Data Maps

The title of this chapter is somewhat unusual—or rather, nonstandard. It should more properly be titled "dimensionality reduction" or "projections and vector embeddings." But we'll stick with "maps." The aim of this chapter is to review several techniques that can represent data—i.e., instances described in attribute space or instances for which distances can be computed—as points in a scatterplot. That is, in two or, if truly necessary, three dimensions. The conditions that such a visualization must meet will depend on the selected technique. In one case, for example, we might want the points—that is, the instances—to be as spread out as possible in the visualization. In another, we may require that the distances between instances reflect as faithfully as possible their distances in the original space. In a third, we might care only about preserving neighborhood relationships.

This chapter on data maps deliberately follows the chapter on clustering. These two techniques are often used together. Most clustering methods are not designed for visualizing data. Exceptions include hierarchical clustering and the related nearest-neighbor merging approach. But all other techniques simply find clusters and do not know how to visualize either the process or the results. With data maps, we aim to visually represent relationships between data—that is, to find neighbors or show pairwise distances in 2D or 3D. We can then overlay, for example using colors, the clustering results. If the results are consistent—i.e., instances from the same cluster also appear grouped in the data map—that confirms that the discovered clusters are real and can be meaningfully presented via visualization.

Data maps also serve the purpose of explanation. Perhaps we shouldn't even say "also" here, since explanation is their primary purpose—we create data maps precisely so that we can visually represent the data, reflect on them, and identify interesting regions of the map that can be interpreted.

If the original data are given in attribute form, then mapping them means reducing the dimensionality from m dimensions—that is, from feature space—to two or three dimensions. In practice, as we write this on paper, we are concerned with two dimensions, as three dimensions cannot be represented. The same applies to computer screens. While 3D visualizations are possible on computers, they require interactive tools or specialized 3D display equipment. Therefore, in this chapter, we will focus on two-dimensional maps, although all techniques we present can also be used for 3D representations.

One final note before the descriptions: with the exception of PCA, we do not explain how the solutions are actually computed. For each method, we describe the objective function, but since most of these (again, PCA is the exception) lack an analytical solution, we simply note that a numerical one exists. This solution is based on gradient descent, an approach we will explore further in upcoming chapters. In general, the techniques described in this and the next chapter are based on defining an objective function, which we then pass—along with the data—to a numerical optimization process that is essentially the same for all techniques.

Principal Component Analysis

The goal of Principal Component Analysis (PCA) is to find the most informative two-dimensional representation of the data. We want to identify directions in feature space along which the data are most spread out—in other words, where they have the greatest variance. This reduces the dimensionality of the data from m (the number of features) to 2 (or 3), while preserving as much information as possible.

Mathematically, PCA is a projection of the data from a higher-dimensional space into a lower-dimensional space. Instead of retaining all original features, we find new axes (components) that are linear combinations of the original features and project the data onto these new axes. These axes are chosen to maximize the variance in the data.

Mathematical Derivation

1. Centering the Data

Let X be a data matrix of dimensions $n \times m$, where n is the number of samples and m the number of features. We assume the data are centered, meaning that for each feature:

$$\frac{1}{n}\sum_{i=1}^{n} X_{i,j} = 0, \quad \text{for each } j = 1, 2, \ldots, m$$

If the data are not centered, we center them in advance.

2. Finding the First Component $\mathbf{u}_1$

We seek a vector $\mathbf{u}_1$ (of size $m \times 1$) that defines the first principal component—a direction in feature space such that the variance of the data projected onto it is maximized. The projection

of the data matrix X onto $\mathbf{u}_1$ is:

$$z = X\mathbf{u}_1$$

The variance of the projection z is:

$$\text{Var}(z) = \frac{1}{n}\|X\mathbf{u}_1\|^2 = \frac{1}{n}\mathbf{u}_1^T X^T X \mathbf{u}_1$$

To maximize this variance, our objective function is:

$$\max_{\mathbf{u}_1} \mathbf{u}_1^T S \mathbf{u}_1$$

where $S = \frac{1}{n}X^T X$ is the **covariance matrix** of the data.

To prevent an unbounded solution, we constrain the length of $\mathbf{u}_1$ to 1:

$$\mathbf{u}_1^T \mathbf{u}_1 = 1$$

3. Optimization and Solution

We solve this optimization problem using Lagrange multipliers. The Lagrangian function is:

$$L(\mathbf{u}_1, \lambda) = \mathbf{u}_1^T S \mathbf{u}_1 - \lambda(\mathbf{u}_1^T \mathbf{u}_1 - 1)$$

We find the stationary points:

$$\frac{\partial L}{\partial \mathbf{u}_1} = 2S\mathbf{u}_1 - 2\lambda\mathbf{u}_1 = 0$$

$$S\mathbf{u}_1 = \lambda\mathbf{u}_1$$

This is the eigenvalue equation for the covariance matrix. Thus, $\mathbf{u}_1$ is an eigenvector of S, and the corresponding eigenvalue λ satisfies:

$$S\mathbf{u}_1 = \lambda_1\mathbf{u}_1$$

4. Solution

The variance of the projection is:

$$\text{Var}(z) = \frac{1}{n}\|X\mathbf{u}_1\|^2 = \mathbf{u}_1^T S \mathbf{u}_1$$

Using the eigenvalue equation:

$$S\mathbf{u}_1 = \lambda_1 \mathbf{u}_1$$

we get:

$$\mathbf{u}_1^T S \mathbf{u}_1 = \mathbf{u}_1^T \lambda_1 \mathbf{u}_1 = \lambda_1 (\mathbf{u}_1^T \mathbf{u}_1) = \lambda_1$$

since we assumed $\mathbf{u}_1^T \mathbf{u}_1 = 1$. Therefore:

$$\mathrm{Var}(z) = \lambda_1$$

This means that the eigenvalue λ_1 is equal to the variance of the data in the direction of the first principal component $\mathbf{u}_1$.

To maximize the projection variance, we select the eigenvector corresponding to the largest eigenvalue λ_1. The variance in the direction $\mathbf{u}_1$ equals this eigenvalue:

$$\mathrm{Var}(z) = \lambda_1$$

Scree Plot

PCA results are often visualized using a scree plot, which shows explained variances. The x -axis represents the principal components (1st, 2nd, 3rd, …), and the y-axis shows the proportion of variance each component explains. For each component k, the explained variance is computed as the ratio between its eigenvalue λ_k and the sum of all eigenvalues:

$$\text{Explained Variance}_k = \frac{\lambda_k}{\sum_{j=1}^{m} \lambda_j}$$

The scree plot provides a visual tool for deciding how many components to retain. Typically, we look for the “elbow” in the plot—the point where additional components contribute very little extra explained variance.

Interpretation

The principal axes determined by PCA are represented as linear combinations of the original features, meaning each component is a weighted sum of the original attributes. If $\mathbf{u}_1$ is the first principal component, then it can be written as:

$$\mathbf{u}_1 = (w_1, w_2, \ldots, w_m)^T$$

where w_j are the weights for each feature. The absolute value $|w_j|$ indicates how much each attribute contributes to the component—the larger the weight, the greater the influence of that feature.

When interpreting the weights, we must consider that the data have been normalized beforehand—that is, each feature has a similar scale (e.g., mean 0 and standard deviation 1). This is important because PCA otherwise favors features with greater numerical spread, regardless of their actual informational value. Normalization ensures that contributions to components reflect the structure of the data rather than differing measurement units or ranges.

Alternative Numerical Solution

In practice, when we want to project the data into two dimensions, we only need the first two principal components. There is no need to compute all m eigenvectors of the covariance matrix, which would be computationally expensive for large m. Instead, we can use faster procedures such as the power method and Gram-Schmidt orthogonalization.

The power method is a simple algorithm for finding the dominant eigenvector of a symmetric matrix S, such as our covariance matrix. It works by repeatedly multiplying an arbitrary initial vector by S and normalizing the result each time:

$$\mathbf{v}^{(k+1)} = \frac{S\mathbf{v}^{(k)}}{\|S\mathbf{v}^{(k)}\|}$$

This process is repeated until $\mathbf{v}^{(k)}$ stabilizes. The result is the first eigenvector $\mathbf{u}_1$, corresponding to the largest eigenvalue λ_1.

To find the second eigenvector $\mathbf{u}_2$, which must be orthogonal to $\mathbf{u}_1$, we use Gram-Schmidt orthogonalization to "clean" the next approximation:

1. Compute an approximation $\mathbf{v}$ for the next eigenvector (e.g., again using the power method).
2. Subtract the projection onto $\mathbf{u}_1$:

$$\mathbf{v}_\perp = \mathbf{v} - (\mathbf{v}^T\mathbf{u}_1)\mathbf{u}_1$$

3. Normalize $\mathbf{v}_\perp$ to get $\mathbf{u}_2$.

Computing all m eigenvectors and eigenvalues requires $O(m^3)$ time, as it involves solving the general eigenvalue problem. But if we only need the first two components, we can use the power method and Gram-Schmidt to obtain the desired results in $O(m^2)$ time, since we only multiply a matrix by a vector. This is significantly faster, especially for high-dimensional data where m is large.

This approach enables efficient computation of the two principal axes needed to prepare a two-dimensional projection of the data for visualization.

Multidimensional Scaling

Multidimensional Scaling (MDS) is a method that finds a low-dimensional representation of data (typically in two or three dimensions) such that the distances between points are as close as possible to the original distances between instances in the high-dimensional space. The purpose of MDS is thus to visualize data by preserving the distances between instances. The input to MDS is the matrix of pairwise distances between data points; the method does not require an attribute-based representation. It operates directly on distances, not features.

The objective function that MDS minimizes is the sum of squared differences between the original and the newly computed distances:

$$\min_{\mathbf{z}_1,\ldots,\mathbf{z}_n} \sum_{i<j} \left(d_{ij} - \|\mathbf{z}_i - \mathbf{z}_j\|\right)^2$$

where d_{ij} is the given distance between instances i and j, and $\|\mathbf{z}_i - \mathbf{z}_j\|$ is the distance between them in the new low-dimensional representation.

Solving this problem—embedding the points in a new low-dimensional space—is done via numerical optimization, as the problem is nonlinear and lacks an analytical solution. This is unlike PCA, which does admit an analytical solution. The process of finding the MDS solution typically follows these steps:

1. **Initialize** the points $\mathbf{z}_1, \ldots, \mathbf{z}_n$, for example randomly in a plane. In practice, initialization is often based on the projection obtained from PCA.
2. **Iteratively improve** the configuration by moving the points $\mathbf{z}_i$ in directions that decrease the objective function. This is done using **gradient descent** or other optimization

algorithms such as iterative majorization (e.g., the SMACOF algorithm).

3. For gradient descent, compute the **gradient of the objective function** with respect to each coordinate $\mathbf{z}_i$. This means finding the direction to move each point to reduce the error.
4. **Repeat** until the objective function stabilizes—that is, until **convergence** is reached.

MDS can be computationally expensive because the number of point pairs grows quadratically with n—there are $n(n-1)/2$ pairs to consider, which becomes very large for big datasets. Because of this, faster or approximate algorithms have been developed that consider only a subset of distances (e.g., distances to nearest neighbors) or use optimization shortcuts such as stochastic gradient descent, which updates only a random subset of points in each iteration. One of the well-known accelerated variants is the SMACOF method (*Scaling by Majorizing a Complicated Function*), which solves an approximation of the original problem in each step and converges faster.

It is important to note that MDS is not a projection, because the new coordinates are not linear combinations of the original features; rather, it is an embedding into a new vector space. The new dimensions have no intrinsic meaning—they exist only to help preserve distances. Furthermore, the solution is invariant to rotation, reflection, and translation: if all points are rotated or shifted, the result is still valid, since the distances remain the same.

The key difference between embedding and projection is that in projection (like PCA), the new axes are combinations of existing features and sometimes interpretable, whereas in embedding (like MDS), the axes are defined solely for the purpose of preserving pairwise distances and have no inherent meaning.

One drawback of MDS is that it attempts to preserve *all* distances between point pairs, which can lead to overemphasizing large distances at the expense of preserving local structure. This often makes MDS less effective at revealing local clusters and neighborhoods.

Example: Suppose we have two points that are very close together in the original space, with a distance of 1, and two other points that are very far apart, with a distance of 100. If the embedding distorts all distances by 10%, the close distance becomes 1.1, while the large distance becomes 110. Although the relative error is the same, the absolute error is much larger for the distant pair (10 units vs. 0.1). Consequently, MDS will prioritize correcting larger absolute errors at the cost of precision in small distances—even though the small distances are often more important for understanding local structure.

Neighborhood Preservation (t-SNE)

The t-SNE method (*t-distributed Stochastic Neighbor Embedding*) is designed to find a good data representation in which similar data points are located close together, and dissimilar ones are placed far apart. The key idea behind t-SNE is to focus on preserving the *local structure* of the data—ensuring that neighboring points in high-dimensional space remain neighbors in the two-dimensional visualization.

t-SNE first computes how likely each pair of data points is to be neighbors in the original space. Then, it arranges the points in a low-dimensional space so that these neighbor probabilities are as similar as possible. Unlike methods that preserve distances (such as MDS), t-SNE preserves neighborhoods, making it particularly useful for identifying clusters and local patterns in the data.

Mathematically, t-SNE defines for each pair of points i and j a probability p_{ij} that reflects how close the points are in the original space. This probability is computed using a Gaussian distribution:

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|\mathbf{x}_i - \mathbf{x}_k\|^2 / 2\sigma_i^2)}$$

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

The parameter σ_i controls how far the neighborhood extends around point $\mathbf{x}_i$. A larger σ_i means more distant points are treated as neighbors; a smaller σ_i limits the neighborhood to closer points. To adjust σ_i for each point appropriately, a global parameter called *perplexity* is used, which specifies the expected number of neighbors. The algorithm then tunes each σ_i so that the number of effective neighbors matches the chosen perplexity. Perplexity thus balances local and global structure: smaller values highlight fine-grained clusters, while larger ones capture broader structure.

In the low-dimensional space, where we have points $\mathbf{z}_1, \ldots, \mathbf{z}_n$, a similar probability q_{ij} is defined, reflecting how close the points are in 2D. Instead of a Gaussian, t-SNE uses a Student t-distribution with one degree of freedom (i.e., the Cauchy distribution), which allows for larger distances and thus more natural spacing of distant points:

$$q_{ij} = \frac{(1 + \|\mathbf{z}_i - \mathbf{z}_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|\mathbf{z}_k - \mathbf{z}_l\|^2)^{-1}}$$

The main optimization goal is to minimize the difference between the probabilities p_{ij} and q_{ij}. This difference is measured using the Kullback-Leibler (KL) divergence:

$$\min_{\mathbf{z}_1, \ldots, \mathbf{z}_n} KL(P \| Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

The objective function measures how far the neighborhood structure in 2D deviates from that in the original space. The smaller this difference, the better the match between the 2D and high-dimensional neighborhood structures.

The solution is found via iterative numerical optimization, typically using gradient descent. In each iteration, the gradient of the objective function with respect to the coordinates $\mathbf{z}_i$ is computed, and the coordinates are adjusted to reduce the objective. Due to the nature of the objective and the large number of point pairs, t-SNE is computationally demanding and relatively slow, especially on large datasets. To address this, faster variants such as Barnes-Hut t-SNE and stochastic optimization methods have been developed.

It is important to understand that t-SNE is not a projection but an embedding into a new vector space. The new dimensions do not represent any linear combinations of original features and have no intrinsic meaning. t-SNE finds a placement of points that best preserves the neighborhood structure of the original data. Like with MDS, the result is invariant under rotation, reflection, and translation.

When using t-SNE, it is essential to interpret the layout in terms of *neighborhoods*. That is, we should focus on which points are close to one another—because t-SNE primarily preserves local relationships. The distances between more distant clusters should not be interpreted literally: two clusters that appear far apart in the embedding may actually be close (or even further apart) in the original space. Therefore, the output of t-SNE should be interpreted by examining the internal structure of clusters and their local neighborhoods, not by comparing the distances between clusters.

Data Embedding Based on Manifolds and Graphs (UMAP)

The UMAP method (*Uniform Manifold Approximation and Projection*) is based on manifold theory and graph theory and aims to find a data representation that preserves both local characteristics (neighborhoods) and aspects of global structure, such as inter-cluster distances. Compared to t-SNE, UMAP is often less sensitive to parameter settings and better at maintaining the overall structure of the data.

The core idea of the method is to first build a neighborhood graph in high-dimensional space that captures local relationships between data points, and then find a low-dimensional layout that preserves those neighborhood connections.

Mathematically, UMAP first defines probabilities p_{ij} that quantify how strongly point j is connected to point i in the high-dimensional space. These probabilities are computed using distances and the following function:

$$p_{ij} = \exp\left(-\frac{d(\mathbf{x}_i, \mathbf{x}_j) - \rho_i}{\sigma_i}\right)$$

where $d(\mathbf{x}_i, \mathbf{x}_j)$ is the distance between points i and j, ρ_i is the distance to the nearest neighbor (to ensure local adaptation), and σ_i is the neighborhood radius, which is chosen so that the average number of neighbors matches a target perplexity. In the low-dimensional space (e.g., 2D), UMAP defines similar probabilities q_{ij}, describing relationships between points in the new layout:

$$q_{ij} = \frac{1}{1 + a\|\mathbf{z}_i - \mathbf{z}_j\|^{2b}}$$

where a and b are parameters that define the shape of the curve and are selected to best model the distribution of distances.

The objective function UMAP minimizes is similar to Kullback-Leibler divergence but is based on binary cross-entropy between the probabilities p_{ij} and q_{ij}:

$$C = \sum_{i \neq j} \left[p_{ij} \log q_{ij} + (1 - p_{ij}) \log(1 - q_{ij})\right]$$

This function guides UMAP to find a configuration in which points connected in the high-dimensional graph are placed close together in the low-dimensional space, while disconnected pairs are placed far apart. As in t-SNE, the optimization is performed using gradient descent (described in more detail in the following chapters), where coordinates in the low-dimensional space are iteratively adjusted to minimize the cost.

UMAP is therefore a fast and effective method for embedding data into a low-dimensional space while preserving both local and, to some extent, global structure. This makes it especially suitable for visualizing large and complex datasets.

Autoencoders

Autoencoders are neural network models designed for unsupervised dimensionality reduction. Unlike methods such as PCA that perform linear projection, autoencoders can learn *nonlinear* mappings, enabling them to capture complex structures in data. An autoencoder consists of two parts: an **encoder**, which compresses the input data into a lower-dimensional representation (the "bottleneck"), and a **decoder**, which reconstructs the original data from this compressed form.

The key idea is that, by training the network to minimize the difference between the original input and its reconstruction, the encoder must learn a compact internal representation that preserves essential information. This internal representation can then be used as a low-dimensional embedding for visualization, clustering, or as input to other models.

Formally, given input data $\mathbf{x} \in \mathbb{R}^m$, the encoder maps $\mathbf{x}$ to a latent representation $\mathbf{z} \in \mathbb{R}^d$ where $d < m$, via a function f_θ, typically a neural network. The decoder, a function g_ϕ, reconstructs the input as $\hat{\mathbf{x}} = g_\phi(f_\theta(\mathbf{x}))$. The model is trained to minimize the reconstruction loss:

$$\min_{\theta,\phi} \sum_{i=1}^{n} ||\mathbf{x}_i - \hat{\mathbf{x}}_i||^2$$

Because the encoder-decoder pair is flexible and can model nonlinear transformations, autoencoders are particularly useful when linear techniques such as PCA fail to preserve important relationships in the data. Once trained, the encoder alone can be used to produce a

2D or 3D embedding suitable for visualization—just like other methods discussed in this chapter.

Variants such as **variational autoencoders (VAEs)** impose additional structure on the latent space, often with the goal of making the space more continuous or interpretable. However, for the purposes of data maps and visual embeddings, even standard autoencoders often perform well, particularly when combined with nonlinear activation functions and regularization techniques to prevent overfitting.

Interpreting Data Maps

Aside from a brief mention in the section on principal component analysis, we haven't yet addressed the topic of interpreting data maps—despite the fact that interpretation is the primary reason for constructing them in the first place. While we wait for the authors of these notes to enrich this chapter with examples (including those discussed in lectures), we can note here that interpreting data maps is similar to interpreting clusters, as described in the previous chapter: we select a group of points and identify which features and feature values are characteristic of the group and distinguish it from the rest of the data.

Ideally, such groups and their interpretations would be automatically identified by algorithms that summarize their findings—perhaps with the help of large language models. However, software that performs this kind of automatic interpretation reliably is still (mostly) nonexistent, leaving ample opportunity for future development in this area.