Two-Dimensional Data Embeddings

In this chapter, we discuss methods that embed instances into a two-dimensional space such that the geometric relationships between points reflect as faithfully as possible the relationships between instances in the original space. Such embeddings are used primarily to visualize the structure of data for the purpose of discovering interesting patterns and groups, as well as for ranking and discovering relationships between individual instances from the training set.

Principal component analysis, which we discussed in the previous chapter, is not an embedding in the same sense as the methods considered in this chapter. The principal component method is a projection method that constructs a mathematical mapping, say U (a matrix composed of the directional vectors of the principal components u_1 and u_2), which projects the data X into two dimensions, that is, $Z = XU$. A projection is generally not the same as an embedding. In PCA, we seek a mapping $f(x)$ that projects the data into a new coordinate system, whereas in the methods discussed in this chapter, we seek directly a placement of points $y_1, \ldots, y_n \in \mathbb{R}^2$, that preserves the selected relationships between instances as faithfully as possible. Therefore, such methods generally do not provide a simple mapping for new data into the same space.

The notation y_i here does not represent a class label (as in predictive models), but rather the coordinates of the embedded point corresponding to instance x_i, i.e., the position of this instance in the two-dimensional space.

We will examine three approaches to two-dimensional data embeddings: multidimensional scaling (MDS), the t-SNE method, and force-directed embeddings. The first is important from a historical perspective, the second emphasizes the local structure of the data, and the third originates from network visualization. For small datasets, the embeddings produced by MDS, t-SNE, and force-directed methods may be similar, whereas for larger and high-dimensional data, the differences between the approaches become much more pronounced.

Common to all three described methods is the basic approach of embedding instances into two dimensions. We will use an objective function derived from distances between instances in the original space and relate them appropriately to the positions of points in the embedding space. The initial embedding is obtained by placing the

points (instances) randomly in the plane, after which their positions are gradually adjusted in the direction of the gradient of the objective function with respect to the coordinates of these points. Thus, this time the model is not represented by weights or mapping parameters, but rather by the coordinates of the points in the embeddings themselves, so the number of parameters is equal to $2n$, where n is the number of instances.

Data

Let us begin with the data.

All the methods that we will present in this chapter can be based on distances or similarities between instances. We will assume that these relationships are already given. Thus, this time we will not describe instances explicitly by attributes, as we were accustomed to in previous chapters, but solely by the distances between them. These distances would usually be written in a (symmetric) matrix, but they can also be represented by the dictionary below:

In this chapter, the data can be described solely by the distances between instances, without an explicit attribute-based representation. Such a description, however, is not a direct input for PCA, since this method requires a representation of instances using attributes.

```
distances = {
    ("Novo Mesto", "Maribor"): 170,
    ("Novo Mesto", "Celje"): 83,
    ("Novo Mesto", "Koper"): 169,
    ("Novo Mesto", "Kranj"): 99,
    ("Novo Mesto", "Ljubljana"): 72,
    ("Novo Mesto", "Postojna"): 116,

    ("Maribor", "Celje"): 55,
    ("Maribor", "Koper"): 232,
    ("Maribor", "Kranj"): 156,
    ("Maribor", "Ljubljana"): 128,
    ("Maribor", "Postojna"): 178,

    ("Celje", "Koper"): 183,
    ("Celje", "Kranj"): 105,
    ("Celje", "Ljubljana"): 77,
    ("Celje", "Postojna"): 130,

    ("Koper", "Kranj"): 128,
    ("Koper", "Ljubljana"): 107,
    ("Koper", "Postojna"): 58,

    ("Kranj", "Ljubljana"): 30,
    ("Kranj", "Postojna"): 77,

    ("Ljubljana", "Postojna"): 53,
}
```

Multidimensional Scaling

Let us denote the distance between instances i and j by d_{ij}. In multidimensional scaling, we wish to arrange instances in a two-dimensional space such that each instance i is described by coordinates $y_i = (y_{i1}, y_{i2}) \in \mathbb{R}^2$. In this target space, the distance between instances should be Euclidean,

$$\|y_i - y_j\| = \sqrt{(y_{i1} - y_{j1})^2 + (y_{i2} - y_{j2})^2},$$

that is, the same as we would obtain in an ordinary scatter plot.

Multidimensional scaling (MDS) was developed by Torgerson (1952), Shepard (1962), and Kruskal (1964) within the field of psychometrics, a discipline concerned with measuring psychological properties (e.g., perceptions, similarities between stimuli, and preferences) using statistical methods.

The goal of multidimensional scaling is to preserve the given distances as faithfully as possible, meaning that the distances $\|y_i - y_j\|$ in the embedding space should be as close as possible to the original distances d_{ij}. Accordingly, multidimensional scaling minimizes the objective function

$$J(Y) = \sum_{i<j} \left(\|y_i - y_j\| - d_{ij} \right)^2,$$

where $Y = \{y_1, \ldots, y_n\}$ represents the sought placement of points in the plane.

The term *scaling* refers to arranging objects on a scale such that the distances between them reflect their mutual differences or similarities. The adjective *multidimensional* emphasizes that this scale is not formed in only one dimension, but rather in a multidimensional space, most often in two or three dimensions.

The Euclidean distance in the embedding space therefore depends entirely on the positions of the points representing the instances; these positions are also the parameters of our model. We will determine them using gradient descent, employing automatic differentiation. We already developed the library for automatic differentiation and learning with gradient descent in the previous chapters, so here we provide only the code for the class representing the multidimensional scaling model.

```
class MDS:
    def __init__(self, items):
        random.seed(100)
        self.pos = {i: [Value(random.uniform(-1, 1)) for _ in range(2)]
                    for i in sorted(items)}

    def parameters(self):
        return [p for pos in self.pos.values() for p in pos]

    def d(self, a, b):
        return sum([(ai - bi) ** 2 for ai, bi in \
                    zip(self.pos[a], self.pos[b])]) ** 0.5

    def loss(self, xs, ys):
        n = len(xs)
        return sum((self.d(*pair) - y) ** 2 \
                   for pair, y in zip(xs, ys)) / Value(n)
```

The class MDS constructs the computational graph of the objective function for multidimensional scaling for the given input data (distances). Each instance is assigned a random initial position in the plane (self.pos), and these coordinates represent the parameters of the model. The method d computes the Euclidean distance between two instances in the current embedding, while loss measures the discrepancy between these distances and the given distances d_{ij}.

Using gradient descent, we then adjust the positions of the points so that this error is reduced as much as possible:

```
items = sorted(list(set(i for pair in distances.keys() for i in pair)))
pairs = list(distances.keys())
targets = [distances[p] for p in pairs]

model = MDS(items)
model = train(model, pairs, targets, n_epochs=1000,
              learning_rate=0.1, report_every=100)
```

There are only a few cities in our training set, and the optimization converges quickly. Instead of printing the positions of individual cities, it is of course more appropriate here to visualize the data map:

```
plt.figure(figsize=(4, 4))
for city in sorted(items):
    x, y = model(city)
    plt.scatter(x.data, y.data, color="k", s=10)
    plt.annotate(
        city,
        (x.data, y.data),
        xytext=(0, 4),
        textcoords="offset points",
        ha="center",
        va="bottom",
    )
plt.savefig("mds-cities.pdf")
```

The map of cities is shown in Figure 30. It is somewhat rotated and upside down, but who says that north must be at the top and that we must look from above. The relationships between the cities are nevertheless quite accurate. Maribor, Celje, Ljubljana, Postojna, and Koper lie on the main diagonal in the correct order, Kranj is slightly off to the side, and Novo mesto is on the other side. The input data contained road distances between cities, which are not the same as straight-line distances; the Euclidean distance in the embedding space, however, corresponds to straight-line distances. This is probably the main source of some minor inaccuracies, but in general we successfully reconstructed the positions of Slovenian cities from road distances.

For easier comparison with the geographic maps we are accustomed to, we could rotate our map of Slovenian cities by 180 degrees.

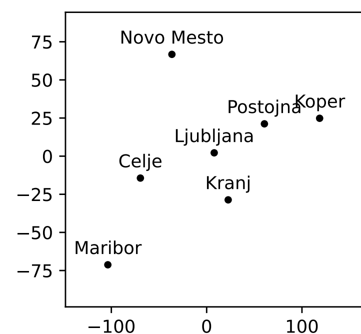

Figure 30: Two-dimensional embedding using multidimensional scaling.

A warning, however: there are infinitely many MDS embeddings with equally good values of the objective function $J(Y)$. Without changing the value of the objective function, embeddings can be arbitrarily rotated and reflected. All these transformations preserve distances, and in fact only distances appear in the objective function. The resulting axes therefore have no intrinsic meaning and cannot be associated with any interpretation. Unlike in PCA, that is. But recall: in PCA we start from instances represented by attributes, whereas in MDS, we start from distances.

Neighbor Embeddings

Multidimensional scaling has a hidden problem: the objective function squares the difference between distances in the input data and those in the embedding space. Learning the model by gradient descent will therefore "focus more" on reducing larger deviations. If deviations are measured proportionally, for example in percentages, and if the distances in the embedding deviate from the input distances by 10%, then the deviation between Ljubljana and Kranj will have a much smaller impact on learning than the deviation between Koper and Maribor. MDS therefore tends to preserve primarily the larger distances. The influence of shorter distances between neighboring instances is much smaller.

Preserving larger distances is, of course, desirable if we wish to obtain data maps with appropriate global arrangements. However, in data analysis we are often primarily interested in neighborhoods, and in data maps we may only seek groups of data and attempt to interpret them, while global relationships, or groups that are far apart from one another, are of little interest. Thus, we are interested in visualizations where groups of mutually similar instances are emphasized, while the interpretation of distances between groups that are far apart is not important. Such a visualization is shown, for example, on the right-hand side of Figure 31, and is clearly very different from the visualization of the same data produced by MDS.

We are therefore interested in preserving neighborhoods. For this, we need a measure that indicates how close instances are in the original space, and a measure that indicates how close they are in the embedding space. These measures must ignore instances that are far apart and focus only on nearby instances. Here we can use a trick: instead of distance, we estimate the probability that two instances are close, both in the original space and in the embedding space, and then use as the objective function a measure of similarity between these two probability vectors, where by vectors we mean the lists of proximity probabilities for all pairs of instances.

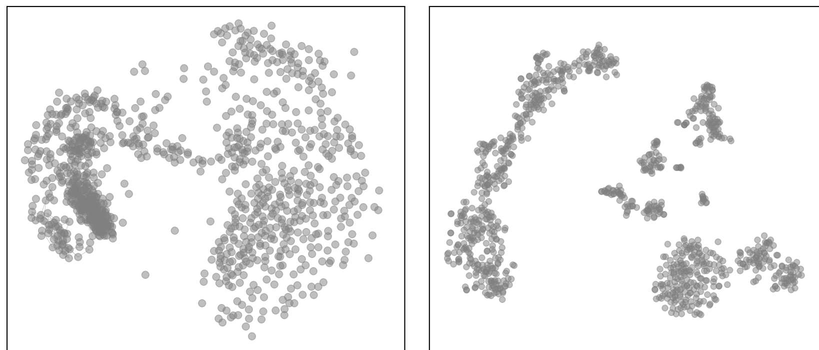

Figure 31: Comparison of MDS (left) and t-SNE (right) embeddings on gene expression data from bone marrow cells (a sample of one thousand cells and the expression levels of one thousand genes). Groups of points are expected to correspond to individual cell types.

Let us approach this more formally. As in MDS, let x_i denote an instance in the original space and y_i its representation in the embedding. For each pair of instances (i, j), we define similarity in the original space as the conditional probability

$$p_{j|i} = \frac{\exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2}\right)}{\sum_{k \neq i} \exp\left(-\frac{\|x_i - x_k\|^2}{2\sigma_i^2}\right)}.$$

This probability indicates how likely we are to choose instance j as a neighbor of instance i. The parameter σ_i determines the "width" of the neighborhood around x_i. Since the conditional probabilities $p_{j|i}$ are generally not symmetric, meaning that in general $p_{j|i} \neq p_{i|j}$, in t-SNE we combine them into a symmetric joint probability

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n},$$

where n is the total number of instances.

In the embedding space, we similarly define neighborhood probabilities, but use a distribution with heavier tails (Student's t-distribution with one degree of freedom):

$$q_{ij} = \frac{\left(1 + \|y_i - y_j\|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \|y_k - y_l\|^2\right)^{-1}}.$$

This choice alleviates the crowding problem, since it allows larger distances in the embedding to be represented more distinctly than with a Gaussian distribution.

The goal of this approach is to find embeddings y_i such that the distributions p_{ij} and q_{ij} are as similar as possible. This is achieved by minimizing the Kullback-Leibler divergence:

$$\mathcal{L}(Y) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}.$$

This objective function penalizes primarily those cases where two points are close in the original space (large p_{ij}), but far apart in the embedding (small q_{ij}), thereby encouraging the preservation of the local structure of the data.

The objective function, as defined above, is used by the t-SNE method (*t-distributed Stochastic Neighbor Embedding*). This approach extends the older SNE method, which was likewise based on comparing probability distributions of neighborhood relationships between pairs of instances in the original space and the embedding space. However, SNE had several important shortcomings. First, it used asymmetric probabilities $p_{j|i}$ and $q_{j|i}$, which complicated optimization and interpretation. Second, because it also used a Gaussian distribution in the embedding space, it suffered from the so-called "crowding problem," where larger distances are difficult to represent appropriately in a low-dimensional space, causing points to cluster together excessively.

The SNE method was proposed by Geoffrey E. Hinton and Sam T. Roweis (2002), while t-SNE was introduced by Hinton and Laurens van der Maaten (2008). Hinton is also one of the fathers of modern artificial intelligence and received the Nobel Prize in 2024 for his research contributions.

The t-SNE method resolves the main shortcomings of SNE in two ways: it uses the symmetric similarity p_{ij} and introduces Student's t-distribution in the embedding space.

In the following, we will not implement the full standard version of the t-SNE method, but rather a simplified version that preserves the basic idea: nearby pairs in the original space should also have high probability in the embedding space, and the objective function should measure the difference between the two distributions.

```
class tSNE:
    def __init__(self, items, sigma=40.0, eps=1e-12):
        random.seed(100)
        self.pos = {
            i: [Value(random.uniform(-1, 1)) for _ in range(2)]
            for i in sorted(items)
        }
        self.sigma = sigma  # constant bandwidth of the Gaussian
        self.eps = eps  # parameter for numerical stability

    def parameters(self):
        return [p for pos in self.pos.values() for p in pos]

    def p_distribution(self, pairs, distances):
        unnorm = []
        for d in distances:
            pij = math.exp(-(d ** 2) / (2 * self.sigma ** 2))
            unnorm.append(pij)

        z = sum(unnorm) + self.eps
        return [p / z for p in unnorm]
```

```
    def sqdist(self, a, b):
        return sum((ai - bi) ** 2 \
                   for ai, bi in zip(self.pos[a], self.pos[b]))

    def q_distribution(self, pairs):
        numerators = []
        for a, b in pairs:
            numerators.append((Value(1.0) + self.sqdist(a, b)) ** -1)

        z = sum(numerators)
        return [q / z for q in numerators]

    def loss(self, pairs, distances):
        p = self.p_distribution(pairs, distances)
        q = self.q_distribution(pairs)

        loss = Value(0.0)
        for pij, qij in zip(p, q):
            loss = loss + Value(pij) * \
              (math.log(pij + self.eps) - (qij + self.eps).log())

        return loss
```

The class 'tSNE', in the constructor '__init__', first randomly initializes a two-dimensional embedding 'self.pos' for each instance. The method 'p_distribution' computes the similarity distribution p_{ij} from the given distances in the original space by converting each distance into a weight using a Gaussian function and then normalizing all weights into probabilities. For the embedding space, the method 'sqdist' computes the squared Euclidean distance, while 'q_distribution' uses this to compute the distribution q_{ij} from the current positions of the points. In doing so, it uses a kernel of the form

$$(1 + \|y_i - y_j\|^2)^{-1}.$$

We implemented the objective function in the method 'loss', which first determines the distributions 'p' and 'q', and then, to compute the Kullback-Leibler divergence, sums $p_{ij} \log(p_{ij}/q_{ij})$ over all pairs.

Here we use a simplified version of the t-SNE method. In the standard implementation, the width of the Gaussian kernel is not determined by a single parameter, but is instead adjusted separately for each point so that it corresponds to the chosen perplexity. Early exaggeration is also often used, where in the initial iterations we increase the values of p_{ij} and thereby emphasize the formation of local groups.

The use of the above implementation from this point onward is no different from that for MDS:

```
model = tSNE(items, sigma=40.0)
model = train(model, pairs, targets,
              n_epochs=3000, learning_rate=0.05, report_every=100)
```

Here, too, the inputs are distances, and the outputs are embeddings of the instances. These can again be displayed in a scatter plot (Figure 32). The resulting embedding is not substantially different from the embedding produced by MDS. This is interesting, since the two methods have completely different objective functions. The differences between MDS and t-SNE really become apparent with larger and high-dimensional data. There, the local neighborhood is often more informative than the global geometry, so t-SNE better highlights groups of similar instances, whereas MDS continues to follow primarily the global distances. An example of such a very different embedding is shown in Figure 31, where with MDS it is difficult to distinguish groups that are clearly visible in the t-SNE embedding.

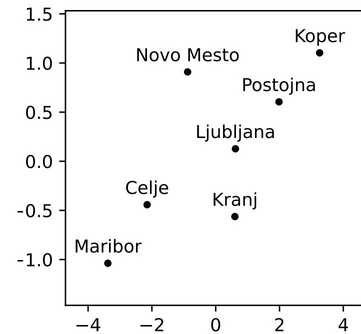

Figure 32: Two-dimensional embedding with t-SNE.

Force-Directed Embeddings

Let us examine a third approach. We consider it because its starting point differs substantially from the previous two and because it is very well established in the field of network visualization. The approach assumes that instances are points between which forces act. Pairs of instances for which distances are known, Here we introduce another important distinction compared to the previous examples: so far, we have considered data where distances were given for all pairs of instances, which, however, are not necessary for force-directed methods. It is often sufficient to know distances or connections only for selected pairs. are connected by "springs" that attempt to maintain a prescribed length, while at the same time a repulsive force acts between all pairs to prevent the points from becoming too densely packed. The positions of the points in the plane are then obtained by finding a force equilibrium, which we determine by minimizing an appropriate energy function. Although such an approach was developed in the field of network visualization, where the goal is to arrange graph nodes clearly, it can be used equally successfully for data embeddings.

As before, let $y_i \in \mathbb{R}^2$ again denote the position of instance i in the plane. For pairs of instances (i, j) for which the distance d_{ij} is known, we introduce a "spring" energy

$$E_{\text{spring}} = \frac{1}{|E|} \sum_{(i,j)\in E} \left(\|y_i - y_j\| - d_{ij}\right)^2,$$

where E denotes the set of pairs with known distances. This term forces connected pairs to preserve the prescribed distances in the

embedding.

At the same time, a repulsive force acts between all pairs of instances, which we describe with the energy

$$E_{\text{repulsion}} = \frac{1}{\binom{n}{2}} \sum_{i<j} \frac{1}{\|y_i - y_j\|},$$

We write the total energy as the sum of both terms

$$E = E_{\text{spring}} + C\, E_{\text{repulsion}},$$

where the constant C determines the relative influence of the repulsive term. The solution consists of such positions of the points y_i that minimize this total energy.

At first glance, the above objective function appears most similar to the one used by MDS. However, the similarity is only superficial and holds only when distances are given for all pairs. In MDS, a single term treats all pairs of instances equally and attempts to globally preserve all distances. In force-directed embeddings, however, the objective function splits into multiple parts: springs act only between selected pairs (e.g., connected in a network) and maintain the local structure, while repulsion acts between all pairs and prevents points from clustering too densely. In this way, we obtain a balance between the local preservation of distances and the global separation of points, which leads to substantially different embeddings than in MDS.

This time, the implementation code is somewhat longer, due to the two terms in the objective function and the necessary normalizations:

```
class ForceDirectedLayout:
    def __init__(self, items, rep_strength=5000.0):
        random.seed(100)
        self.items = sorted(items)
        self.rep_strength = rep_strength

        self.pos = {
            i: [Value(random.uniform(-1, 1)) for _ in range(2)]
            for i in self.items
        }

    def parameters(self):
        return [p for pos in self.pos.values() for p in pos]

    def d(self, a, b):
        dx = self.pos[a][0] - self.pos[b][0]
        dy = self.pos[a][1] - self.pos[b][1]
        return (dx ** 2 + dy ** 2 + Value(1e-6)) ** 0.5
```

```
def loss(self, edges, target_lengths):
    # springs between connected pairs
    E = edges
    L = target_lengths
    spring = Value(0.0)
    nE = len(E)

    for (a, b), t in zip(E, L):
        dist = self.d(a, b)
        spring += (dist - Value(t)) ** 2

    spring = spring / Value(nE)

    # repulsion between all pairs
    rep = Value(0.0)
    N = len(self.items)
    cnt = 0

    for i in range(N):
        for j in range(i + 1, N):
            a = self.items[i]
            b = self.items[j]
            dist = self.d(a, b)
            rep +=  dist ** (-1)
            cnt += 1

    rep = rep * self.rep_strength / cnt

    return spring + rep
```

In exactly the same way as for MDS and t-SNE, we also obtain force-directed embeddings. Here, too, convergence is fast, and the resulting visualization is surprisingly similar to those obtained with the previous two methods:

```
model = ForceDirectedLayout(items, rep_strength=5000.0)
model = train(model, pairs, targets, n_epochs=3000,
    learning_rate=0.05, report_every=100)
```

If we additionally wish to prevent the entire configuration from “drifting” away from the origin, we add a weak centering term

Have we already mentioned such a centering term in previous chapters? For example in linear regression? Regularization!

$$E_{\text{center}} = \frac{1}{n} \sum_{i=1}^{n} \|y_i\|^2,$$

We now write the total energy as the sum of all three terms

$$E = E_{\text{spring}} + C\, E_{\text{repulsion}} + \lambda\, E_{\text{center}},$$

where by setting the value of the constant λ we determine the influence of the centering term. In the code, we change only the final part

of the `loss` method

```
        # centering loss
        cent = Value(0.0)
        for i in self.items:
            x, y = self.pos[i]
            cent += x * x + y * y

        cent = self.cent_strength * cent / N

        return spring + rep + cent
```

and add the setting `self.cent_strength` to the class initialization. In our implementation, we set its value to 0.001, and for the problem defined in this way it did not have a particularly large effect.

Connection Between t-SNE and Force-Directed Embeddings

We presented the t-SNE technique as an approach that minimizes the Kullback-Leibler divergence between the similarity distributions in the original space and the embedding space:

$$\mathcal{L}(Y) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}},$$

where

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|y_k - y_l\|^2)^{-1}}.$$

To understand the optimization dynamics, we compute the gradient of the objective function with respect to the positions of the points y_i. After differentiation (here we present only the result), we obtain

$$\frac{\partial \mathcal{L}}{\partial y_i} = 4 \sum_{j \neq i} (p_{ij} - q_{ij}) \frac{(y_i - y_j)}{1 + \|y_i - y_j\|^2}.$$

This expression can be interpreted as a sum of forces acting on the point y_i. The gradient can be decomposed into two terms:

$$\frac{\partial \mathcal{L}}{\partial y_i} = 4 \sum_{j \neq i} p_{ij} \frac{(y_i - y_j)}{1 + \|y_i - y_j\|^2} - 4 \sum_{j \neq i} q_{ij} \frac{(y_i - y_j)}{1 + \|y_i - y_j\|^2}.$$

The first term can be interpreted as representing attractive forces between points i and j, whose strength is proportional to the similarity p_{ij} in the original space. These forces are strongest for pairs that are close in the original space and pull the corresponding points together in the embedding. The second term represents repulsive forces acting between all pairs of points and preventing them from clustering

too densely. Their strength arises from the distribution q_{ij}, which depends on the current arrangement of points in the embedding.

The gradient of t-SNE can therefore be interpreted as a balance between attractive and repulsive forces. In this sense, t-SNE is related to force-directed embeddings, except that the interactions are determined indirectly through a probabilistic model. The difference is that in force-directed approaches these interactions are specified directly through spring and repulsive terms, whereas in t-SNE they arise from the probabilistic model. The attractive forces in t-SNE explicitly emphasize local neighbors through the weights p_{ij}, while the form of the repulsive term with denominator $1 + \|y_i - y_j\|^2$ ensures a long-range repulsion and thereby prevents the clustering of points.

The t-SNE approach can therefore be understood as a probabilistically grounded force-directed embedding, where the balance between attractive and repulsive forces determines the final arrangement of points in space.

On the Implementations

The implementations in this chapter are intentionally simplified. Their purpose is not numerical efficiency, but rather a clear illustration of the connection between the objective function, the gradient, and the final embedding. We therefore did not provide the most numerically efficient solutions. This is especially true for the t-SNE method, where we used a considerably simplified version: the width of the Gaussian distribution was determined by a single parameter σ, identical for all points, whereas in practice σ_i is adjusted separately for each instance so that all instances in the original space have an "equal" number of neighbors. In addition, modern implementations include numerous improvements, such as smarter initialization and approximate force computations (e.g., the Barnes–Hut approximation, where distant points are replaced by a single average point), which substantially accelerate execution on larger datasets. Another useful trick is early exaggeration, where in the initial iterations we artificially increase the values of p_{ij}, thereby strengthening the attractive forces between nearby points. In this way, groups first become clearly formed and separated, and only afterward are their mutual relationships more precisely adjusted.

The parameter that determines the number of neighbors in t-SNE is called *perplexity*.

The same applies to multidimensional scaling. Although the objective function can be minimized using gradient descent, in practice the SMACOF algorithm (*Scaling by MAjorizing a COmplicated Function*) is used most frequently. This approach is based on the idea of majorization: instead of directly optimizing a difficult objective function, in each step we construct a simpler upper approximation that can be minimized in closed form. This yields an iterative procedure that monotonically decreases the value of the objective function and is generally more stable and faster than simple gradient descent.

Perhaps the closest to practical use, therefore, is the force-directed

embedding approach, where even basic models with springs and repulsion often produce useful results. Yet even there, additional improvements are encountered in practice, such as weighting of connections, adjusting the strength of forces over time (so-called "cooling"), and faster approximate computations of repulsive interactions. For practical use, the key point is that the different approaches arise from different objectives: MDS attempts to preserve distances, t-SNE preserves local neighborhoods, while force-directed methods seek a balance between attractive and repulsive interactions.