_l - \tilde{z}_\lambda|^2 = |\tilde{z}_l|^2 + |\tilde{z}_\lambda|^2 - 2\,\tilde{z}_l^{\mathrm{T}}\,\tilde{z}_\lambda = S_{l,l} + S_{\lambda,\lambda} - 2\,S_{l,\lambda} = |z_l - \bar{z} - (z_\lambda - \bar{z})|^2 = |z_l - z_\lambda|^2$$

we see that the reconstructed data set $\tilde{z}_1, \ldots, \tilde{z}_n$ and the original data set $z_1, \ldots, z_n$ have identical distance matrices.

10.3.3 The Minimization Problem

Classical MDS fits inner products, not distances. The following steps lead from a Euclidean distance matrix $D \in \mathbb{R}^{n\times n}$ in high dimensions to a set of points $u_1, \ldots, u_n \in \mathbb{R}^p$ in low dimensions:

- Calculate the inner products matrix $S \in \mathbb{R}^{n\times n}$ in high dimensions:

$$S = -\frac{1}{2}\,(I - M)\,D\,(I - M).$$

Here we do not have to know the underlying data set $x_1, \ldots, x_n \in \mathbb{R}^m$ explicitly, because inner products are determined by pairwise Euclidean distances (up to centering).

- Solve

$$\sum_{l=1}^{n}\sum_{\lambda=1}^{n}(S_{l,\lambda} - T_{l,\lambda})^2 \to \min_{T\in\mathbb{R}^{n\times n}} \qquad \text{considering only symmetric } T \text{ of rank } p.$$

- Interpret the optimal T as inner product matrix of n points in $\mathbb{R}^p$ and reconstruct corresponding points $u_1, \ldots, u_n$.

In contrast to metric MDS classical MDS does not fit pairwise distances directly, but transforms distances to inner products and fits those inner products. Distance matrices and inner product matrices carry identical information (up to translation, rotation, mirrowing of the data set), but due to different objective functions both variants of MDS yield different results. Metric MDS minimizes the means squared error (MSE) of pairwise distances:

$$\sum_{l=1}^{n}\sum_{\lambda=1}^{n}\left(|x_l - x_\lambda|^2 - |u_l - u_\lambda|^2\right)^2.$$

Classical MDS minimizes (assuming centered data in high dimensions):

$$\sum_{l=1}^{n}\left(|x_l|^2 - |u_l|^2\right)^2 + \sum_{l=1}^{n}\sum_{\substack{\lambda=1\\ \lambda\neq l}}^{n}\left({x_l}^{\mathrm{T}}\,x_\lambda - {u_l}^{\mathrm{T}}\,u_\lambda\right)^2,$$

which is the MSE of vetor lengths (diagonal of inner product matrix) plus the MSE of angles (inner products are cosines of angles multiplied by both vector lengths).

It remains to answer the question how to solve the above minimization problem. We aim at a set of points in $\mathbb{R}^p$. So corresponding inner product matrix has rank of at most p. That's the reason why we restrict optimization to symmetric matrices of rank p. A standard result from linear algebra (Eckart-Young-Mirsky theorem)[35] tells us, that we have to look at the eigendecomposition of S:

$$S = A\,\Lambda\,A^{\mathrm{T}} \qquad \text{cf. above.}$$

Then the optimal T is (assuming $p \leq n$)

$$T = B\,\Theta\,B^{\mathrm{T}},$$

where $B \in \mathbb{R}^{n\times p}$ contains the first p eigenvectors of S as columns and $\Theta \in \mathbb{R}^{p\times p}$ is the diagonal matrix of the highest p eigenvalues of S. So solving the minimization problem reduces to an eigenvalue problem for S.

As a by by-product of minimization we get the eigendecomposition of T, which is required for reconstructing the data set $u_1, \ldots, u_n$ in low dimensions from T. Eigenvalues and eigenvectors of T coincide with the first p eigenvalues and eigenvectors of S.

[35] https://en.wikipedia.org/wiki/Low-rank_approximation#Basic_low-rank_approximation_problem

10.3.4 Non-Euclidean Distances

Classical MDS solely relies on the distance matrix D in high dimensions. The assumption that distances are Euclidean ensures that the transformed distance matrix $S=-\frac{1}{2}\,(I-M)\,D\,(I-M)$ is symmetric and positive semi-definite. So we may replace the assumption of Euclidean distances by the more direct assumption that the transformed distance matrix S is symmetric and positive semi-definite.

D may contain arbitrary (non-Euclidean) pairwise distances as long as $-\frac{1}{2}\,D$ is symmetric and positive semi-definite. So it can be regarded as a kernel matrix resulting from inner products of nonlinearly transformed data. The transform from $-\frac{1}{2}\,D$ to S is simple double centering, which is equivalent to centering the transformed data set.

We may even drop the assumption that S has to be positive semi-definite. If S is non-definite, some eigenvalues will be negative. When approximating S by some T of rank p we only consider the p largest positive eigenvalues. As long as the absolute values of negative eigenvalues is small, the error induced by ignoring them is not higher than the error resulting from low-rank approximation (that is, from dropping small positive eigenvalues). If D is based on some kind of distance, possible negative eigenvalues will be small.

As long as D is symmetric and is related to some almost arbitrary kind of distance classical MDS is applicable.

10.3.5 Relation to PCA and Kernel PCA

For centered data and Euclidean distances we have $S=X\,X^{\mathrm{T}}$. Else, S can be regarded as a double centered kernel matrix originating from a kernel matrix $-\frac{1}{2}\,D$. The transform from S to $u_1,\ldots,u_n$ in classical MDS coincides with the transform from K to the data set's first p coordinates w.r.t. to the kernel PCA coordinate system. In both cases we use the same parts of the eigendecomposition of S or K, respectively.

Classical MDS and kernel PCA are two different motivations for one and the same algorithm. In case of centered data and Euclidean distances classical MDS and (non-kernel) PCA coincide. Classical Euclidean MDS originated in the 1950s. Kernel PCA appeared in 1996 and the connection to classical MDS had been discovered a few years later.

10.4 Non-Metric MDS

For the sake of completeness we mention a third variant of MDS known as non-metric MDS, but we do not go into the details. Non-metric MDS does not focus on getting the distances right, but only on preserving the ordering of distances.

The similarity measure in low dimensions is

$$d(u_l,u_\lambda)=f(|u_l-u_\lambda|^2).$$

with some monotonically increasing function f.

Next to $u_1,\ldots,u_n$ the function f is a variable in the optimization process. A typical numerical approach is to alternate optimization steps for $u_1,\ldots,u_n$ and f. Concrete algorithms following this idea are *smallest space analysis (SSA)* and *Shepard-Kruskal algorithm*[36] (German Wikipedia).

10.5 Isomap

Isomap assumes that the data set lies on low dimensional manifold in the high dimensional space. Instead of Euclidean distances it calculates (approximate) geodesic distances with respect to the manifold. In other words, it looks for the shortest distance between two points if a traveller between both points is not allowed to leave the manifold. Pairwise geodesic distances at hand classical MDS is applied to the distance matrix.

The only thing we have to discuss is how to find (approximate) geodesic distances. This requires two steps:

[36] https://de.wikipedia.org/wiki/Multidimensionale_Skalierung#Shepard-Kruskal_Algorithmus

- Create a neighborhood graph. The nodes are the samples. A node is connected to another node by an (undirected) edge if the other node belongs to the k nearest neighbors of the first one for some prescribed k. Each edge is assigned a weight, which equals the Euclidean distance between samples connected by the edge.
- Get length of shortest path in neighborhood graph between each pair of nodes. Length of a path is the sum of all edge weights belonging to the path. Finding shortest paths in a graph is a standard task and can be solved by Dijkstra's algorithm[37].

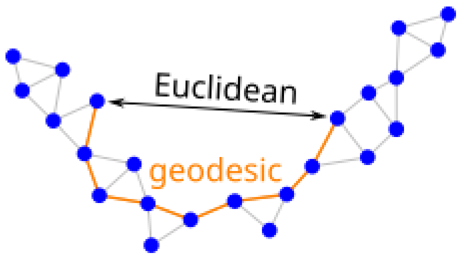


Fig. 10.1: Isomap uses approximate geodesic distances instead of Euclidean distances.

An alternative to the k nearest neighbors is to connect a node to all other nodes within a prescribed distance.

Next to high computational costs a major problem with Isomap is that there may be so called short-circuit errors. That is, the neighborhood graph contains edges betwenn non-neighboring points on the manifold. This happens especially for large k or sparse data sets.

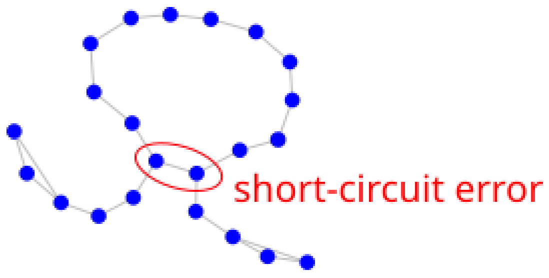


Fig. 10.2: Isomap may suffer from short-circuit errors.

Scikit-Learn has the `Isomap`[38] class in the `manifold` module.

```
data_files = ['omega.npz', 'sphere.npz', 'cube.npz', 'clouds.npz']

for file in data_files:

    loaded = np.load(file)
    x = loaded['x']
    y = loaded['y']
    z = loaded['z']
    red = loaded['red']
    green = loaded['green']
    blue = loaded['blue']

    im = manifold.Isomap(n_components=2, n_neighbors=5)
    U = im.fit_transform(np.stack((x, y, z), axis=1))

    fig = make_subplots(rows=1, cols=2, specs=[[{'type': 'scatter3d'}, {'type':
↪'xy'}]])
    fig.update_layout(width=1000, height=600, scene_aspectmode='cube')
    fig.add_trace(go.Scatter3d(
        x=x, y=y, z=z,
```

(continues on next page)

[37] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[38] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.Isomap.html

(continued from previous page)

```
        mode='markers',
        marker={'size': 1.5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,
↪ green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=1)
    fig.add_trace(go.Scatter(
        x=U[:, 0], y=U[:, 1],
        mode='markers',
        marker={'size': 5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=2)
    fig.show()
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
/home/jef19jdw/anaconda3/envs/ds_book/lib/python3.10/site-packages/sklearn/
 ↪manifold/_isomap.py:373: UserWarning:

The number of connected components of the neighbors graph is 2 > 1. Completing␣
 ↪the graph to fit Isomap might be slow. Increase the number of neighbors to␣
 ↪avoid this issue.

/home/jef19jdw/anaconda3/envs/ds_book/lib/python3.10/site-packages/scipy/sparse/
 ↪_index.py:103: SparseEfficiencyWarning:

Changing the sparsity structure of a csr_matrix is expensive. lil_matrix is␣
 ↪more efficient.
```

```
<IPython.core.display.HTML object>
```

CHAPTER

ELEVEN

LOCALLY LINEAR EMBEDDING

The idea of locally linear embedding (LLE) appeared at the same time (and journal) as Isomap, uses Euclidean distances locally only like Isomap, and results in an eigenvalue problem (again, like Isomap). But details and motivation differ.

The basic assumption is, that the data set lies on a low dimensional manifold which can be decomposed into (approximately) linear snippets. LLE tries to arange those snippets in low (often 2) dimensions without modifying their neighborhood relations.

Finding an LLE requires two steps:

- Represent each sample as an affine combination of its neighbors.
- Arrange low dimensional points such that they can be represented by the same affine combination of neighbors as in high dimensions.

11.1 Local Affine Combinations

For each sample x_l the k nearest neighbors $x_{\lambda_1},\ldots,x_{\lambda_k}$ are determined. These neighbors span an *affine manifold* (that is, a translated subspace)

$$\{a_1\,x_{\lambda_1}+\cdots+a_k\,x_{\lambda_k}\,:\,a_1,\ldots,a_k\in\mathbb{R},\;a_1+\ldots+a_k=1\}.$$

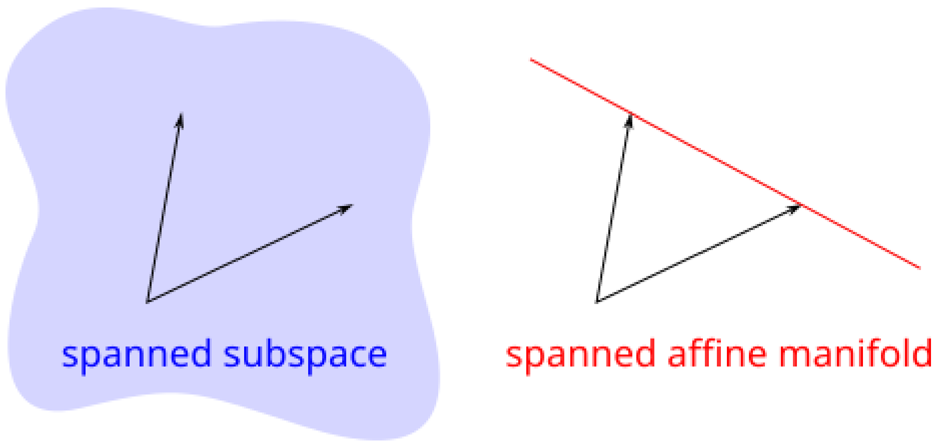


Fig. 11.1: Two vectors may span a two-dimensional subspace or a one-dimensional affine manifold depending on the set of coefficients considered.

Projecting x_l orthogonally onto the affine manifold spanned by its neighbors yields representation

$$x_l\approx w_{l,\lambda_1}\,x_{\lambda_1}+\cdots+w_{l,\lambda_k}\,x_{\lambda_k}.$$

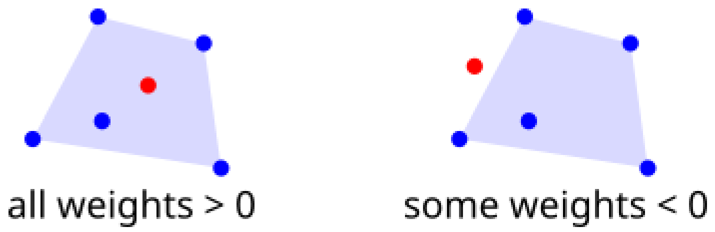


Fig. 11.2: All coefficients in an affine combination are nonnegative if and only if the resulting point belongs to the convec hull of the points being combined.

The coefficients $w_{l,\lambda_1},\ldots,w_{l,\lambda_k}$ may be regarded as weights, because their sum is 1 and often they are positive if x_l is surrounded by its neighbors (more precicely, if x_l lies in the convex hull of its neighbors.

If the data set is 'locally linear', then x_l equals the weighted sum and the error induced by representing samples as weighted sums of their neighbors is zero. The more nonlinear a data set behaves locally, the less reliable the low dimensional representation obtained via LLE.

This first step of LLE yields at most n^2 weights $w_{l,\lambda}$. Weights $w_{l,\lambda}$ for which corresponding samples aren't neighbors are set to zero. Weights are not symmetric, that is, $w_{l,\lambda}\neq w_{\lambda,l}$ in general.

To get the weights one first determines the indices $\lambda_1(l),\ldots,\lambda_k(l)$ of the k nearest neightbors for all samples x_l. Then one solves the constrained minimization problem

$$\begin{aligned}&\sum_{l=1}^{n}\left(w_{l,\lambda_1(l)}\,x_{\lambda_1(l)}+\cdots+w_{l,\lambda_k(l)}\,x_{\lambda_k(l)}-x_l\right)^2\to\min_{w_{l,\lambda}}\\&\text{with constraints}\quad w_{l,\lambda_1(l)}+\cdots+w_{l,\lambda_k(l)}=1\quad\text{for } l=1,\ldots,n\\&\text{(unused } w_{l,\lambda}\text{ are set to zero).}\end{aligned}$$

This minimization problem is quadratic and can be solved numerically in different efficient ways, for instance, by solving a system of linear equations.

11.2 Low Dimensional Fitting

Given weights $w_{l,\lambda}$ the second step of LLE is to find points $u_1,\ldots,u_n\in\mathbb{R}^p$ in low dimensions which solve

$$\begin{aligned}&\sum_{l=1}^{n}\left(w_{l,\lambda_1(l)}\,u_{\lambda_1(l)}+\cdots+w_{l,\lambda_k(l)}\,u_{\lambda_k(l)}-u_l\right)^2\to\min_{w_{l,\lambda}}\\&\text{with constraints}\quad u_1+\cdots+u_n=0\quad\text{and}\quad\text{covariance matrix is identity.}\end{aligned}$$

The objective is the same as in the first step, but now in low dimensions and with fixed weights. Thus, LLE tries to reconstruct the local linear structure from high dimensions in low dimensions. Without constraints choosing all low dimensional points to be zero would solve the minimization problem. The covariance constraint excludes such trivial solutions by requiring that featurewise variance is 1. Thus, solutions have to be scattered in space to some extent. Covariance of zero prevents some other trivial solutions and avoids solution non-uniqueness due to rotations. Non-uniqueness due to translations is avoided by the mean zero contraint.

The solution to the minimization problem can be obtained from an eigenvalue problem similar to kernel PCA.

11.3 LLE with Scikit-Learn

Scikit-Learn has the `LocallyLinearEmbedding`[39] class in the `manifold` module.

```
import numpy as np
import sklearn.manifold as manifold
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

```
data_files = ['omega.npz', 'sphere.npz', 'cube.npz', 'clouds.npz']

for file in data_files:

    loaded = np.load(file)
    x = loaded['x']
    y = loaded['y']
    z = loaded['z']
    red = loaded['red']
    green = loaded['green']
    blue = loaded['blue']

    lle = manifold.LocallyLinearEmbedding(n_components=2, n_neighbors=30)
    U = lle.fit_transform(np.stack((x, y, z), axis=1))

    fig = make_subplots(rows=1, cols=2, specs=[[{'type': 'scatter3d'}, {'type':
↪'xy'}]])
    fig.update_layout(width=1000, height=600, scene_aspectmode='cube')
    fig.add_trace(go.Scatter3d(
        x=x, y=y, z=z,
        mode='markers',
        marker={'size': 1.5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,
↪ green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=1)
    fig.add_trace(go.Scatter(
        x=U[:, 0], y=U[:, 1],
        mode='markers',
        marker={'size': 5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=2)
    fig.show()
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[39] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.LocallyLinearEmbedding.html

CHAPTER

TWELVE

STOCHASTIC NEIGHBOR EMBEDDING

Up to now we only considered deterministic dimensionality reduction methods. Stochastic neighbor embedding (SNE) does not use neighborhood relations and distances directly. Instead it estimates the probability that two samples are neighbors in high dimensions from pairwise distances. Then it tries to find points in low dimensions which yield identical neighborhood probabilities.

SNE appeared in 2002 and several variants have been developed since then. The most prominent one is known as t-SNE and originated in 2008. Like MDS, SNE does not require direct knowledge of the underlying data set $x_1,\ldots,x_n$, but only uses pairwise distances.

Related projects:

- *MNIST Character Recognition* (page 117)
  - *t-SNE for QMNIST* (page 122)

12.1 Basic idea

12.1.1 Probabilities in High Dimensions

Given pairwise distances $D_{l,\lambda}$ $(l,\lambda=1,\ldots,n)$ in high dimensions we may fix a sample x_l and assign to all other samples probabilities $\tilde{p}_{l,\lambda}$ reflecting the neighborhood relations to x_λ. The closer x_λ to x_l the higher $\tilde{p}_{l,\lambda}$. We may use Gaussian probabilities:

$$\tilde{p}_{l,\lambda}:=\frac{\mathrm{e}^{\frac{-(x_l-x_\lambda)^2}{2\,\sigma_l^2}}}{\sum\limits_{\substack{i=1\\i\neq l}}^{n}\mathrm{e}^{\frac{-(x_l-x_i)^2}{2\,\sigma_l^2}}},\quad \lambda\neq l\qquad\text{and}\qquad \tilde{p}_{l,l}:=0.$$

The parameter σ_l is chosen numerically (by bisection, for instance) to fix the entropy of the neighborhood distribution of x_l at some prescribed value, which is independent of l. The entropy here is

$$-\sum_{\lambda=1}^{n}\tilde{p}_{l,\lambda}\,\log\tilde{p}_{l,\lambda}.$$

If σ_l is too small there will be only few neighbors of x_l with high probabilities. Then entropy is very low. If σ_l is too large many neighbors of x_l will have similar probabilities. Then entropy is high. Adjusting σ_l to get some prescibed medium entropy ensures that the probability distribution for the neighbors takes the data set's local density into account.

In general $\tilde{p}_{l,\lambda}\neq\tilde{p}_{\lambda,l}$. To enforce symmetry (which simplifies some computations) we set

$$p_{l,\lambda}:=\frac{\tilde{p}_{l,\lambda}+\tilde{p}_{\lambda,l}}{2\,n}.$$

The sum of all these n^2 values equals 1. Instead of n probability distributions (one for each l) we now only have one distribution and this distribution is symmetric.

All in all we converted pairwise distances to pairwise probabilities that two samples are neighbors. But the conversions is not direct by proportionality, but also takes local density of the data set into account.

12.1.2 Probabilities in Low Dimensions

Given points $u_1, \ldots, u_n$ in low dimensions we may use the same construction as in high dimensions to obtain probabilities $q_{l,\lambda}$. Distances are Euclidean and σ-values can be set to one to get uniform local densities in low dimensions.

There exist different involved reasons to choose non-Gaussian probabilities in low dimensions. Several choices have been proposed yielding a range of different SNE variants. Below we give the details for a variant known as t-SNE.

12.1.3 Fitting Probabilites

SNE tries to find low dimensional points $u_1, \ldots, u_n$ such that the corresponding probability distribution fits the high dimensional probability distribution as good as possible. Instead of using MSE of both sets of probability values SNE prefers the Kullback-Leibler divergence[40]:

$$\sum_{l=2}^{n}\sum_{\lambda=1}^{l-1} p_{l,\lambda}\,\log\frac{p_{l,\lambda}}{q_{l,\lambda}(u_1,\ldots,u_n)} \to \min_{u_1,\ldots,u_n}.$$

This minimization problem can be solved numerically via gradient descent. There also exist some more efficient methods adapted to the specifics of SNE.

12.2 t-SNE

The t-SNE variant of SNE defines low dimensional probabilites $q_{l,\lambda}$ based on the Student's t-distribution[41]. There are two reasons for this choice:

- It's computationally more efficient because no exponentiation is required.
- Student's t-distribution decays slower than a Gaussian distribution, which compensates (to some degree) for effects caused by the curse of dimensionality. A ball around a sample in high dimensions has much higher volume than a same sized ball (same radius) in low dimensions. To get similar sample densities (neighbors per volume) in both high and low dimensions in low dimensions we have to assign higher probabilities to more distant samples than in high dimensions. Else the lower dimensional embedding would look much denser than the original data set and clusters may get indistinguishable.

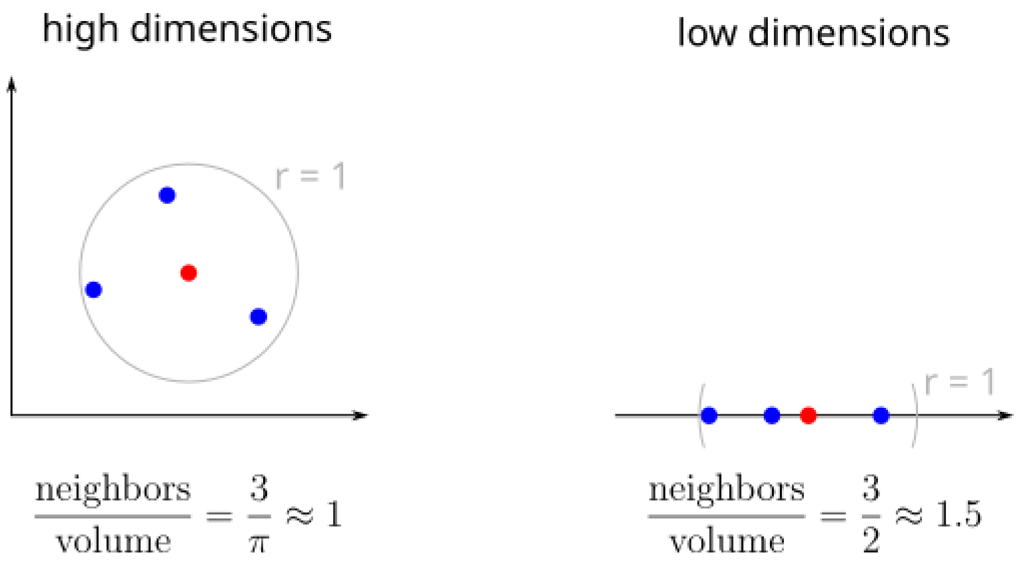

high dimensions
low dimensions
r = 1
r = 1
$\frac{\text{neighbors}}{\text{volume}} = \frac{3}{\pi} \approx 1$
$\frac{\text{neighbors}}{\text{volume}} = \frac{3}{2} \approx 1.5$

Fig. 12.1: Identical number of neighbors yields higher neighbor density in low dimensions than in high dimensions.

[40] https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence
[41] https://en.wikipedia.org/wiki/Student%27s_t-distribution

12.3 t-SNE with Scikit-Learn

Scikit-Learn has the `TSNE`[42] class in the `manifold` module. The `perplexity` parameter controls the desired entropy. Entropy is the base 2 logarithm of perplexity.

```
import numpy as np
import sklearn.manifold as manifold
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

```
data_files = ['omega.npz', 'sphere.npz', 'cube.npz', 'clouds.npz']

for file in data_files:

    loaded = np.load(file)
    x = loaded['x']
    y = loaded['y']
    z = loaded['z']
    red = loaded['red']
    green = loaded['green']
    blue = loaded['blue']

    sne = manifold.TSNE(n_components=2, perplexity=30)
    U = sne.fit_transform(np.stack((x, y, z), axis=1))

    fig = make_subplots(rows=1, cols=2, specs=[[{'type': 'scatter3d'}, {'type':
↪'xy'}]])
    fig.update_layout(width=1000, height=600, scene_aspectmode='cube')
    fig.add_trace(go.Scatter3d(
        x=x, y=y, z=z,
        mode='markers',
        marker={'size': 1.5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,
↪ green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=1)
    fig.add_trace(go.Scatter(
        x=U[:, 0], y=U[:, 1],
        mode='markers',
        marker={'size': 5, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
        hoverinfo = 'none',
        showlegend=False
    ), row=1, col=2)
    fig.show()
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

[42] https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html

CHAPTER

THIRTEEN

SELF-ORGANIZING MAPS

Self-organizing maps (SOMs), also known as Kohonen networks, appeared in the 1980s. In contrast to all other dimensionality reduction techniques discussed up to now, SOMs drop the requirement that the low dimensional version of a data set has to have as many points as the original data set. Instead, SOMs allow to choose the number of points in low dimensions and then try to find a topology preserving mapping between low and high dimensional data. Here, preservation of topology means preservation of neighborhood relations. Neighboring points in low dimensions are mapped to neighboring points in high dimensions.

A motivation for SOMs can be found in biology. Like artificial neural networks they are inspired by insights into the human brain, see lateral inhibition[43] for details. For this reason some people classify SOMs as a kind of ANNs, but structure and training differ a lot from usual ANNs.

13.1 SOM Structure

Let p be the dimension of the low dimensional space (almost always $p = 2$, sometimes $p = 1$ or $p = 3$) and let q be the desired number of data points $u_1, \ldots, u_q \in \mathbb{R}^p$ in low dimensions. Assign fixed locations to all low dimensional points in $\mathbb{R}^p$. Usually, the $u_1, \ldots, u_q$ are arranged in a regular rectangular or hexagonal grid. These positions will never change.

A SOM maps each u_i into the high dimensional data space by assigning a corresponding weight $w_i \in \mathbb{R}^m$. Although the images of the u_i are called weight, the w_i simply are points in the high dimensional data space. The aim of SOM training is to choose weights scattered over the whole high dimensional data set, but preserving the low dimensional neighborhood relations.

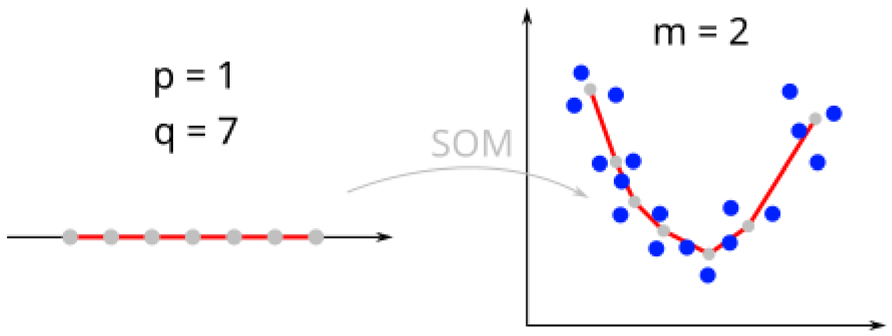


Fig. 13.1: SOMS approxiate high dimensional data by a low dimensional grid of points.

The u_i do not contain any information about the data set. To visualize a SOM we have to visualize properties of the weights w_i in the regular grid defined by the u_i (see below).

[43] https://en.wikipedia.org/wiki/Lateral_inhibition

13.2 Training

Fitting weights $w_1, \ldots, w_q$ to the data set $x_1, \ldots, x_n$ is done iteratively:

- Choose random initial weights (or create a grid in the p dimensional PCA manifold).
- Iterate over all samples:
  - Find the w_i closest to the current sample.
  - Move the best matching w_i and its neighbors closer to the current sample.
- Stop iteration if change of weights is small.

Moving weights towards the current sample is done via a neighborhood function $h : \mathbb{R}^p \to \mathbb{R}$, which typically takes values in $[0, 1]$ but in some cases may attain small negative values, too. Examples are Gaussian bells or the Mexican hat[44]. If w_i is the best matching weight for the current sample x_l, the update rule for all weights is

$$w_j^{\text{new}} := w_j^{\text{old}} + \alpha\, h\left(\frac{1}{r}\,(u_j - u_i)\right)\,(x_l - w_j^{\text{old}}).$$

The parameter $r > 0$ controls the size of the neighborhood. The parameter $\alpha \in (0, 1]$ controls the attracting force of x_l.

Training usually starts with high values for r and α to capture the data set's rough structure in few iterations. Then both values are decreased to allow for accurate fitting of the SOM to the data.

Batch training is possible, too. Here several samples are processed at once. For each sample the closest weight is determined. Then all weights are updated.

13.3 Prediction

With a trained SOM we may assign high dimensional data points to low dimensional grid points. This is also true for data not available during training. The chosen grid point is the one with weight closest to the sample under consideration.

Each low dimensional grid point may be regarded as a cluster of high dimensional points. The described mapping from high to low dimensions simply is 1-NN with the weights as training data.

13.4 Visualization

A straight forward SOM visualization is to color code a selected feature (component of the weights) in the low dimensional grid. This yields as many visualizations as there are dimensions in the data space.

To get some distance information from a SOM one may calculate Euclidean distances between weights of neighboring grid points and visualize those distances on the grid. The result is known as *unified distance matrix* (U-matrix). Dark (low values) areas indicate samples belonging to one cluster and light (high values) areas indicate boundaries between clusters. From the U-matrix one may extract clusters by applying standard image processing algorithms.

Many other kinds of visualizations may be useful, but details depend on the data set under consideration.

[44] https://en.wikipedia.org/wiki/Ricker_wavelet

13.5 SOMs with Python

Scikit-Learn does not support SOMs. But there are lots of Python modules for SOMs, for instance

- `sklearn-som`[45],
- `MiniSom`[46],
- `SOMPY`[47],
- `SimpSOM`[48],
- `somoclu`[49].

The first one focuses on clustering and lacks some more general features like direct access to the weights. `MiniSam` and `SOMPY` come without documentation. Usage has to be deduced from provided code examples. `SimpSOM` has at least some basic documention, but implementation is incomplete. The last one, `somoclu`, is a Python wrapper for an advanced stand-alone SOM software. It's well documented, so we use `somoclu` here. Implementation details may be found in somoclu: An Efficient Parallel Library for Self-Organizing Maps[50].

```
import numpy as np
import matplotlib.pyplot as plt
import plotly.graph_objects as go

import somoclu
```

```
file = 'clouds.npz'

loaded = np.load(file)
x = loaded['x']
y = loaded['y']
z = loaded['z']
red = loaded['red']
green = loaded['green']
blue = loaded['blue']

fig = go.Figure(layout_width=800, layout_height=600, layout_scene_aspectmode='cube
 ↪')
fig.add_trace(go.Scatter3d(
    x=x, y=y, z=z,
    mode='markers',
    marker={'size': 2, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
 ↪green, blue)]},
    hoverinfo = 'none'
))
fig.show()
```

```
<IPython.core.display.HTML object>
```

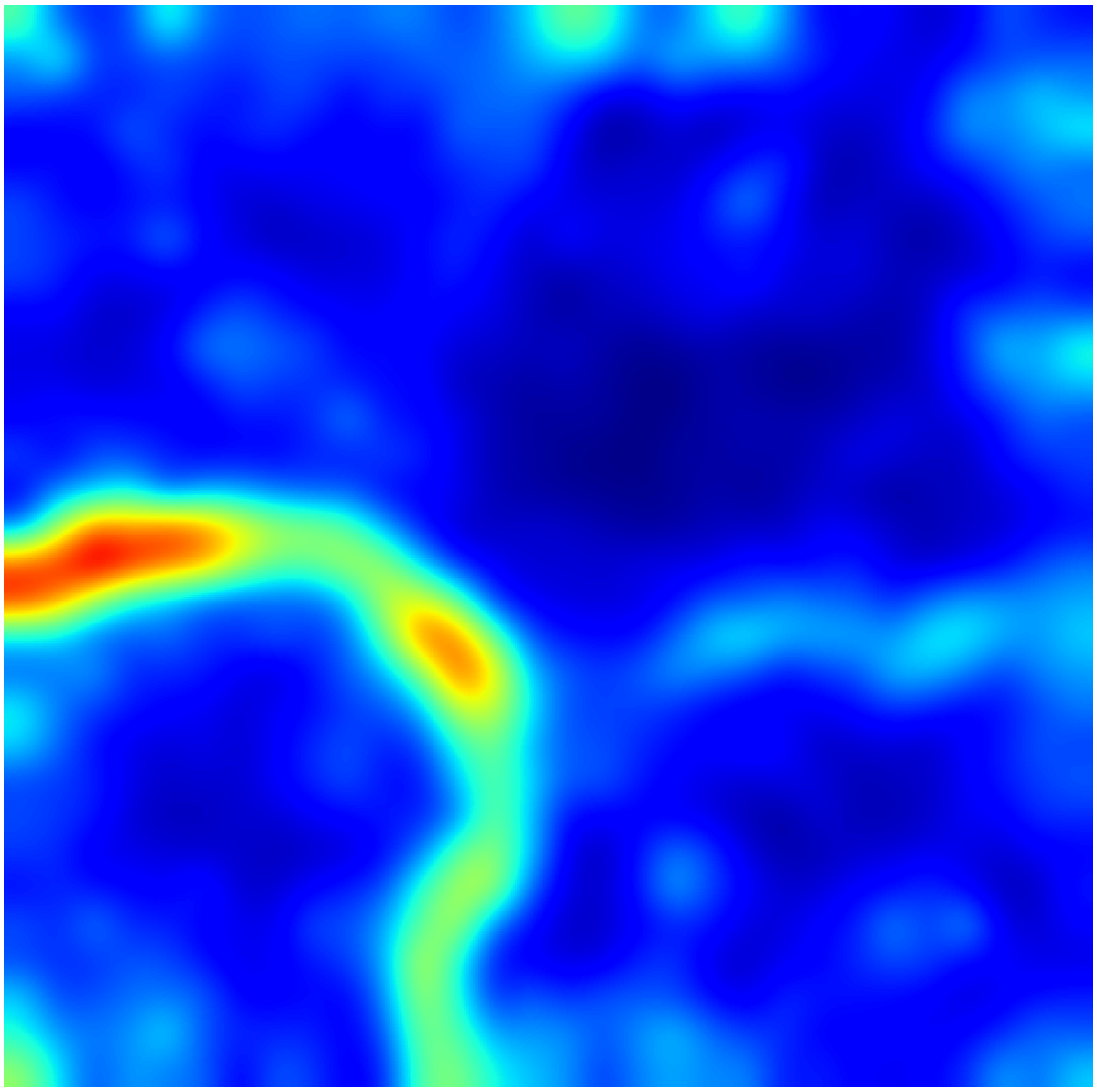
```
cols = 30
rows = 30

som = somoclu.Somoclu(cols, rows, initialization='pca')
som.train(np.stack((x, y, z), axis=1), epochs=1000)
```

[45] https://pypi.org/project/sklearn-som/
[46] https://github.com/JustGlowing/minisom
[47] https://github.com/sevamoo/SOMPY
[48] https://github.com/fcomitani/simpsom
[49] https://somoclu.readthedocs.io/en/stable/index.html
[50] https://www.jstatsoft.org/article/download/v078i09/1125

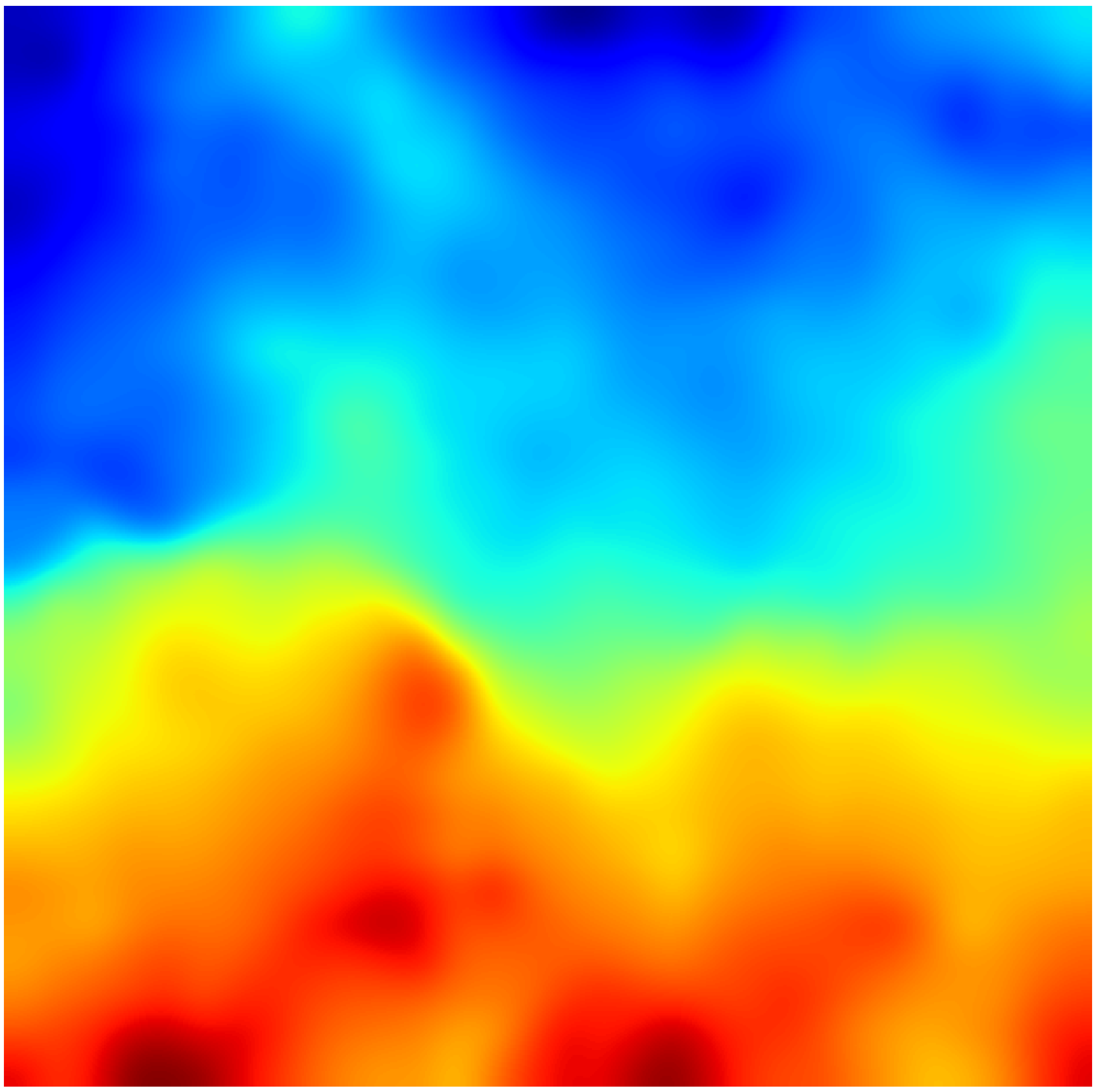
```
Warning: data was not float32. A 32-bit copy was made
```

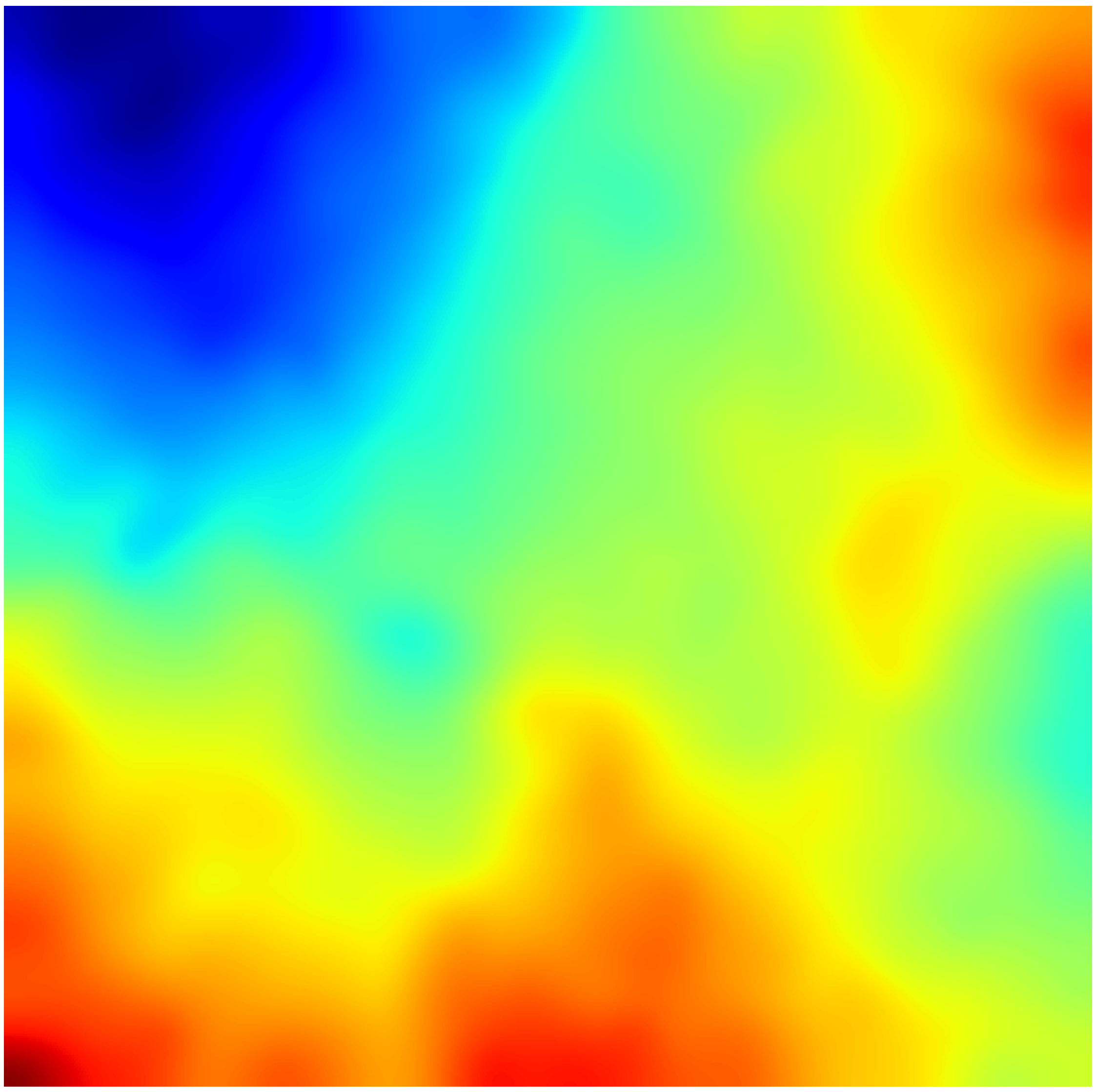
```
print('U-matrix')
som.view_umatrix(colormap='jet')

print('3x components of weight vectors')
som.view_component_planes(colormap='jet')
```

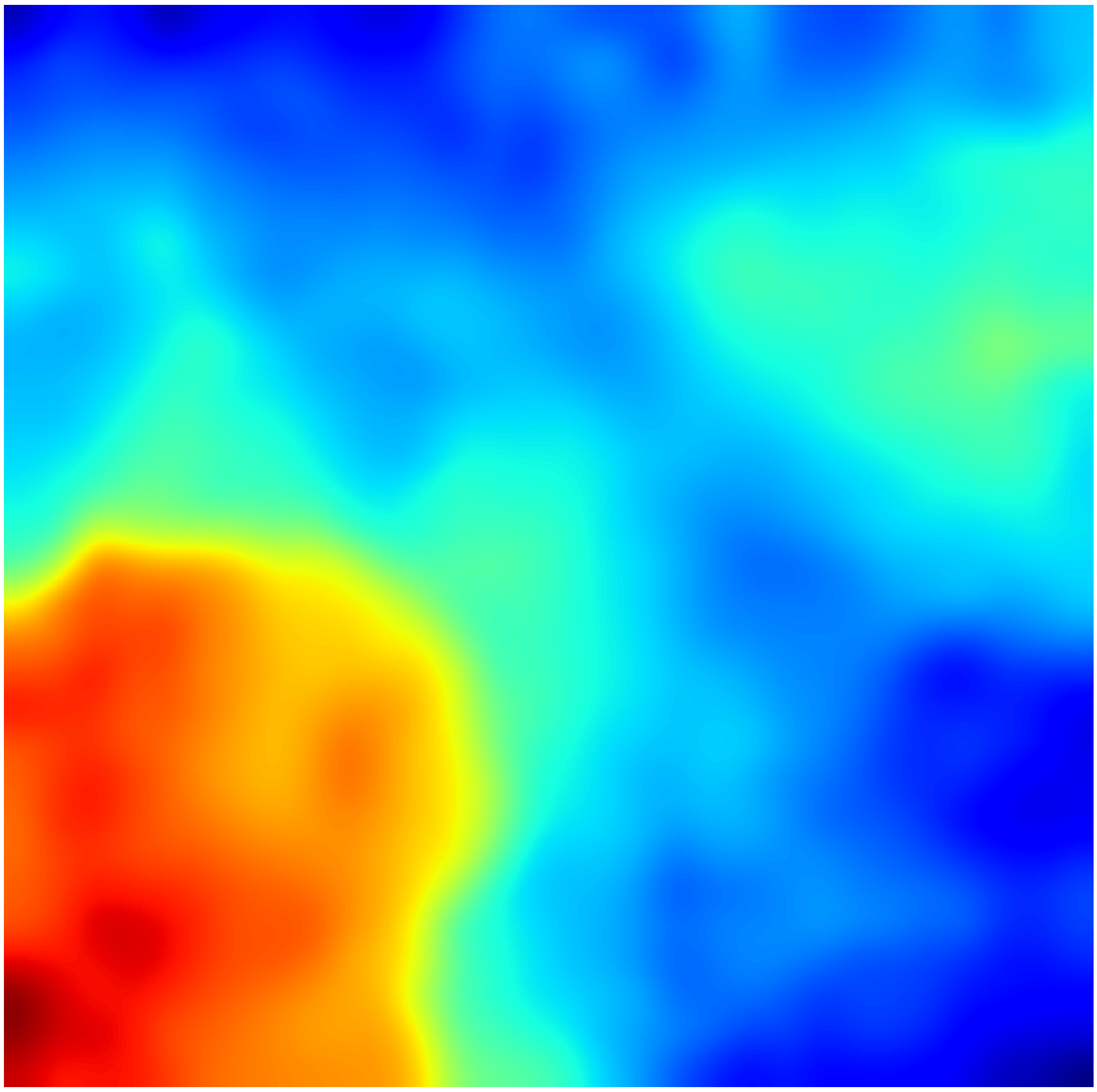
```
U-matrix
```



```
3x components of weight vectors
```





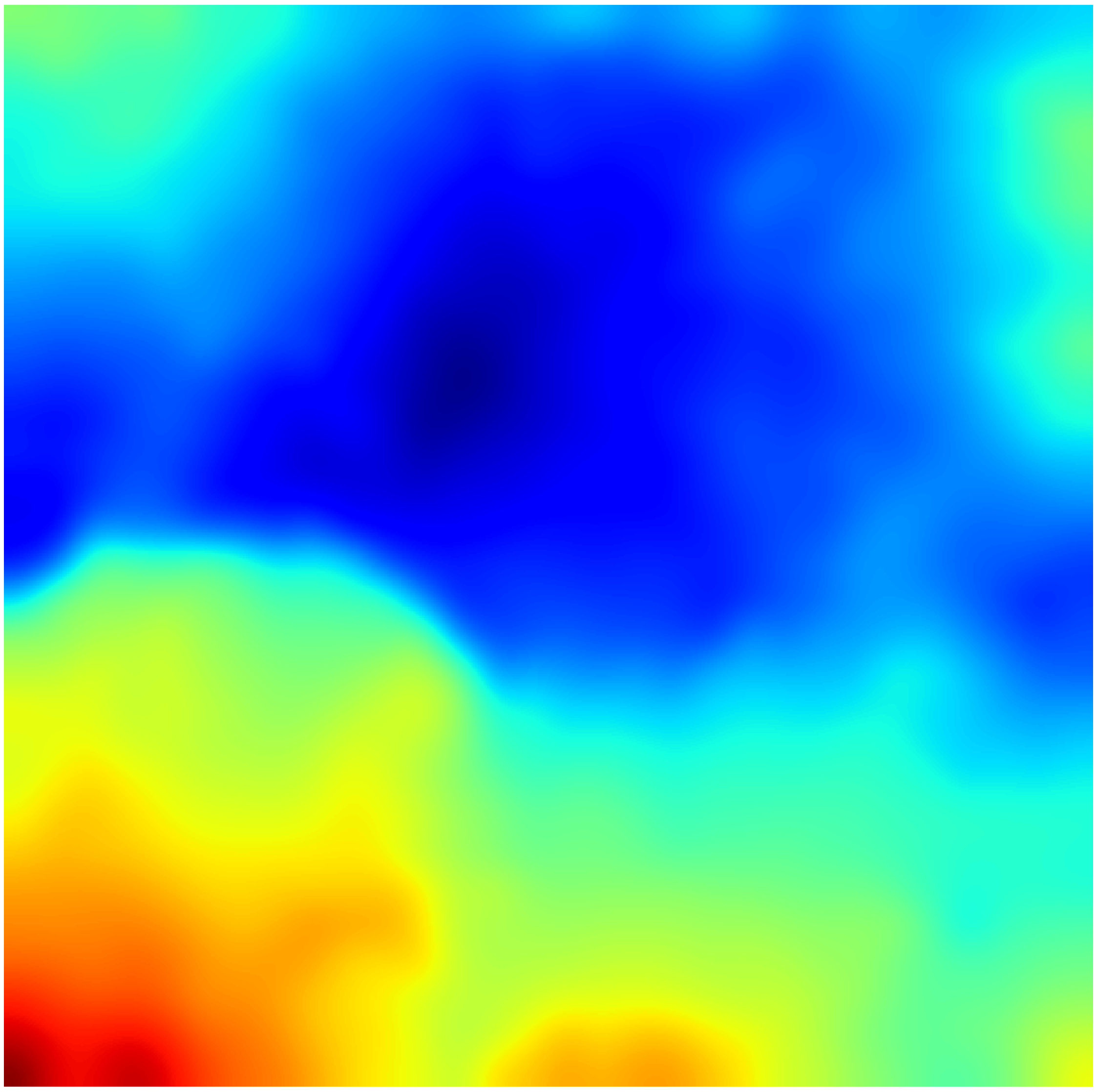


```
<module 'matplotlib.pyplot' from '/home/jef19jdw/anaconda3/envs/ds_book/lib/
↪python3.10/site-packages/matplotlib/pyplot.py'>
```

The `view_activation_map` computes the distance of a sample to all weights and visualizes corresponding matrix.

```
print('activation map')
som.view_activation_map(np.array([[x[700], y[700], z[700]]]), colormap='jet')
```

```
activation map
```



```
<module 'matplotlib.pyplot' from '/home/jef19jdw/anaconda3/envs/ds_book/lib/
↪python3.10/site-packages/matplotlib/pyplot.py'>
```

The `codebook` member variable contains the 3d NumPy array of the SOM's weights (rows x columns x features).

```
fig = go.Figure(layout_width=800, layout_height=600, layout_scene_aspectmode='cube
↪')

fig.add_trace(go.Scatter3d(
    x=x, y=y, z=z,
    mode='markers',
    marker={'size': 2, 'color': [f'rgb({r},{g},{b})' for r, g, b in zip(red,␣
↪green, blue)]},
    hoverinfo = 'none'
))


for r in range(0, rows):
    fig.add_trace(go.Scatter3d(
        x=som.codebook[r, :, 0], y=som.codebook[r, :, 1], z=som.codebook[r, :, 2],
        mode='lines',
        line={'color': 'rgb(0,0,0)'},
```

(continues on next page)

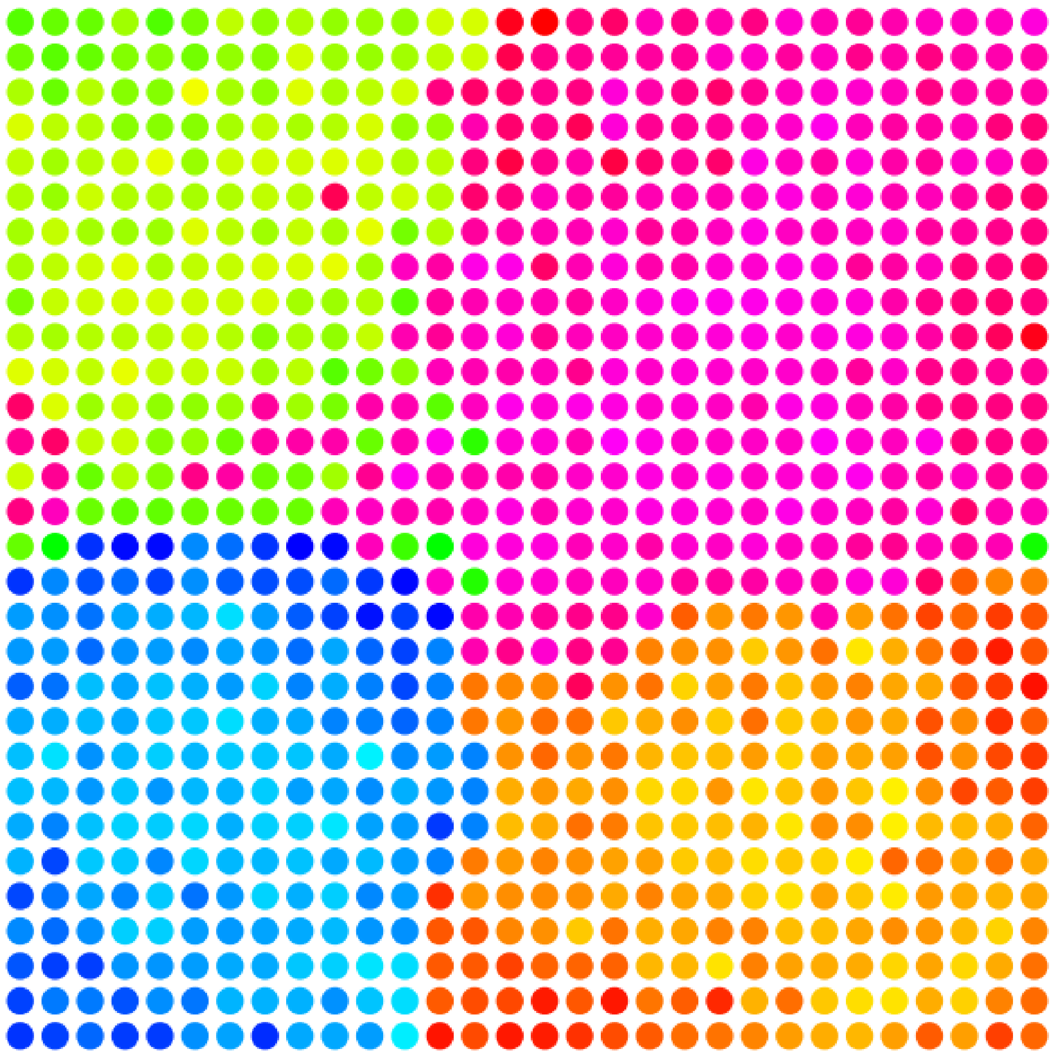
(continued from previous page)

```
        hoverinfo = 'none'
    ))
for c in range(0, cols):
    fig.add_trace(go.Scatter3d(
        x=som.codebook[:, c, 0], y=som.codebook[:, c, 1], z=som.codebook[:, c, 2],
        mode='lines',
        line={'color': 'rgb(0,0,0)'},
        hoverinfo = 'none'
    ))

fig.show()
```

```
<IPython.core.display.HTML object>
```

The `get_surface_state` methods yields the distances between each pair of weight vector and training sample. Return values is a 2d NumPy array (samples x weights). We may use it to visualize weights in 2d by assigning to each weight the closest training sample.

```
closest_sample = np.argmin(som.get_surface_state(), axis=0).reshape(rows, cols)

u1, u2 = np.meshgrid(np.linspace(0, 1, cols), np.linspace(1, 0, rows))
u1 = u1.reshape(-1)
u2 = u2.reshape(-1)

fig, ax = plt.subplots(figsize=(8, 8))
colors = np.stack((red[closest_sample].reshape(-1), green[closest_sample].
↪reshape(-1), blue[closest_sample].reshape(-1)), axis=1) / 255
ax.scatter(u1, u2, c=colors, s=100)
ax.axis('equal')
ax.axis('off')
plt.show()
```



Part II

Projects

CHAPTER

FOURTEEN

MNIST CHARACTER RECOGNITION

A major application of data science and artificial intelligence is recognition of handwritten characters. I a series of projects we will implement different techniques for this task based on the famous MNIST data set (and related data sets) for training recognition systems. MNIST is provided by the National Institute of Standards and Technology[51]

14.1 Semisupervised Classification

The QMNIST data set contains 120000 labeled images of handwritten digits. It was created to train digit recognition systems via supervised learning methods. Obtaining such a massiv amount of labeled samples is an expensive and time consuming work. The creators designed forms for writing prescribed digits, which were filled in by hundreds of people. Then the forms were scanned and digits separated. Hopefully every writer wrote the correct digit in each field.

In the project we want to train a digit recognition model based an QMNIST images without using any labels. Obtaining scanned images of handwritten digits is cheap and simple. One does not need forms with prescribed digits and nobody has to manually label scanned images. There are lots of pages with handwritten digits out there. We would have to scan them and separate digits by standard image processing routines.

Of course, having only unlabeled data at hand we cannot use supervised learning methods. Unsupervised methods do not yield labels, but only unlabeled clusters of similar images. The obvious idea is to do some unsupervised clustering and then manually label each cluster. Adding (a small amount of) manual labels to an unsupervised learning procedure is known as *semi-supervised learning*.

14.1.1 Preprocessing

Task: Load QMNIST training images (without labels). Center and crop images to 20x20.

Solution:

```
# your solution
```

We have a 400 dimensional data space. This might be t0o much to obtain useful results from k-means (curse of dimensionality). Maybe we have to use PCA.

Task: Check whether our data suffers from the curse of dimensionality (don't use all samples!). Why isn't this the case?

Solution:

[51] https://www.nist.gov

```
# your solution
```

14.1.2 Clustering

Although we know that there are 10 different classes there might be much more clusters. A cluster contains similar samples, but different people tend to write the same digits in several different ways. From this point of view it is not clear how many cluster we can expect.

We have to choose k by elbow method or silhouette score or Davies-Bouldin index. Calculating silhouette scores is very slow for large data sets (why?), so we calculate it only for a subset. We also should use mini-batch k-means to save computation time.

Task: Apply k-means to the data and find the best k based on elbow method, silhouette score (10000 samples) and Davies-Bouldin index. Make a first run with $k = 5, 10, 15, \ldots, 100$. Then choose a smaller intervall for k and run k-means for each k in this interval.

Solution:

```
# your solution
```

Task: Choose a good k and visualize all cluster centers together with cluster sizes (samples per cluster). Write a function for visualizing the cluster centers. The function shall take a NumPy array of cluster centers and a list of title strings as arguments.

Solution:

```
# your solution
```

14.1.3 Prediction

To use our model (set of cluster centers) for recognizing digits we have to label the cluster centers manually. For testing the prediction quality of our model we use QMNIST test images and labels.

Task: Load and preprocess QMNIST test images and labels.

Solution:

```
# your solution
```

Task: Create a mapping (1d array) from cluster labels (indices) to digit labels (manual labeling). Then label all test images and calculate correct classifiaction rate.

Solution:

```
# your solution
```

14.1.4 Inspection

Task: Calculate correct classification rate per cluster and show results together with corresponding cluster centroids.

Solution:

```
# your solution
```

14.1.5 More clusters?

Task: What do you think about using more clusters than suggested by silouette score and Davies-Bouldin-index?

Solution:

```
# your answer
```

Task: Try $k = 100$.

Task: What happens for $k = 60000$.

Solution:

```
# your answer
```

14.1.6 Random cluster centers

k-means with k manually labeled cluster centers is equivalent to 1NN with the cluster centers as training set. The idea behind k-means is that the cluster centers are not chosen at random, but much more sensible.

Task: Try 1NN with 100 randomly choosen and manually labeled training samples. Calculate correct classification rate on the test set.

Solution:

```
# your solution
```

14.2 Generating Handwritten Digits

Gaussian mixture models are generative models, that is, we may use a trained model to generate new samples looking similar to the training samples. If we train a Gaussian mixture model on QMNIST, then it should be possible to generate images of handwritten digits, which are not exactly equal to one of the QMNIST images.

14.2.1 Loading and Preprocessing Data

Task: Load QMNIST training images, labels and writer IDs. Center bounding boxes and crop images to 20x20 pixels.

Solution:

```
# your solution
```

Task: Use PCA with 50 components to reduce dimensionality of the data space. Else training the model will take too long. Show some image together with its projection onto the 50 dimensional subspace constructed by PCA.

Solution:

```
# your solution
```

14.2.2 Training the Model

Task: Fit a Gaussian mixture model to the data. What's a sensible number of clusters.

Solution:

```
# your solution
```

14.2.3 Generating Images

Task: Use GaussianMixture.sample[52] to generate 100 new images showing handwritten digits. Show all images.

Solution:

```
# your solution
```

14.2.4 Gaussian Mixture Unmixed

If we already know the clusters and only want to generate new images, we may fit a Gaussian distribution to each cluster manually. That is, for each cluster we compute mean vector and covariance matrix.

Task: Find the writer with the highest number of images available. Show all images for this writer.

Solution:

```
# your solution
```

Task: Get means and covariances for the ten classes of digitis written by the writer.

Solution:

```
# your solution
```

Task: Use NumPy's random number generator to generate 10 new images per class.

Solution:

```
# your solution
```

14.3 Autoencoder for QMNIST

When clustering QMNIST images with Gaussian mixtures we had to apply PCA to get acceptable computation times. Now we have a nonlinear dimensionality reduction technique at hand which might yield lower quality loss than PCA.

Task: Load QMNIST training images (without further preprocessing).

Solution:

```
# your solution
```

[52] https://scikit-learn.org/stable/modules/generated/sklearn.mixture.GaussianMixture.html#sklearn.mixture.GaussianMixture.sample

14.3.1 Autoencoder Training

We use the following encoder model:

```
# disable GPU if TensorFlow with GPU causes problems
#import os
#os.environ["CUDA_VISIBLE_DEVICES"] = "-1"

import tensorflow.keras as keras
```

```
encoder = keras.Sequential(name='encoder')

encoder.add(keras.Input(shape=(28, 28, 1)))
encoder.add(keras.layers.Conv2D(4, 3, activation='relu', name='conv1'))
encoder.add(keras.layers.Conv2D(4, 3, activation='relu', name='conv2'))
encoder.add(keras.layers.MaxPooling2D(name='pool1'))
encoder.add(keras.layers.Conv2D(8, 3, activation='relu', name='conv3'))
encoder.add(keras.layers.Conv2D(8, 3, activation='relu', name='conv4'))
encoder.add(keras.layers.MaxPooling2D(name='pool2'))
encoder.add(keras.layers.Conv2D(12, 3, activation='relu', name='conv5'))

encoder.summary()
```

Task: Create a symmetric decoder and the autoencoder model.

Solution:

```
# your solution
```

Task: Train the autoencoder.

Solution:

```
# your solution
```

14.3.2 Evaluation

Code space has 48 dimensions. So we could compare results to PCA with 48 components to see whether the autoencoder yields better results than PCA.

Task: Transform all images with PCA with 48 components. Visualize for some samples original image, autoencoder reconstruction and PCA transformed image. Compute RMSE for autoencoder and PCA results.

Solution:

```
# your solution
```

Visualizing codes provides some information on how well clusters can be separated by looking at the codes.

Task: Load training labels and visualize 300 codes for each digit.

Solution:

```
# your solution
```

14.3.3 Anomaly Detection

Looking at RMSE for each image we may identify images not similar to most other images. Removing such outliers from the data set could increase training success for supervised learning methods.

Task: Calculate RMSE for each image (autoencoder only) and plot images with highest RMSE.

Solution:

```
# your solution
```

14.3.4 Generating New Images

Task: Generate images from random codes.

Solution:

```
# your solution
```

Task: Generate images from random perturbations of a training image's code.

Solution:

```
# your solution
```

Task: Generate a transition from one training image to another training image

- by interpolating images directly and
- by interpolating their codes.

Solution:

```
# your solution
```

Task: Create an animation of the code based transition.

Solution:

```
# your solution
```

14.4 t-SNE for QMNIST

Many projects ago we used PCA to get 2d and 3d visualization of QMNIST training images. Now we have more powerful dimensionality reduction techniques at hand. Let's try t-SNE.

Task: Load QMNIST training images, center bounding boxes, and crop images to 20x20.

Solution:

```
# your solution
```

Task: Use t-SNE to get a 2d visualization of QMNIST images.

Solution:

```
# your solution
```

Task: Load QMNIST training labels and color the 2d plot according to the labels.

Solution:

```
# your solution
```

CHAPTER

FIFTEEN

HOUSE PRICES

This series of projects extends the results obtained in and .

15.1 House Prices SOM

We already used regression techniques to predict house prices. Now it's time to visualize the underlying data set using SOMs.

Task: Load the preprocessed German housing data set generated in . Convert categorical features to numeric features. For one hot encoding use as many code features as there are categories (do not drop one of them). Why?

```
# your solution
```

Task: Create a list of feature names, which we will use below to label plots. Create a NumPy array holding the data set.

```
# your solution
```

Task: Train a SOM on the data. Don't forget to standardize data.

```
# your solution
```

Task: Plot the U-matrix.

```
# your solution
```

Task: Visualize each (high dimensional) feature in 2d. Arrange all plots in a 4 by 6 grid.

```
# your solution
```

Task: Create a new sample and get its position (best matching unit) in the SOM. Show corresponding activation map and mark the best matching unit in the map.

```
# your solution
```

CHAPTER

SIXTEEN

SUPERMARKET CUSTOMERS

To understand general customer behavior and for targeted advertising supermarket companies have to identify groups of customers with similar behavior. Sending all customers identical advertisments fails most customers needs. Producing individual advertisements for each potential customer would be too expensive. Thus, clustering customers into a handful of groups should be a sensible middle ground.

There are several data sets for supermarkets available. We use the one provided at Michele Coscia's website[53]. It is sufficiently ridge and has simple structure. The data set is free to use (private communication with M. Coscia). Data comes from italian Coop supermarkets. In Explaining the Product Range Effect in Purchase Data[54] the data set is described in more detail.

16.1 Understanding the Data Set

Task: Get the data set. Read section III of the accompanying paper till the end of the left column on page 3. Answer the following questions:

- How many shops?
- How many customers?
- Are there customers missing in the data?
- Which time interval?
- What's the detail level of products?
- How many products?

Solution:

```
# your answer
```

Task: Load prices and purchases data.

Solution:

```
# your solution
```

Task: Collect following product information:

- number of shops selling the product,
- total quantity sold,
- number of customers who bought the product,
- maximum quantity bought by one customer.

[53] https://www.michelecoscia.com/?page_id=379
[54] https://www.michelecoscia.com/wp-content/uploads/2013/09/geocoop.pdf

Get minimum, average, maximum for all values.

Solution:

```
# your solution
```

Task: Collect following customer information:

- number of shops visited,
- total number of items bought,
- number of different products bought.

Get minimum, average, maximum for all values.

Solution:

```
# your solution
```

16.2 Cleaning the Data Set

The aim of this project is to cluster the set of customers into a handful of groups for targeted advertising. Outliers are not of interest because else the number of groups would become to large and advertising too expensive.

We only are interested in average customers and products. For example, products bought only by very few customers or not available in all shops should be removed from the data set. Customers buying only occasionally at the shops should be removed, too.

Task: Remove all products, customers, purchases such that remaining data satisfies the following conditions:

- each product has been sold in all shops,
- each product has been sold at least 1000 times,
- each product has been bought by at least 100 different customers,
- each product has been bought at least 4 times by at least one customer,
- each customer bought at least 10 items per month (on average),
- each customer bought at least 20 different products.

How many products, customers, purchases do we have now?

Solution:

```
# your solution
```

16.3 Preparing Data for Clustering

We want to use Scikit-Learn's k-means implementation for clustering. Thus, we have to bring our data into the right shape.

Task: Create a NumPy array with one row per customer and one column per product. Store quantities of products bought by each customer in the array. Sort customers descending with respect to the total number of items they bought and products descending with respect to the total quantity sold (sorting may simplify visualization later on).

Solution:

```
# your solution
```

16.4 Scaling

Scaling the data will influence the clustering process. We have several options:

- Without scaling data, products sold in large quantities will dominate the Euclidean distance between the product vectors of two customers. Thus, customers will have small distance if the products they bought most often do coincide.
- Standardization per product ensures that the total quantity sold of a product does not matter. The distance between two product vectors is the mean squared difference of per product quantities bought by both customers. Customers buying similar quantities of each product will have small distance.
- If we are more interested in the selection of products of each customer than in the quantities bought, we should normalize the product vectors. Then the total quantity bought by each customer is identical and data only contains information on the composition of each customers shopping cart. Here ℓ^1-norm should be used. Then all product quantities will sum to 1 and can be interpreted as probability that a product is bought by the customer.

Task: Prepare productwise standardized data and customerwise normalized data.

Solution:

```
# your solution
```

16.5 Clustering

Task: Cluster the data set with k-means for unscaled, standardized, and normalized data. Choose some good k for each variant. Keep the three `KMeans` objects with best k for further analysis.

Solution:

```
# your solution
```

16.6 Analyzing the Clusters

Now that we have identified groups of customers with similar behavior, it's time to understand those groups. Remember, that we want to adapt our advertising campaign to customer behavior.

Task: Visualize cluster centers in a quantity versus product index plot. Don't forget to back scale the data.

Solution:

```
# your solution
```

Each cluster center can be regarded as a prototype customer of the cluster.

Task: Characterize the prototype customers for unscaled and standardized data in words a person designing advertising campaigns understands.

Solution:

```
# your answer
```

The first two clusterings are more or less trivial and useless for targeted advertising. The third cluster deserves further investigation.

Task: Get the 100 most popular products (highest average per customer quantity) per cluster for the third clustering. By how many products differ the top 100 products? Do the same for the first clustering and for a random clustering.

Solution:

```
# your solution
```

Task: Consider the third clustering only. Does one of the customer groups buy higher quantities than the other? Visualize the answer to this question for different price regions.

Solution:

```
# your solution
```

CHAPTER

SEVENTEEN

CHINESE CELADONS

This series of projects investigates the relations between different types of Chinese celadons (early porcelains).

17.1 Hierarchical Clustering

We want to find clusters in a set of celadons (early porcelains). A data set with material properties of a number of celadons found in China is available from UCI Machine Learning Repository[55]. The data set originates from research work published in Data-driven research on chemical features of Jingdezhen and Longquan celadon by energy dispersive X-ray fluorescence[56] by Ziyang He, Maolin Zhang, Haozhe Zhang. A free preprint PDF file[57] is available, too.

Task: Read (at least) the last paragraph of section 1 and section 2 of the afore mentioned preprint. How many celadon sample do you expect in the data set after reading?

Solution:

```
# your answer
```

17.1.1 Understanding the data

Data comes as a CSV file.

Task: Load the data to a data frame. Why are there so many samples?

Solution:

```
# your solution
```

Task: Create a NumPy array holding the data with one row per sample and one column per feature.

Solution:

```
# your solution
```

Task: Create a list of sample names.

Solution:

```
# your solution
```

[55] https://archive.ics.uci.edu/ml/datasets/Chemical+Composition+of+Ceramic+Samples
[56] https://www.sciencedirect.com/science/article/pii/S0272884215023135
[57] https://arxiv.org/pdf/1511.07825.pdf

17.1.2 Preprocessing

Units of measurement are weight per cent for body features and parts per million for glaze features. Since all features are equally important for finding similar celadons we should standardize features independently. Maybe the assumption of equal importance is not correct, but without further domain knowledge we cannot do better.

Task: Standardize all features.

Solution:

```
# your solution
```

17.1.3 Hierarchical Clustering

Task: Plot a dendrogram and determine a sensible number of clusters.

Solution:

```
# your solution
```

Task: Cluster data into the chosen number of clusters.

Solution:

```
# your solution
```

17.1.4 k-Means Clustering

Task: Use k-means for clustering. Determine a good k.

Solution:

```
# your solution
```

Task: Compare results from hierarchical and k-means clustering.

Solution:

```
# your solution
```

17.2 Density-based Clustering

Task: Load, rearrange, standardize the celadons data set.

Solution:

```
# your solution
```

Task: Plot a histogram of pairwise distances to get a feeling for distances in the data set.

Solution:

```
# your solution
```

Task: Use DBSCAN algorithm for clustering. Print sample names for each cluster (and outliers).

Solution:

```
# your solution
```

Task: Use OPTICS algorithm for clustering. Choose clusters manually.

Solution:

```
# your solution
```

Task: Visualize the data set exploiting the tree structure generated by OPTICS algorithm. Label each node with the sample name.

Solution:

```
# your solution
```

CHAPTER

EIGHTEEN

COLOR PERCEPTION

Human color perception is not as trivial as it may look at first glance. The number of independent colors recognizable by humans, and thus the dimension of the human color space, is not obvious. In the project we want to use MDS to get some insights into human color perception.

The data set we want to analyze is from 1954 and can be found in a table on page 2 of Dimensions of Color Vision[58]. Corresponding CSV file ships with this project's notebook. It contains similarity scores for pairs of colors (light of different wave lengths).

Task: Read the following excerpt from Ekman's 1954 paper ('Table 1' is in the CSV file). How did Ekman obtaine the similarity scores?

Solution:

```
# your notes
```

Task: Load the data set. Create a list of wave lengths (for labeling plots below) and a distance (!) matrix.

Solution:

```
# your solution
```

18.1 1d Embedding

Wave lengths are real numbers and, thus, contained in a one dimensional linear manifold. If human perception of color differences is proportional to wave length, then the color space should have a nice embedding into 1d space.

Task: Use metric MDS to get a 1d embedding of the data set.

Solution:

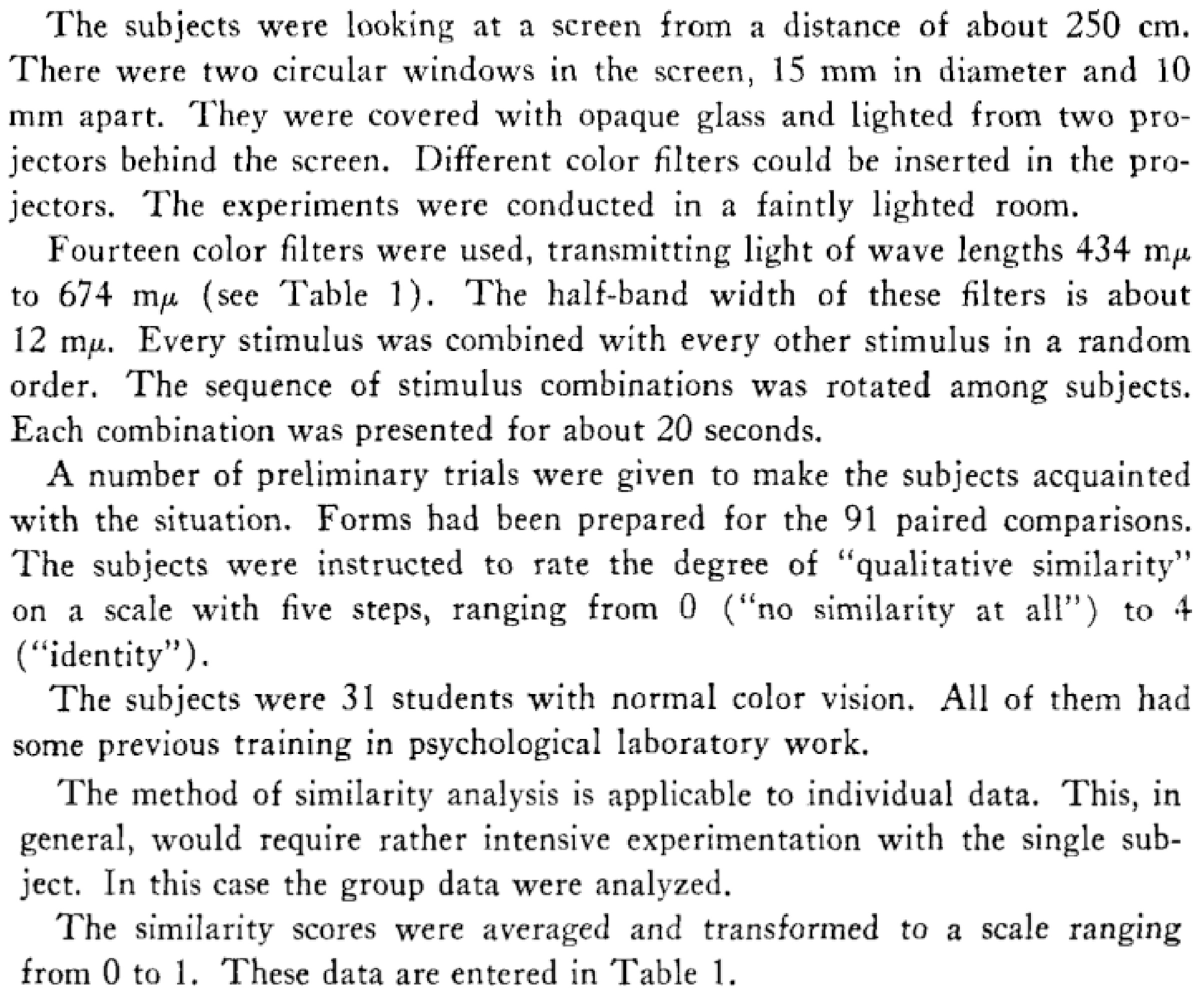
```
# your solution
```

Task: Get the distance matrix of the embedded data set. Visualize (dis)similarity of both distance matrices.

Solution:

```
# your solution
```

[58] https://doi.org/10.1080/00223980.1954.9712953



The subjects were looking at a screen from a distance of about 250 cm. There were two circular windows in the screen, 15 mm in diameter and 10 mm apart. They were covered with opaque glass and lighted from two projectors behind the screen. Different color filters could be inserted in the projectors. The experiments were conducted in a faintly lighted room.

Fourteen color filters were used, transmitting light of wave lengths 434 mμ to 674 mμ (see Table 1). The half-band width of these filters is about 12 mμ. Every stimulus was combined with every other stimulus in a random order. The sequence of stimulus combinations was rotated among subjects. Each combination was presented for about 20 seconds.

A number of preliminary trials were given to make the subjects acquainted with the situation. Forms had been prepared for the 91 paired comparisons. The subjects were instructed to rate the degree of "qualitative similarity" on a scale with five steps, ranging from 0 ("no similarity at all") to 4 ("identity").

The subjects were 31 students with normal color vision. All of them had some previous training in psychological laboratory work.

The method of similarity analysis is applicable to individual data. This, in general, would require rather intensive experimentation with the single subject. In this case the group data were analyzed.

The similarity scores were averaged and transformed to a scale ranging from 0 to 1. These data are entered in Table 1.

Fig. 18.1: Data table in Ekman's 1954 paper. The opposite of 'big data'.

18.2 2d Embedding

Task: Repeat steps from above for a 2d embedding.

Solution:

```
# your solution
```

18.3 3d Embedding

Task: Repeat steps from above for a 3d embedding.

```
# your solution
```

18.4 Higher Dimensions

Task: Use PCA to determine the number of dimensions needed to accurately model the space of human perceivable colors.

Solution:

```
# your solution
```

CHAPTER

NINETEEN

FOREST FIRES

In this project we want to obtain some insight into different types of forest fires in a Portuguese national park. Data is available at the Website of Paulo Cortez[59]. It covers forest fires from January 2000 till December 2003 and provides several numerical features of soil moisture and weather.

Task: Get the data and have a look at Data Mining Approach to Predict Forest Firesusing Meteorological Data[60]. What are FFMC, DMC, DC, ISI?

Solution:

```
# your answer
```

Task: Load the data. Remove outliers. Scale data where necessary. Create a NumPy array containing for all samples FFMC, DMC, DC, ISI.

Solution:

```
# your solution
```

Task: Visualize data with 2d Isomap. Interpret the 2d embedding. Use visualizations of all the other features (not only FFMC, DMC, DC, ISI). Can you identify clusters or any other useful structure? Describe different types of forest fires.

Solution:

```
# your solution
```

Task: Use PCA to project data into 2 dimensions. Can you see different fire types here, too?

Solution:

```
# your solution
```

[59] http://www3.dsi.uminho.pt/pcortez/forestfires/
[60] https://core.ac.uk/download/pdf/55609027.pdf