Chapter 9

Unsupervised Learning

"Education is not the filling of a pot but the lighting of a fire."– W.B. Yeats

9.1 Introduction

The supervised learning methods discussed in the previous chapters try to learn how the features in the data relate to specific target variables. Unsupervised learning methods try to learn how the features are related to one another. In other words, unsupervised learning methods do not have a specific goal in mind in order to supervise the learning process. Rather, unsupervised methods learn the key patterns in the underlying data that relate all data points and attributes to one another without a specific focus on any particular data items. In supervised learning, specific attributes (e.g., regressors or class labels) are more important, and therefore play the role of teachers (i.e., *supervisors*) to the learning process.

Unsupervised methods generally learn aggregate trends in the data matrix, and in many cases, they create a compressed model of key data characteristics. This compressed model can even be used in order to recreate examples of typical data points. From a human intelligence point of view, we experience a lot of sensory inputs on a day-to-day basis, and often store away the key aspects of these experiences. These experiences often turn out to be useful for more specific tasks. A similar analogy holds true in machine learning, where unsupervised learning is often leveraged in supervised learning tasks. The common types of relationships captured by unsupervised learning are as follows:

1. **Row-wise relationships:** In this case, we attempt to learn which rows of the data set are closely related to one another. Therefore, the problem reduces to one of finding important clusters in the data.

2. **Column-wise relationships:** In this case, the goal is to create a data set represented by a smaller set of columns, by using the interrelationships and correlations among the columns of the original data set. This problem is referred to as *dimensionality reduction*.

C. C. Aggarwal, *Artificial Intelligence*, https://doi.org/10.1007/978-3-030-72357-6_9

3. **Combining row-wise and column-wise relationships:** Several forms of dimensionality reduction are also well suited to clustering, because they capture the relationships among the rows as well. This is because they capture both row-wise and column-wise relationships.

In this chapter, we will discuss different types of unsupervised models like clustering and dimensionality reduction. Clustering and dimensionality reduction are both forms of data compression, and can be used to represent the data approximately in a smaller amount of space. The ability to represent the data approximately in small amount of space is a natural characteristic of unsupervised learning methods, which try to learn the broader patterns in the data. Unsupervised models build compressed models of the data, so that one can express each point $\overline{X}_i$ approximately as a function of itself:

$$\overline{h}_i = F_{compress}(\overline{X}_i), \quad \overline{X}_i \approx F_{decompress}(\overline{h}_i)$$
$$\overline{X}_i \approx G(\overline{X}_i) = F_{decompress}(F_{compress}(\overline{X}_i))$$

The compressed representation $\overline{h}_i$ can be viewed as a concise description containing the most important characteristics of $\overline{X}_i$. The "hidden" representation $\overline{h}_i$ is typically not visible to the end user. The reconstruction of a data point from this hidden representation is approximate, and might sometimes drop unusual artifacts from the point. The hidden representation can be used for various applications like generative modeling and outlier detection.

This chapter is organized as follows. The next section introduces the problem of dimensionality reduction and matrix factorization. Methods are discussed for singular value decomposition and nonnegative matrix factorization. In addition, neural networks for nonlinear dimensionality reduction are discussed. Clustering methods are discussed in Section 9.3. A discussion of various applications of unsupervised learning is provided in Section 9.4. A summary is given in Section 9.5.

9.2 Dimensionality Reduction and Matrix Factorization

Consider an $n \times d$ data matrix D in which significant correlations exist among the rows (or columns). The presence of relationships among the rows (or columns) implies that there is an inherent redundancy in the data representation. One way of capturing this data redundancy is through the use of matrix factorization. In matrix factorization, an $n \times d$ data matrix D is represented as the product of two matrices that are much smaller:

$$D \approx UV^T \tag{9.1}$$

The matrix U is of size $n \times k$, and the matrix V is of size $d \times k$, where k is much less than $\min\{n, d\}$. The value of k is referred to as the *rank* of the factorization. One can always reconstruct the matrix exactly using a factorization of rank $k = \min\{n, d\}$, and this rank is referred to as *full-rank* factorization. It is more common to use small values of k, which is referred to as a *low-rank* factorization, but exact reconstruction is not possible in such cases. Furthermore, in such cases, the total number of entries in U and V is much less than the number of entries in D:

$$(n + d)k \ll nd$$

Therefore, the matrices U and V provide a compressed representation of the data, and the data matrix D can be approximately reconstructed as UV^T. Furthermore, the matrix

UV^T will often differ significantly in those entries of D that do not naturally conform to the aggregate trends in the data. In other words, such entries are *outliers*. In fact, outlier detection is a complementary form of unsupervised learning to clustering and dimensionality reduction.

This factorization is referred to as *low-rank* because the ranks of each of U, V, and UV^T are at most $k \ll d$, whereas the rank of D might be $\min\{n, d\}$. Note that there will always be some *residual error* $(D - UV^T)$ from the factorization. In fact, the entries in U and V are often discovered by solving an optimization problem in which the sum of squares (or other aggregate function) of the residual errors in $(D - UV^T)$ are minimized. Almost all forms of dimensionality reduction and matrix factorization are special cases of the following optimization model over matrices U and V:

Maximize similarity between entries of D and UV^T
subject to:
Constraints on U and V

By varying the objective function and constraints, dimensionality reductions with different properties are obtained. The most commonly used objective function is the sum of the squares of the entries in $(D - UV^T)$, which is also defined as the (squared) *Frobenius norm* of the matrix $(D - UV^T)$. The (squared) Frobenius norm of a matrix is also referred to as its *energy*, because it is the sum of the second moments of all data points about the origin. However, some forms of factorizations with probabilistic interpretations use a *maximum-likelihood* objective function. Similarly, the constraints imposed on U and V enable different properties of the factorization. For example, if we impose orthogonality constraints on the columns of U and V, this leads to a model known as *singular value decomposition (SVD)* or *latent semantic analysis (LSA)*. The latter terminology is used in the context of document data. The orthogonality of the basis vectors is particularly helpful in mapping new data points (i.e., data points not included in the original data set on which factorization is applied) to the transformed space in a simple way. On the other hand, better semantic interpretability can be obtained by imposing nonnegativity constraints on U and V. This chapter will discuss various types of reductions and their relative advantages.

9.2.1 Symmetric Matrix Factorization

Singular value decomposition can be viewed as the generalization of a type of factorization that is performed on positive semi-definite matrices. A square and symmetric $n \times n$ matrix A is positive semi-definite, if and only if for any n-dimensional column vector $\overline{x}$, we have $\overline{x}^T A \overline{x} \geq 0$.

Positive semi-definite matrices have the property that they can be *diagonalized* to the following form:

$$A = Q \Delta Q^T$$

Here Q is a $n \times n$ matrix with orthonormal columns, and Δ is an $n \times n$ diagonal matrix with nonnegative entries. Since Δ has nonnegative entries, it is common to express Δ as the square of another $n \times n$ diagonal matrix Σ, and define the diagonalization in the following way:

$$A = Q \Sigma^2 Q^T$$

The columns of Q are referred to as *eigenvectors*, which represent an orthonormal set of n vectors. Therefore, we have $Q^T Q = I$. The diagonal entries of Δ are referred to as

eigenvalues. What does an eigenvector and eigenvalue mean? Note that the relationship above can be expressed as follows (by postmultiplying with P and setting $Q^TQ = I$):

$$AQ = Q\Delta$$

If the ith column of Q is $\overline{q}_i$ and the ith diagonal entry of Δ is $\delta_i \geq 0$, we can express the above relationship as follows:

$$A\overline{q}_i = \delta_i \overline{q}_i$$

Note that multiplying an eigenvector with a matrix A simply scales the magnitude of the eigenvector, without changing its direction. Positive semi-definite matrix factorization is fundamental to machine learning because it is used in all sorts of machine learning applications, such as kernel methods.

One can also express positive semi-definite matrix factorization as a form of *symmetric* matrix factorization:

$$A = Q\Sigma^2 Q^T = \underbrace{(Q\Sigma)}_{U}(Q\Sigma)^T = UU^T$$

Note that this is a factorization of *full rank* in which we are not losing any accuracy of representation of A via factorization.

How does one relate positive semi-definite matrix factorization to the generic optimization formulation of the previous section? It can be shown that *truncated forms* of symmetric factorization, in which low-rank factorization is used, reduce to the following matrix factorization:

$$\begin{aligned} &\text{Minimize}_U \, \|A - UU^T\|^2 \\ &\text{subject to:} \\ &\text{No constraints on } U \end{aligned}$$

Here U is an $n \times k$ matrix rather than an $n \times n$, where $k \ll n$. Therefore, this factorization is of low rank. It can be shown that the optimal solution to this problem yields $U = Q_k\Sigma_k$, where Q_k is an $n \times k$ matrix containing the top-k eigenvectors of A in its columns (i.e., eigenvectors with largest eigenvalues), and Σ_k is a diagonal matrix containing the square-root of the corresponding eigenvalues of A. Singular value decomposition is a natural generalization of this idea to asymmetric and rectangular matrices.

9.2.2 Singular Value Decomposition

Singular value decomposition (SVD) is the most common form of dimensionality reduction for multidimensional data. Consider the simplest possible factorization of the $n \times d$ matrix D into an $n \times k$ matrix $U = [u_{ij}]$ and the $d \times k$ matrix $V = [v_{ij}]$ as an *unconstrained matrix factorization problem*:

$$\begin{aligned} &\text{Minimize}_{U,V} \, \|D - UV^T\|_F^2 \\ &\text{subject to:} \\ &\text{No constraints on } U \text{ and } V \end{aligned}$$

Here $\|\cdot\|_F^2$ refers to the (squared) Frobenius norm of a matrix, which is the sum of squares of its entries. The matrix $(D - UV^T)$ is also referred to as the *residual matrix*, because its entries contain the residual errors obtained from a low-rank factorization of the original matrix D. This optimization problem is the most basic form of matrix factorization

with a popular objective function and no constraints. This formulation has infinitely many alternative optimal solutions. However, one[1] of them is such that the columns of V are orthonormal, which allows transformations of new data points (i.e., rows not included in D) with simple axis rotations (i.e., matrix multiplication). A remarkable property of the unconstrained optimization problem above is that imposing orthogonality constraints does not worsen the optimal solution. *The following constrained optimization problem shares at least one optimal solution as the unconstrained version* [50, 175]:

$$\begin{aligned} &\text{Minimize}_{U,V} \|D - UV^T\|_F^2 \\ &\text{subject to:} \\ &\text{Columns of } U \text{ are mutually orthogonal} \\ &\text{Columns of } V \text{ are mutually orthonormal} \end{aligned}$$

In other words, one of the alternative optima to the unconstrained problem also satisfies orthogonality constraints. It is noteworthy that *only* the solution satisfying the orthogonality constraint is considered SVD because of its interesting properties, even though other optima do exist.

Another remarkable property of the solution (satisfying orthogonality) is that it can be computed using *eigen-decomposition* of either of the positive semi-definite matrices $D^T D$ or DD^T. The following properties of the solution can be shown:

1. The columns of V are defined by the top-k unit eigenvectors of the $d \times d$ positive semi-definite and symmetric matrix $D^T D$. The diagonalization of a symmetric and positive semi-definite matrix results in orthonormal eigenvectors with non-negative eigenvalues. After V has been determined, we can also compute the reduced representation U as DV, which is simply an axis rotation operation on the rows in the original data matrix. This is caused by the orthogonality of the columns of V, which results in $DV \approx U(V^T V) = U$. One can also use this approach to compute the reduced representation $\overline{X}V$ of any row-vector $\overline{X}$ that was not included in D.

2. The columns of U are also defined by the top-k *scaled* eigenvectors of the $n \times n$ *dot-product matrix* DD^T in which the (i, j)th entry is the dot-product similarity between the ith and jth data points. The scaling factor is defined so that each eigenvector is multiplied with the square-root of its eigenvalue. In other words, the *scaled eigenvectors of the dot-product matrix can be used to directly generate the reduced representation.* This fact has some interesting consequences for the nonlinear dimensionality reduction methods, which replace the dot product matrix with another similarity matrix. This approach is also efficient for linear SVD when $n \ll d$, and therefore the $n \times n$ matrix DD^T is relatively small. In such cases, U is extracted first by eigen-decomposition of DD^T, and then V is extracted as $D^T U$.

3. Even though the n eigenvectors of DD^T and d eigenvectors of $D^T D$ are different, the top $\min\{n, d\}$ eigenvalues of DD^T and $D^T D$ are the same values. All other eigenvalues are zero.

4. The total squared error of the approximate matrix factorization of SVD is equal to the sum of the eigenvalues of $D^T D$ that are *not* included among the top-k eigenvectors. If we set the rank of the factorization k to $\min\{n, d\}$, we can obtain an *exact* factorization into orthogonal basis spaces with zero error.

[1] This solution can be unique up to multiplication of any column of U or V with -1 under some conditions on uniqueness of retained singular values.

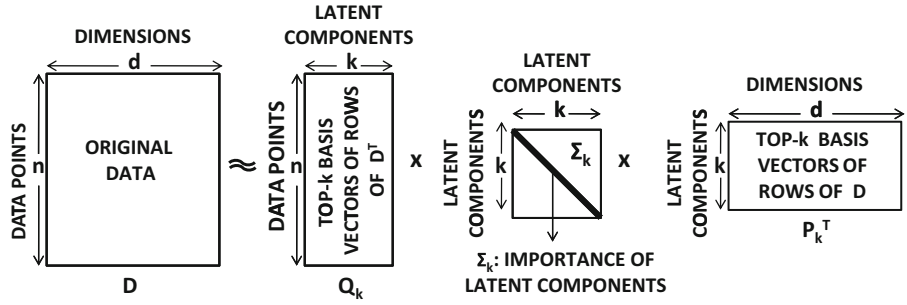


Figure 9.1: Dual interpretation of SVD in terms of the basis vectors of both D and D^T

This factorization of rank $k = \min\{n, d\}$ with zero error is of particular interest. We convert the two-way factorization (of zero error) into a three-way factorization, which results in a standard form of SVD:

$$D = Q\Sigma P^T = \underbrace{(Q\Sigma)}_{U} \underbrace{P^T}_{V^T} \tag{9.2}$$

Here, Q is an $n \times k$ matrix containing all the $k = \min\{n, d\}$ *non-zero* eigenvectors of DD^T, and P is a $d \times k$ matrix containing all the $k = \min\{n, d\}$ non-zero eigenvectors of $D^T D$. The columns of Q are referred to as the *left singular vectors*, whereas the columns of P are referred to as the *right singular vectors*. Furthermore, Σ is a (nonnegative) diagonal matrix in which the (r, r)th value is equal to the square-root of the rth largest eigenvalue of $D^T D$ (which is the same as the rth largest eigenvalue of DD^T). The diagonal entries of Σ are also referred to as *singular values*. Note that the singular values are always nonnegative by convention. The sets of columns of P and Q are each orthonormal because they are the unit eigenvectors of symmetric matrices. It is easy to verify (using Equation 9.2) that $D^T D = P\Sigma^2 P^T$ and that $DD^T = Q\Sigma^2 Q^T$, where Σ^2 is a diagonal matrix containing the top-k non-negative eigenvalues of $D^T D$ and DD^T (which are the same).

SVD is formally defined as the exact decomposition with zero error. What about the *approximate* variant of SVD, which is the primary goal of matrix factorization? In practice, one always uses values of $k \ll \min\{n, d\}$ to obtain *approximate* or *truncated* SVD:

$$D \approx Q\Sigma P^T \tag{9.3}$$

Using truncated SVD is the standard use-case in practical settings. Throughout this book, our use of the term "SVD" always refers to truncated SVD.

Just as the matrix P contains the d-dimensional basis vectors of D in its columns, the matrix Q contains the n-dimensional basis vectors of D^T in its columns. In other words, *SVD simultaneously finds approximate bases of both points and dimensions.* This ability of SVD to simultaneously find approximate bases for the row space and column space is shown in Figure 9.1. Furthermore, the diagonal entries of the matrix Σ provide a quantification of the relative dominance of the different semantic concepts.

One can express SVD as a weighted sum of rank-1 matrices. Let Q_i be the $n \times 1$ matrix corresponding to the ith column of Q and P_i be the $d \times 1$ matrix corresponding to the ith column of P. Then, the SVD product can be decomposed in *spectral form* using simple matrix-multiplication laws as follows:

$$Q\Sigma P^T = \sum_{i=1}^{k} \Sigma_{ii} Q_i P_i^T \tag{9.4}$$

Note that each $Q_i P_i$ is a rank-1 matrix of size $n \times d$ and a Frobenius norm of 1. Furthermore, it is possible to show that the Frobenius norm of $Q\Sigma P^T$ is given by $\sum_{i=1}^{k} \Sigma_{ii}^2$, which is the amount of *energy retained* in the representation. Maximizing the retained energy is the same as minimizing the loss defined by the sum of squares of the truncated singular values (which are small), because the sum of the two is always equal to $\|D\|_F^2$. The energy retained in the approximated matrix is the same as that in the transformed representation, because squared distances do not change with axis rotation. Therefore, the sum of the squares of the retained singular values provides the energy in the transformed representation DP. An important consequence of this observation is that the projection $D\overline{p}$ of D on any column $\overline{p}$ of P has an L_2-norm, which is equal to the corresponding singular value. In other words, SVD naturally selects the orthogonal directions along which the transformed data exhibits the largest scatter.

9.2.2.1 Example of SVD

An example of SVD helps in illustrating its inner workings. Consider a 6×6 matrix D defined over a document data set containing the following 6 words:

lion, tiger, cheetah, jaguar, porsche, ferrari

The data matrix D is illustrated below:

$$D = \begin{pmatrix} & \text{lion} & \text{tiger} & \text{cheetah} & \text{jaguar} & \text{porsche} & \text{ferrari} \\ \text{Document-1} & 2 & 2 & 1 & 2 & 0 & 0 \\ \text{Document-2} & 2 & 3 & 3 & 3 & 0 & 0 \\ \text{Document-3} & 1 & 1 & 1 & 1 & 0 & 0 \\ \text{Document-4} & 2 & 2 & 2 & 3 & 1 & 1 \\ \text{Document-5} & 0 & 0 & 0 & 1 & 1 & 1 \\ \text{Document-6} & 0 & 0 & 0 & 2 & 1 & 2 \end{pmatrix}$$

Note that this matrix represents topics related to both cars and cats. The first three documents are primarily related to cats, the fourth is related to both, and the last two are primarily related to cars. The word "jaguar" is *polysemous* because it could correspond to either a car or a cat. Therefore, it is often present in documents of both categories and presents itself as a confounding word. We would like to perform an SVD of rank-2 to capture the two dominant concepts corresponding to cats and cars, respectively. Then, on performing the SVD of this matrix, we obtain the following decomposition:

$$D \approx Q\Sigma P^T$$

$$\approx \begin{pmatrix} -0.41 & 0.17 \\ -0.65 & 0.31 \\ -0.23 & 0.13 \\ -0.56 & -0.20 \\ -0.10 & -0.46 \\ -0.19 & -0.78 \end{pmatrix} \begin{pmatrix} 8.4 & 0 \\ 0 & 3.3 \end{pmatrix} \begin{pmatrix} -0.41 & -0.49 & -0.44 & -0.61 & -0.10 & -0.12 \\ 0.21 & 0.31 & 0.26 & -0.37 & -0.44 & -0.68 \end{pmatrix}$$

$$= \begin{pmatrix} 1.55 & 1.87 & 1.67 & 1.91 & 0.10 & 0.04 \\ 2.46 & 2.98 & 2.66 & 2.95 & 0.10 & -0.03 \\ 0.89 & 1.08 & 0.96 & 1.04 & 0.01 & -0.04 \\ 1.81 & 2.11 & 1.91 & 3.14 & 0.77 & 1.03 \\ 0.02 & -0.05 & -0.02 & 1.06 & 0.74 & 1.11 \\ 0.10 & -0.02 & 0.04 & 1.89 & 1.28 & 1.92 \end{pmatrix}$$

The reconstructed matrix is a very good approximation of the original document-term matrix. Furthermore, each point gets a 2-dimensional embedding corresponding to the rows of $Q\Sigma$. It is clear that the reduced representations of the first three documents are quite similar, and so are the reduced representations of the last two. The reduced representation of the fourth document seems to be somewhere in the middle of the representations of the other documents. This is logical because the fourth document corresponds to both cars and cats. From this point of view, the reduced representation seems to satisfy the basic intuitions one would expect in terms of *relative* coordinates. However, one annoying characteristic of this representation is that it is hard to get any *absolute* semantic interpretation from the embedding. For example, it is difficult to match up the two latent vectors in P with the original concepts of cats and cars. The dominant latent vector in P is $[-0.41, -0.49, -0.44, -0.61, -0.10, -0.12]$, in which all components are negative. The second latent vector contains both positive and negative components. Therefore, the correspondence between the topics and the latent vectors is not very clear. A part of the problem is that the vectors have both positive and negative components, which reduces their interpretability. The lack of interpretability of singular value decomposition is its primary weakness, as a result of which other nonnegative forms of factorization are sometimes preferred.

9.2.2.2 Alternate Optima via Gradient Descent

SVD provides one of the alternate optima to the problem of unconstrained matrix factorization. As discussed above, the problem of unconstrained matrix factorization is defined as follows:

$$\text{Minimize}_{U,V}\ J = \frac{1}{2}\|D - UV^T\|_F^2$$

Here, D, U, and V are matrices of sizes n, d, and k, respectively. The value of k is typically much smaller than the rank of the matrix D. In this section, we will investigate a method that finds a solution to the unconstrained optimization problem with the use of gradient descent. This approach does not guarantee the orthogonal solutions provided by singular value decomposition; however, the formulation is equivalent and should (ideally) lead to a solution with the same objective function value. The approach also has the advantage that it can easily adapted to more difficult settings such as the presence of missing values in the matrix. A natural application of this type of approach is that of matrix factorization in recommender systems. Recommender systems use the same optimization formulation as SVD; however, the resulting basis vectors of the factorization are not guaranteed to be orthogonal.

In order to perform gradient descent, we need to compute the derivative of the unconstrained optimization problem with respect to the parameters in the matrices $U = [u_{iq}]$ and $V = [v_{jq}]$. The simplest approach is to compute the derivative of the objective function J with respect to each parameter in the matrices U and V. First, the objective function is expressed in terms of the individual entries in the various matrices. Let the (i, j)th entry of the $n \times d$ matrix D be denoted by x_{ij}. Then, the objective function can be restated in terms of the entries of the matrices D, U, and V as follows:

$$\text{Minimize } J = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{d}\left(x_{ij} - \sum_{s=1}^{k} u_{is} \cdot v_{js}\right)^2$$

The quantity $e_{ij} = x_{ij} - \sum_{s=1}^{k} u_{is} \cdot v_{js}$ is the error of the factorization for the (i, j)th entry. Note that the objective function J minimizes the sum of squares of e_{ij}. One can compute the

partial derivative of the objective function with respect to the parameters in the matrices U and V as follows:

$$\frac{\partial J}{\partial u_{iq}} = \sum_{j=1}^{d} \left(x_{ij} - \sum_{s=1}^{k} u_{is} \cdot v_{js} \right) (-v_{jq}) \quad \forall i \in \{1 \ldots n\}, q \in \{1 \ldots k\}$$
$$= \sum_{j=1}^{d} (e_{ij})(-v_{jq}) \quad \forall i \in \{1 \ldots n\}, q \in \{1 \ldots k\}$$
$$\frac{\partial J}{\partial v_{jq}} = \sum_{i=1}^{n} \left(x_{ij} - \sum_{s=1}^{k} u_{is} \cdot v_{js} \right) (-u_{iq}) \quad \forall j \in \{1 \ldots d\}, q \in \{1 \ldots k\}$$
$$= \sum_{i=1}^{n} (e_{ij})(-u_{iq}) \quad \forall j \in \{1 \ldots d\}, q \in \{1 \ldots k\}$$

One can also express these derivatives in terms of matrices. Let $E = [e_{ij}]$ be the $n \times d$ matrix of errors. In the denominator layout[2] of matrix calculus, the derivatives can be expressed as follows:

$$\frac{\partial J}{\partial U} = -(D - UV^T)V = -EV$$
$$\frac{\partial J}{\partial V} = -(D - UV^T)^T U = -E^T U$$

The above matrix calculus identity can be verified by using the relatively tedious process of expanding the (i, q)th and (j, q)th entries of each of the above matrices on the right-hand side, and showing that they are equivalent to the (corresponding) scalar derivatives $\frac{\partial J}{\partial u_{iq}}$ and $\frac{\partial J}{\partial v_{jq}}$. one can also find an optimal solution by using gradient descent. The updates for gradient descent are as follows:

$$U \Leftarrow U - \alpha \frac{\partial J}{\partial U} = U + \alpha EV$$
$$V \Leftarrow V - \alpha \frac{\partial J}{\partial V} = V + \alpha E^T U$$

Here, $\alpha > 0$ is the learning rate.

The optimization model is identical to that of SVD. If the aforementioned gradient descent method is used (instead of the power iteration method of the previous chapter), one will typically obtain solutions that are equally good in terms of objective function value, but for which the columns of U (or V) are not mutually orthogonal. The power iteration methods yields solutions with orthogonal columns. Although the standardized SVD solution with orthonormal columns is typically not obtained by gradient descent, the k columns of U will span the same subspace as the columns of Q_k, and the columns of V will span the same subspace as the columns of P_k. One can also add the regularization term $\lambda(\|U\|_F^2 + \|V\|_F^2)/2$ to the objective function. Here, λ is the regularization parameter. Adding the regularization term leads to the following updates:

$$U \Leftarrow U(1 - \alpha\lambda) + \alpha EV \tag{9.5}$$

[2]In this layout, the (i, j)th entry of $\frac{\partial J}{\partial U}$ is $\frac{\partial J}{\partial u_{ij}}$. In the numerator layout, the (i, j)th entry of $\frac{\partial J}{\partial U}$ is $\frac{\partial J}{\partial u_{ji}}$.

$$V \Leftarrow V(I - \alpha\lambda) + \alpha E^T U \tag{9.6}$$

The gradient-descent approach can be implemented efficiently when the matrix D is sparse by sampling entries from the matrix for making updates. This is essentially a *stochastic* gradient descent method. In other words, we sample an entry (i, j) and compute its error e_{ij}. Subsequently, we make the following updates to the ith row $\overline{u}_i$ of U and the jth row $\overline{v}_j$ of V, which are also referred to as *latent factors*:

$$\overline{u}_i \Leftarrow \overline{u}_i(1 - \alpha\lambda) + \alpha e_{ij}\overline{v}_j$$
$$\overline{v}_j \Leftarrow \overline{v}_j(1 - \alpha\lambda) + \alpha e_{ij}\overline{u}_i$$

One cycles through the sampled entries of the matrix (making the above updates) until convergence. The fact that we can sample entries of the matrix for updates means that we do not need fully specified matrices in order to learn the latent factors. This basic idea forms the foundations of recommender systems, in which partially specified matrices are used for learning the factors. The resulting factors are then used to reconstruct the entire matrix as UV^T.

9.2.3 Nonnegative Matrix Factorization

Nonnegative matrix factorization is a *highly interpretable* type of matrix factorization in which nonnegativity constraints are imposed on U and V. Therefore, this optimization problem is defined as follows:

$$\begin{aligned} &\text{Minimize}_{U,V} \|D - UV^T\|_F^2 \\ &\text{subject to:} \\ &U \geq 0,\ V \geq 0 \end{aligned}$$

As in the case of SVD, $U = [u_{ij}]$ is an $n \times k$ matrix and $V = [v_{ij}]$ is a $d \times k$ matrix of optimization parameters. Note that the optimization objective is the same but the constraints are different.

This type of constrained problem is often solved using Lagrangian relaxation. For the (i, s)th entry u_{is} in U, we introduce the Lagrange multiplier $\alpha_{is} \leq 0$, whereas for the (j, s)th entry v_{js} in V, we introduce the Lagrange multiplier $\beta_{js} \leq 0$. One can create a vector $(\overline{\alpha}, \overline{\beta})$ of dimensionality $(n+d) \cdot k$ by putting together all the Lagrangian parameters into a vector. Instead of using hard constraints on nonnegativity, Lagrangian relaxation uses penalties in order to relax the constraints into a softer version of the problem, which is defined by the augmented objective function L:

$$L = \|D - UV^T\|_F^2 + \sum_{i=1}^{n}\sum_{r=1}^{k} u_{ir}\alpha_{ir} + \sum_{j=1}^{d}\sum_{r=1}^{k} v_{jr}\beta_{jr} \tag{9.7}$$

Note that violation of the nonnegativity constraints always lead to a positive penalty because the Lagrangian parameters cannot be positive. According to the methodology of Lagrangian optimization, this augmented problem is really a minimax problem because we need to minimize L over all U and V at any particular value of the (vector of) Lagrangian parameters, but we then need to maximize these solutions over all valid values of the Lagrangian parameters α_{is} and β_{js}. In other words, we have:

$$\text{Max}_{\overline{\alpha}\leq 0, \overline{\beta}\leq 0} \text{Min}_{U,V}\, L \tag{9.8}$$

Here, $\overline{\alpha}$ and $\overline{\beta}$ represent the vectors of optimization parameters in α_{is} and β_{js}, respectively. This is a tricky optimization problem because of the way in which it is formulated with simultaneous maximization and minimization over different sets of parameters. The first step is to compute the gradient of the Lagrangian relaxation with respect to the (minimization) optimization variables u_{is} and v_{js}. Therefore, we have:

$$\frac{\partial L}{\partial u_{is}} = -(DV)_{is} + (UV^T V)_{is} + \alpha_{is} \qquad \forall i \in \{1, \ldots, n\}, s \in \{1, \ldots, k\} \tag{9.9}$$

$$\frac{\partial L}{\partial v_{js}} = -(D^T U)_{js} + (VU^T U)_{js} + \beta_{js} \qquad \forall j \in \{1, \ldots, d\}, s \in \{1, \ldots, k\} \tag{9.10}$$

The optimal value of the (relaxed) objective function at any particular value of the Lagrangian parameters is obtained by setting these partial derivatives to 0. As a result, we obtain the following conditions:

$$-(DV)_{is} + (UV^T V)_{is} + \alpha_{is} = 0 \qquad \forall i \in \{1, \ldots, n\}, s \in \{1, \ldots, k\} \tag{9.11}$$

$$-(D^T U)_{js} + (VU^T U)_{js} + \beta_{js} = 0 \qquad \forall j \in \{1, \ldots, d\}, s \in \{1, \ldots, k\} \tag{9.12}$$

We would like to eliminate the Lagrangian parameters and set up the optimization conditions purely in terms of U and V. It turns out the *Kuhn-Tucker optimality conditions* [18] are very helpful. These conditions are $u_{is}\alpha_{is} = 0$ and $v_{js}\beta_{js} = 0$ over all parameters. By multiplying Equation 9.11 with u_{is} and multiplying Equation 9.12 with v_{js}, we can use the Kuhn-Tucker conditions to get rid of these pesky Lagrangian parameters from the aforementioned equations. In other words, we have:

$$-(DV)_{is}u_{is} + (UV^T V)_{is}u_{is} + \underbrace{\alpha_{is}u_{is}}_{0} = 0 \qquad \forall i \in \{1, \ldots, n\}, s \in \{1, \ldots, k\} \tag{9.13}$$

$$-(D^T U)_{js}v_{js} + (VU^T U)_{js}v_{js} + \underbrace{\beta_{js}v_{js}}_{0} = 0 \qquad \forall j \in \{1, \ldots, d\}, s \in \{1, \ldots, k\} \tag{9.14}$$

One can rewrite these optimality conditions, so that a single parameter occurs on one side of the condition:

$$u_{is} = \frac{(DV)_{is}u_{is}}{(UV^T V)_{is}} \qquad \forall i \in \{1, \ldots, n\}, s \in \{1, \ldots, k\} \tag{9.15}$$

$$v_{js} = \frac{(D^T U)_{js}v_{js}}{(VU^T U)_{js}} \qquad \forall j \in \{1, \ldots, d\}, s \in \{1, \ldots, k\} \tag{9.16}$$

Even though these conditions are circular in nature (because the optimization parameters occur on both sides), they are natural candidates for iterative updates.

Therefore, the iterative approach starts by initializing the parameters in U and V to nonnegative random values in $(0, 1)$ and then uses the following updates derived from the aforementioned optimality conditions:

$$u_{is} \Leftarrow \frac{(DV)_{is}u_{is}}{(UV^T V)_{is}} \qquad \forall i \in \{1, \ldots, n\}, s \in \{1, \ldots, k\} \tag{9.17}$$

$$v_{js} \Leftarrow \frac{(D^T U)_{js}v_{js}}{(VU^T U)_{js}} \qquad \forall j \in \{1, \ldots, d\}, s \in \{1, \ldots, k\} \tag{9.18}$$

These iterations are then repeated to convergence. Improved initialization provides significant advantages, and the reader is referred to [109] for such methods. Numerical stability

can be improved by adding a small value $\epsilon > 0$ to the denominator during the updates:

$$u_{is} \Leftarrow \frac{(DV)_{is} u_{is}}{(UV^T V)_{is} + \epsilon} \qquad \forall i \in \{1, \ldots, n\}, s \in \{1, \ldots, k\} \tag{9.19}$$

$$v_{js} \Leftarrow \frac{(D^T U)_{js} v_{js}}{(VU^T U)_{js} + \epsilon} \qquad \forall j \in \{1, \ldots, d\}, s \in \{1, \ldots, k\} \tag{9.20}$$

One can also view ϵ as a type of *regularization parameter* whose primary goal is to avoid overfitting. Regularization is particularly helpful in small data sets.

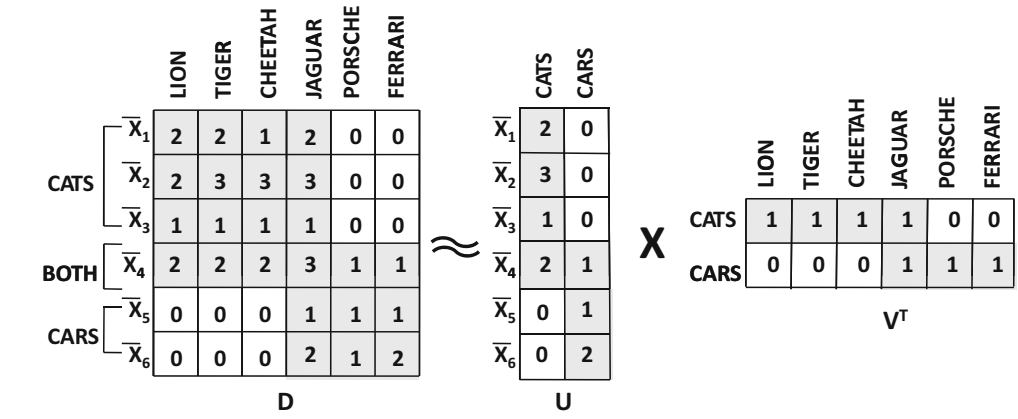
As in all other forms of matrix factorization, it is possible to convert the factorization UV^T into the three-way factorization $Q\Sigma P^T$ by normalizing each column of U and V to unit norm, and storing the product of these normalization factors in a diagonal matrix Σ (i.e., the ith diagonal entry contains the product of the ith normalization factors of U and V). It is common to use L_1-normalization on each column of U and V, or that the columns of the resulting matrices Q and P each sum to 1. Interestingly, this type of normalization makes nonnegative factorization similar to a closely related factorization known as *Probabilistic Semantic Analysis (PLSA)*. The main difference between PLSA and nonnegative matrix factorization is that the former uses a maximum likelihood optimization function whereas nonnegative matrix factorization (typically) uses the Frobenius norm.

9.2.3.1 Interpreting Nonnegative Matrix Factorization

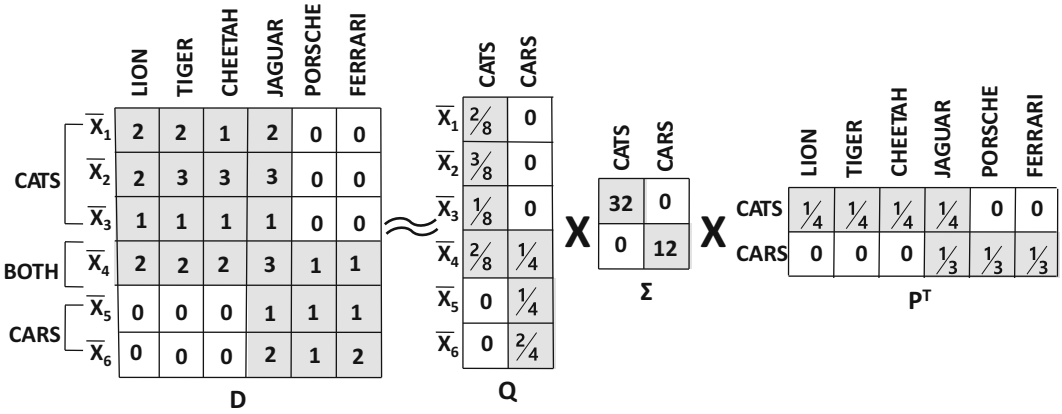
An important property of nonnegative matrix factorization is that it is highly interpretable in terms of the clusters in the underlying data. It is easiest to understand this point in terms of (semantically interpretable) document data. Consider the case in which the matrix D is an $n \times d$ matrix with n documents and d words (terms). The rth columns U_r and V_r of each of U and V respectively contain document- and word-membership information about the rth topic (or cluster) in the data. The n entries in U_r correspond to the nonnegative components (coordinates) of the n documents along the rth topic. If a document strongly belongs to topic r, then it will have a very positive coordinate in U_r. Otherwise, its coordinate will be zero or mildly positive (representing noise). Similarly, the rth column V_r of V provides the frequent vocabulary of the rth cluster. Terms that are highly related to a particular topic will have large components in V_r. The k-dimensional representation of each document is provided by the corresponding row of U. This approach allows a document to belong to multiple clusters, because a given row in U might have multiple positive coordinates. For example, if a document discusses both science and history, it will have components along latent components with science-related and history-related vocabularies. This provides a more realistic "sum-of-parts" decomposition of the corpus along various topics, which is primarily enabled by the nonnegativity of U and V. In fact, one can create a decomposition of the document-term matrix into k different rank-1 document-term matrices corresponding to the k topics captured by the decomposition. Let us treat U_r as an $n \times 1$ matrix and V_r as a $d \times 1$ matrix. If the rth component is related to science, then $U_r V_r^T$ is an $n \times d$ document-term matrix containing the science-related portion of the original corpus. Then the decomposition of the document-term matrix is defined as the sum of the following components:

$$D \approx \sum_{r=1}^{k} U_r V_r^T \tag{9.21}$$

This decomposition is analogous to the spectral decomposition of SVD, except that its nonnegativity often gives it much better correspondence to semantically related topics.



(a) Two-way factorization



(b) Three-way factorization by applying L_1-normalization to (a) above

Figure 9.2: The highly interpretable decomposition of nonnegative matrix factorization

In order to illustrate the semantic interpretability of nonnegative matrix factorization, let us revisit the same example used in Section 9.2.2.1 on singular value decomposition, and create a decomposition in terms of nonnegative matrix factorization:

$$D = \begin{pmatrix} & \text{lion} & \text{tiger} & \text{cheetah} & \text{jaguar} & \text{porsche} & \text{ferrari} \\ \text{Document-1} & 2 & 2 & 1 & 2 & 0 & 0 \\ \text{Document-2} & 2 & 3 & 3 & 3 & 0 & 0 \\ \text{Document-3} & 1 & 1 & 1 & 1 & 0 & 0 \\ \text{Document-4} & 2 & 2 & 2 & 3 & 1 & 1 \\ \text{Document-5} & 0 & 0 & 0 & 1 & 1 & 1 \\ \text{Document-6} & 0 & 0 & 0 & 2 & 1 & 2 \end{pmatrix}$$

This matrix represents topics related to both cars and cats. The first three documents are primarily related to cats, the fourth is related to both, and the last two are primarily related to cars. The word "jaguar" is polysemous because it could correspond to either a car or a cat and is present in documents of both topics.

A highly interpretable nonnegative factorization of rank-2 is shown in Figure 9.2(a). We

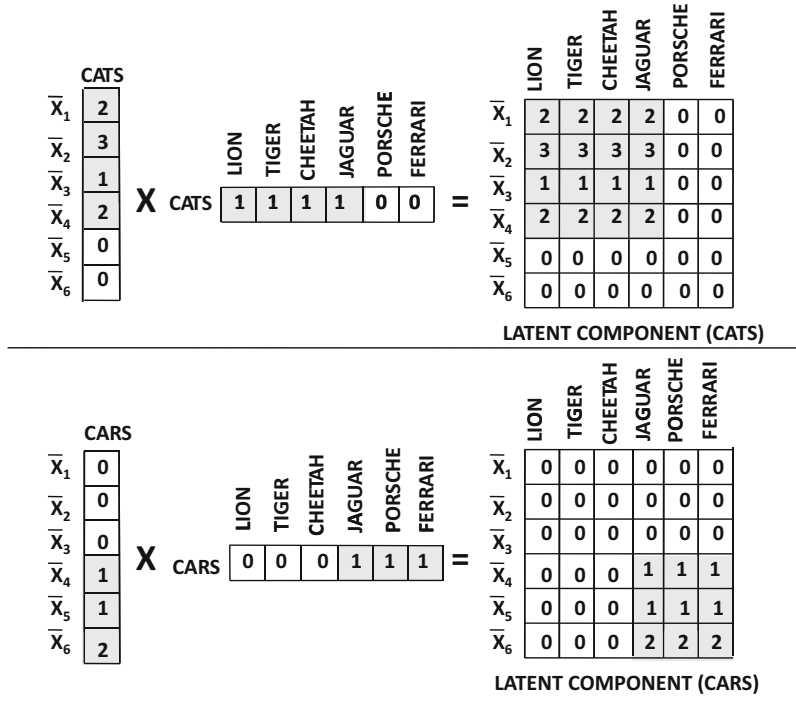


Figure 9.3: The highly interpretable "sum-of-parts" decomposition of the document-term matrix into rank-1 matrices representing different topics

have shown an approximate decomposition containing only integers for simplicity, although the optimal solution would (almost always) be dominated by floating point numbers in practice. It is clear that the first latent concept is related to cats and the second latent concept is related to cars. Furthermore, documents are represented by two non-negative coordinates indicating their affinity to the two topics. Correspondingly, the first three documents have strong positive coordinates for cats, the fourth has strong positive coordinates in both, and the last two belong only to cars. The matrix V tells us that the vocabularies of the various topics are as follows:

Cats: lion, tiger, cheetah, jaguar
Cars: jaguar, porsche, ferrari

It is noteworthy that the polysemous word "jaguar" is included in the vocabulary of both topics, and its usage is automatically inferred from its context (i.e., other words in document) during the factorization process. This fact becomes especially evident when we decompose the original matrix into two rank-1 matrices according to Equation 9.21. This decomposition is shown in Figure 9.3 in which the rank-1 matrices for cats and cars are shown. It is particularly interesting that the occurrences of the polysemous word "jaguar" are nicely divided up into the two topics, which roughly correspond with their usage in these topics.

Any two-way matrix factorization can be converted into a standardized three-way factorization by normalizing the columns of U and V, so that they add to 1, and creating a diagonal matrix from the product of these normalization factors. The three-way normalized

representation is shown in Figure 9.2(b), and it tells us a little bit more about the relative frequencies of the two topics. Since the diagonal entry in Σ is 32 for cats in comparison with 12 for cars, it indicates that the topic of cats is more dominant than cars. This is consistent with the observation that more documents and terms in the collection are associated with cats as compared to cars.

9.2.4 Dimensionality Reduction with Neural Networks

Just as most of the supervised learning methods can be represented using neural networks, it is also possible to represent most of the unsupervised methods with neural networks. A key point is that all forms of learning, including unsupervised learning, can be represented in the form of input-to-output mappings, which creates a computational graph (and therefore a neural architecture as well). Autoencoders represent a fundamental architecture that is used for various types of unsupervised learning applications. In these models, the inputs and outputs are the same. In other words, the inputs are *replicated* to the outputs, with intervening layers that have fewer nodes, so that precise copying from layer to layer is not possible. In other words, the data is compressed in the intermediate layers before being replicated to the outermost layer.

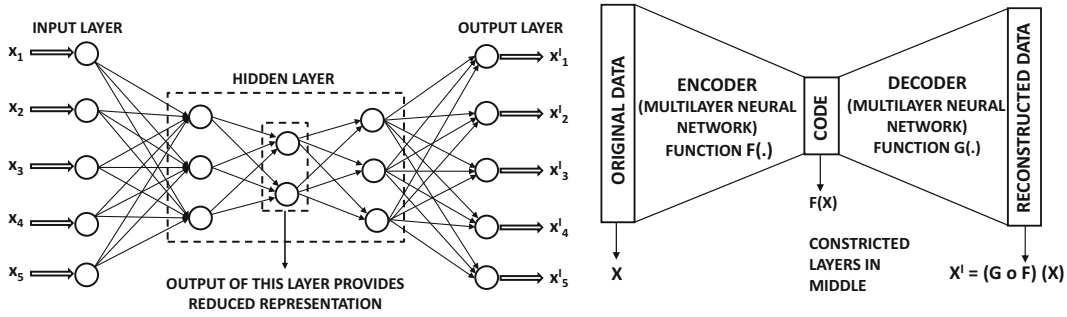
The simplest autoencoders with linear layers map to well-known dimensionality reduction techniques like singular value decomposition. However, deep autoencoders with nonlinearity map to complex models that might not exist in traditional machine learning. Therefore, the goal of this section is to show two things:

1. Classical dimensionality reduction methods like singular value decomposition are special cases of shallow neural architectures.

2. By adding depth and nonlinearity to the basic architecture, one can generate sophisticated nonlinear embeddings of the data. While nonlinear embeddings are also available in traditional machine learning, the latter is limited to loss functions that can be expressed compactly in closed form. The loss functions of deep neural architectures are no longer compact; however, they provide unprecedented flexibility in controlling the properties of the unsupervised representation by making various types of architectural changes (and allowing backpropagation to take care of the complexities of differentiation).

The basic idea of an autoencoder is to have an output layer with the same dimensionality as the inputs. The idea is to try to reconstruct the input data instance. An autoencoder *replicates* the data from the input to the output, and is therefore sometimes referred to as a *replicator neural network*. Although reconstructing the data might seem like a trivial matter by simply copying the data forward from one layer to another, this is not possible when the number of units in the middle are *constricted*. In other words, the number of units in each middle layer is typically fewer than that in the input (or output). As a result, one cannot simply copy the data from one layer to another. Therefore, the activations in the constricted layers hold a reduced representation of the data; the net effect is that this type of reconstruction is inherently *lossy*. This general representation of the autoencoder is given in Figure 9.4(a), where an architecture is shown with three constricted layers. Note that the output layer has the same number of units as the input layer. The loss function of this neural network uses the sum-of-squared differences between the input and the output feature values in order to force the output to be as similar as possible to the input.

It is common (but not necessary) for an M-layer autoencoder to have a symmetric architecture between the input and output, where the number of units in the kth layer is



(a) Three hidden layers (b) General schematic

Figure 9.4: The basic schematic of the autoencoder

the same as that in the $(M-k+1)$th layer. Furthermore, the value of M is often odd, as a result of which the $(M+1)/2$th layer is the most constricted layer. Here, we are counting the (non-computational) input layer as the first layer, and therefore the minimum number of layers in an autoencoder would be three, corresponding to the input layer, constricted layer, and the output layer.

The reduced representation of the data in the most constricted layer is also sometimes referred to as the *code*, and the number of units in this layer is the dimensionality of the reduction. The initial part of the neural architecture before the bottleneck is referred to as the *encoder* (because it creates a reduced code), and the final part of the architecture is referred to as the *decoder* (because it reconstructs from the code). The general schematic of the autoencoder is shown in Figure 9.4(b).

9.2.4.1 Linear Autoencoder with a Single Hidden Layer

A single hidden-layer autoencoder can be understood in the context of matrix factorization. In matrix factorization, we want to factorize the $n \times d$ matrix D into an $n \times k$ matrix U and a $d \times k$ matrix V:

$$D \approx UV^T \tag{9.22}$$

Here, $k \ll n$ is the rank of the factorization. As discussed earlier in this chapter, the objective function of traditional matrix factorization is as follows:

$$\text{Minimize } J = \|D - UV^T\|_F^2$$

Here, the notation $\|\cdot\|_F$ indicates the Frobenius norm. The parameter matrices U and V need to be learned in order to optimize the aforementioned error. Although the gradient-descent steps have already been discussed in earlier sections, our goal here is to capture this optimization problem within a neural architecture. Going through this exercise helps us show that simple matrix factorization is a special case of an autoencoder architecture, which sets the stage for understanding the gains obtained with deeper autoencoders.

From the perspective of neural networks, one would need to design the architecture in such a way that the rows of D are fed to the neural network, and the reduced rows of U are obtained from the constricted layer when the original data is fed to the neural network. The single-layer autoencoder is illustrated in Figure 9.5, where the hidden layer contains k units. The rows of D are input into the autoencoder, whereas the k-dimensional rows of U are the activations of the hidden layer. Note that the $k \times d$ matrix of weights in the decoder

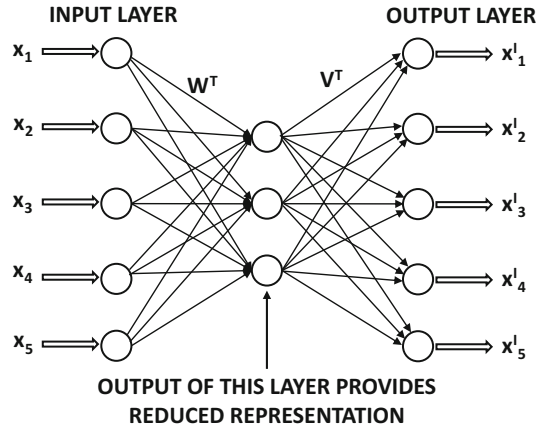


Figure 9.5: A basic autoencoder with a single layer

must be V^T, since it is necessary to be able to multiply the rows of U to reconstruct the rows of $D \approx UV^T$ according to the optimization model discussed above.

The vector of values in a particular layer of the network can be obtained by multiplying the vector of values in the previous layer with the matrix of weights connecting the two layers (with linear activation) Here, $\overline{u}_i$ is the row vector containing the ith row of U and $\overline{X}'_i$ is the reconstruction of the ith row $\overline{X}_i$ of D. One first encodes $\overline{X}_i$ with the matrix W^T, and then decodes it back using V^T:

$$\overline{u}_i = \overline{X}_i W^T \tag{9.23}$$

$$\overline{X}'_i = \overline{u}_i V^T \tag{9.24}$$

It is not difficult to see that Equation 9.24 is a row-wise variant of Equation 9.22. The autoencoder minimizes the sum-of-squared differences between the input and the output, which is equivalent to minimizing $\|D - UV^T\|_F^2$. Therefore, the neural architecture achieves precisely the goals of matrix factorization in terms of the loss value that is optimized. In fact, the basis vectors in V can be shown to span the same subspace as the top-k basis vectors of SVD. However, the optimization problem for SVD has multiple global optima, and the SVD only corresponds to an optimum in which the columns of V are orthonormal and those of U are mutually orthogonal. The neural network might find a different basis of the subspace spanned by the top-k basis vectors of SVD, and will adjust U accordingly so that the reconstruction UV^T remains unaffected.

9.2.4.2 Nonlinear Activations

So far, the discussion has focused on simulating singular value decomposition using a neural architecture. The real power of autoencoders is realized when one starts using nonlinear activations and multiple layers. For example, consider a situation in which the matrix D is binary. In such a case, one can use the same neural architecture as shown in Figure 9.5, but one can also use a sigmoid function in the final layer to predict the output. This sigmoid layer is combined with negative log loss. Therefore, for a binary matrix $B = [b_{ij}]$, the model assumes the following:

$$B \sim \text{sigmoid}(UV^T) \tag{9.25}$$

Here, the sigmoid function is applied in element-wise fashion. Note the use of $\sim$ instead of $\approx$ in the above expression, which indicates that the binary matrix B is an instantiation of random draws from Bernoulli distributions with corresponding parameters contained in sigmoid(UV^T). The resulting factorization can be shown to be equivalent to *logistic matrix factorization*. The basic idea is that the (i, j)th element of UV^T is the parameter of a Bernoulli distribution, and the binary entry b_{ij} is generated from a Bernoulli distribution with these parameters. Therefore, U and V are learned using log-likelihood loss. The log-likelihood loss implicitly tries to find parameter matrices U and V so that the probability of the matrix B being generated by these parameters is maximized.

Logistic matrix factorization has only recently been proposed [92] as a sophisticated matrix factorization method for binary data, which is useful for recommender systems with *implicit feedback* ratings. Implicit feedback refers to the binary actions of users such as buying or not buying specific items. The solution methodology of this recent work on logistic matrix factorization [92] seems to be vastly different from SVD, and it is not based on a neural network approach. However, for a neural network practitioner, the change from the SVD model to that of logistic matrix factorization is a relatively small one, where only the final layer of the neural network needs to be changed. It is this modular nature of neural networks that makes them so attractive to engineers and encourages all types of experimentation.

The real power of autoencoders in the neural network domain is realized when deeper variants with nonlinear activations in hidden layers are used. For example, an autoencoder with three hidden layers is shown in Figure 9.4(a). One can increase the number of intermediate layers in order to further increase the representation power of the neural network. It is noteworthy that it is essential for some of the layers of the deep autoencoder to use a nonlinear activation function to increase its representation power. As shown in Chapter 7, no additional power is gained by a multilayer network when only linear activations are used.

Deep networks with multiple layers provide an extraordinary amount of representation power. The multiple layers of this network provide *hierarchically* reduced representations of the data. For some data domains like images, hierarchically reduced representations are particularly natural. Note that there is no precise analog of this type of model in traditional machine learning, and the backpropagation approach rescues us from the challenges associated in computing the complicated gradient-descent steps. A nonlinear dimensionality reduction might map a manifold of arbitrary shape into a reduced representation. Although several methods for nonlinear dimensionality reduction are known in machine learning, neural networks have some advantages over these methods:

- Many nonlinear dimensionality reduction methods have a very hard time mapping out-of-sample data points to reduced representations, unless these points are included in the training data up front. On the other hand, it is a relatively simple matter to compute the reduced representation of an out-of-sample point by passing it through the network.

- Neural networks allow more power and flexibility in the nonlinear data reduction by varying on the number and type of layers used in intermediate stages. Furthermore, by choosing specific types of activation functions in particular layers, one can engineer the nature of the reduction to the properties of the data. For example, it makes sense to use a logistic output layer with logarithmic loss for a binary data set. Indeed, for multilayer variants of the autoencoder, an exact counterpart often does not even exist in traditional machine learning. This seems to suggest that it is often more natural to

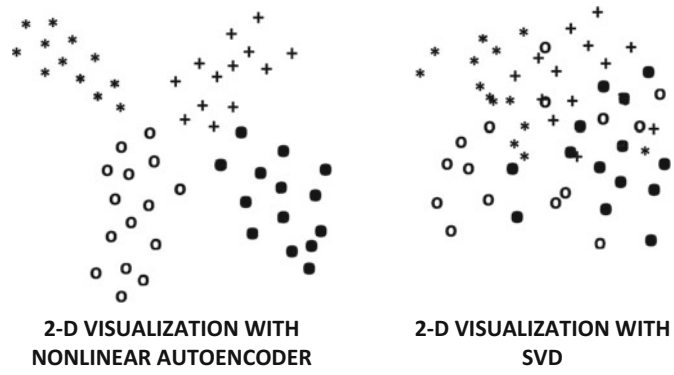


Figure 9.6: A depiction of the typical difference between the embeddings created by nonlinear autoencoders and singular value decomposition (SVD). Nonlinear and deep autoencoders are often able to separate out the entangled class structures in the underlying data, which is not possible within the constraints of linear transformations like SVD

discover sophisticated machine learning algorithms when working with the modular approach of constructing multilayer neural networks.

It is possible to achieve extraordinarily compact reductions by using this approach. Greater reduction is always achieved by using nonlinear units, which implicitly map warped manifolds into linear hyperplanes. The superior reduction in these cases is because it is easier to thread a warped surface (as opposed to a linear surface) through a larger number of points. This property of nonlinear autoencoders is often used for 2-dimensional visualizations of the data by creating a deep autoencoder in which the most compact hidden layer has only two dimensions. These two dimensions can then be mapped on a plane to visualize the points.

An illustrative example of the typical behavior of real data distributions is shown in Figure 9.6, in which the 2-dimensional mapping created by a deep autoencoder seems to clearly separate out the different classes. On the other hand, the mapping created by SVD does not seem to separate the classes well. In many cases, the data may contain heavily entangled spirals (or other shapes) that belong to different classes. Linear dimensionality reduction methods cannot attain clear separation because nonlinearly entangled shapes are not linearly separable. On the other hand, deep autoencoders with nonlinearity are far more powerful and able to disentangle such shapes.

9.3 Clustering

The problem of clustering segments data records into groups, so that similar records are placed in the same group. While dimensionality reduction methods are more deeply focused on relationships among the entries of the matrix, clustering specifically focuses on the rows. Nevertheless, some methods like nonnegative matrix factorization can be used for both clustering and dimensionality reduction. Clustering methods are either *flat* or *hierarchical*. In flat clustering, the data set is partitioned into a set of clusters in one shot, and no hierarchical relationships exist between clusters. In hierarchical clustering, the clusters are organized in tree-like fashion as a taxonomy. This section will discuss several clustering methods, some of which are flat, whereas others are hierarchical.

9.3.1 Representative-Based Algorithms

As the name implies, representative-based algorithms rely on distances (or similarities) to representative points for clustering. Such representative points serve as *prototypes* to which most cluster points are similar. Representative-based algorithms create flat clusters and they do not have hierarchical relationships among them. The partitioning representatives may either be created as a function of the data points in the clusters (e.g., the mean) or may be selected from the existing data points in the cluster. The main insight of these methods is that the discovery of high-quality clusters in the data is equivalent to discovering a high-quality set of representatives. Once the representatives have been determined, a distance function can be used to assign the data points to their closest representatives. This category of methods contains several methods such as the k-means, k-medoids, and the k-medians algorithms. In this section, we will primarily discuss the k-means algorithm.

Typically, it is assumed that the number of clusters, denoted by k, is specified by the user. Consider a data set $\mathcal{D}$ containing n data points denoted by $\overline{X}_1 \ldots \overline{X}_n$ in d-dimensional space. The goal is to determine k representatives $\overline{Y}_1 \ldots \overline{Y}_k$ that minimize the following objective function O:

$$O = \sum_{i=1}^{n} \left[\min_j Dist(\overline{X}_i, \overline{Y}_j)\right] \tag{9.26}$$

In other words, the sum of the distances of the different data points to their closest representatives needs to be minimized. Note that the assignment of data points to representatives depends on the choice of the representatives $\overline{Y}_1 \ldots \overline{Y}_k$. In some variations of representative algorithms, such as k-medoid algorithms, it is assumed that the representatives $\overline{Y_1} \ldots \overline{Y_k}$ are drawn from the original database $\mathcal{D}$, although this will obviously not provide an optimal solution. In general, the discussion in this section will not automatically assume that the representatives are drawn from the original database $\mathcal{D}$, unless specified otherwise.

One observation about the formulation of Equation 9.26 is that the representatives $\overline{Y}_1 \ldots \overline{Y}_k$ and the optimal assignment of data points to representatives are unknown *a priori*, but they depend on each other in a circular way. For example, if the optimal representatives are known, then the optimal assignment is easy to determine, and vice versa. Such optimization problems are solved with the use of an iterative approach where candidate representatives and candidate assignments are used to improve each other. Therefore, the generic k-representatives approach starts by initializing the k representatives S with the use of a straightforward heuristic (such as random sampling from the original data), and then refines the representatives and the clustering assignment, iteratively, as follows:

- (Assign step) Assign each data point to its closest representative in S using distance function $Dist(\cdot, \cdot)$, and denote the corresponding clusters by $\mathcal{C}_1 \ldots \mathcal{C}_k$.
- (Optimize step) Determine the optimal representative $\overline{Y}_j$ for each cluster $\mathcal{C}_j$ that minimizes its *local* objective function $\sum_{\overline{X}_i \in \mathcal{C}_j} \left[Dist(\overline{X}_i, \overline{Y}_j)\right]$.

It will be evident later in this chapter that this two-step procedure is very closely related to generative models of cluster analysis in the form of *expectation-maximization* algorithms. The second step of *local* optimization is simplified by this two-step iterative approach because it no longer depends on an unknown assignment of data points to clusters as in the global optimization problem of Equation 9.26. Typically, the optimized representative can be shown to be some central measure of the data points in the jth cluster $\mathcal{C}_j$, and the precise measure depends on the choice of the distance function $Dist(\overline{X}_i, \overline{Y}_j)$. In particular, for the case of the Euclidean distance and cosine similarity functions, it can be shown that the

Algorithm *GenericRepresentative*(Database: $\mathcal{D}$, Number of Representatives: k)
begin
Initialize representative set S;
repeat
Create clusters $(\mathcal{C}_1 \ldots \mathcal{C}_k)$ by assigning each point in $\mathcal{D}$ to closest representative in S using the distance function $Dist(\cdot, \cdot)$;
Recreate set S by determining one representative $\overline{Y_j}$ for each $\mathcal{C}_j$ that minimizes $\sum_{\overline{X}_i \in \mathcal{C}_j} Dist(\overline{X}_i, \overline{Y_j})$;
until convergence;
return $(\mathcal{C}_1 \ldots \mathcal{C}_k)$;
end

Figure 9.7: Generic representative algorithm with unspecified distance function

optimal centralized representative of each cluster is its mean. However, different distance functions may lead to a slightly different type of centralized representative, and these lead to different variations of this broader approach, such as the k-means and k-medians algorithms. Thus, the k-representative approach defines a *family* of algorithms, in which minor changes to the basic framework allow the use of different distance criteria. The generic framework for representative-based algorithms with an unspecified distance function is illustrated in the pseudocode of Figure 9.7. The idea is to improve the objective function over multiple iterations. Typically, the increase is significant in early iterations, but it slows down in later iterations. When the improvement in the objective function in an iteration is less than a user-defined threshold, the algorithm may be allowed to terminate. The primary computational bottleneck of the approach is the assignment step where the distances need to be computed between all point-representative pairs. The time complexity of each iteration is $O(k \cdot n \cdot d)$ for a data set of size n and dimensionality d. The algorithm typically terminates in a small constant number of iterations.

The k-Means Algorithm

In the k-means algorithm, the sum of the squares of the Euclidean distances of data points to their closest representatives is used to quantify the objective function of the clustering. Therefore, we have:

$$Dist(\overline{X}_i, \overline{Y}_j) = \|\overline{X}_i - \overline{Y}_j\|_2^2 \tag{9.27}$$

Here, $\| \cdot \|_p$ represents the L_p-norm. The expression $Dist(\overline{X}_i, \overline{Y_j})$ can be viewed as the squared error of approximating a data point with its closest representative. Thus, the overall objective minimizes the sum of square errors over different data points. This is also sometimes referred to as SSE. In such a case, it can be shown[3] that the *optimal representative $\overline{Y}_j$ for each of the "optimize" iterative steps is the mean of the data points in cluster $\mathcal{C}_j$*. Thus, the only difference between the generic pseudocode of Figure 9.7 and a k-means pseudocode is the specific instantiation of the distance function $Dist(\cdot, \cdot)$, and the choice of the representative as the local mean of its cluster.

[3]For a *fixed* cluster assignment $\mathcal{C}_1 \ldots \mathcal{C}_k$, the gradient of the clustering objective function $\sum_{j=1}^{k} \sum_{\overline{X}_i \in \mathcal{C}_j} \|\overline{X}_i - \overline{Y}_j\|^2$ with respect to $\overline{Y}_j$ is $2 \sum_{\overline{X}_i \in \mathcal{C}_j} (\overline{X}_i - \overline{Y}_j)$. Setting the gradient to 0 yields the mean of cluster $\mathcal{C}_j$ as the optimum value of $\overline{Y}_j$. Note that the other clusters do not contribute to the gradient, and therefore the approach effectively optimizes the local clustering objective function for $\mathcal{C}_j$.

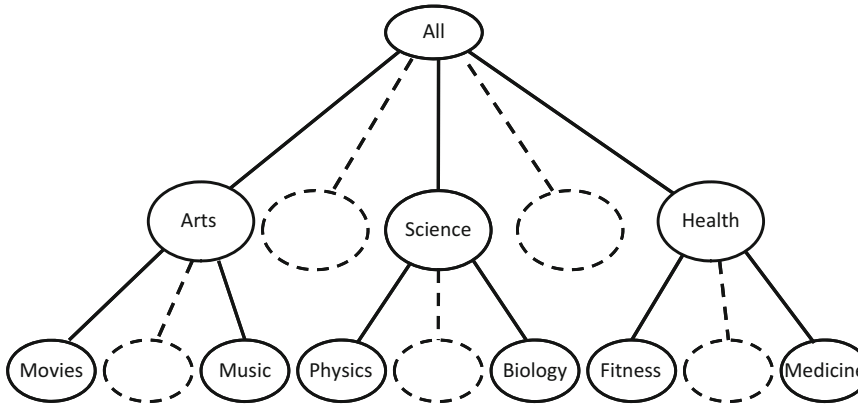


Figure 9.8: Multigranularity insights from hierarchical clustering

9.3.2 Bottom-up Agglomerative Methods

Hierarchical clustering algorithms create clusters that are arranged in the form of a hierarchical taxonomy. Although it is common to perform the clustering with the use of distances, many algorithms use other techniques, such as density- or graph-based methods, as a subroutine for constructing the hierarchy.

Hierarchical clustering algorithms are useful because different levels of clustering granularity provide different application-specific insights. This provides a *taxonomy* of clusters, which may be browsed for semantic insights. As a specific example, consider the taxonomy[4] of Web pages created by the well-known *Open Directory Project (ODP)*. In this case, the clustering has been created by a manual volunteer effort, but it nevertheless provides a good understanding of the multi-granularity insights that may be obtained with such an approach. A small portion of the hierarchical organization is illustrated in Figure 9.8. At the highest level, the Web pages are organized into topics such as arts, science, health, and so on. At the next level, the topic of science is organized into subtopics, such as biology and physics, whereas the topic of health is divided into topics such as fitness and medicine. This organization makes manual browsing very convenient for a user, especially when the content of the clusters can be described in a semantically comprehensible way. In other cases, such hierarchical organizations can be used by indexing algorithms. Furthermore, such methods can sometimes also be used for creating better "flat" clusters. There are two types of hierarchical clustering algorithms, referred to as *bottom-up hierarchical methods* and *top-down hierarchical methods*. This section will discuss bottom-up hierarchical methods, and the next section will discuss top-down methods.

In bottom-up methods, the data points are successively agglomerated into higher-level clusters. The algorithm starts with individual data points in their own clusters and successively agglomerates them into higher-level clusters. In each iteration, two clusters are selected that are deemed to be as close as possible. These clusters are merged and replaced with a newly created merged cluster. Thus, each merging step reduces the number of clusters by 1. Therefore, a method needs to be designed for measuring proximity between clusters containing multiple data points, so that they may be merged. It is in this choice of computing the distances between clusters, that most of the variations among different methods arise.

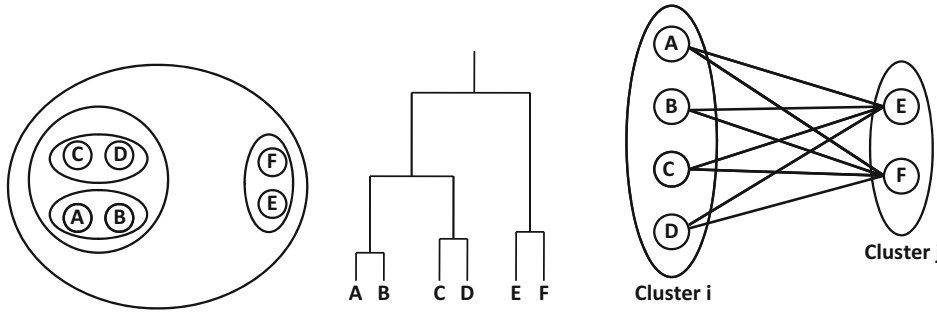
[4] http://www.dmoz.org

```
Algorithm AgglomerativeMerge (Data: D)
begin
  Initialize n × n distance matrix M using D;
  repeat
    Pick closest pair of clusters i and j using M;
    Merge clusters i and j;
    Delete rows/columns i and j from M and create
      a new row and column for newly merged cluster;
    Update the entries of new row and column of M;
  until termination criterion;
  return current merged cluster set;
end
```

Figure 9.9: Generic agglomerative merging algorithm with unspecified merging criterion

Let n be the number of data points in the d-dimensional database $\mathcal{D}$, and $n_t = n - t$ be the number of clusters after t agglomerations. At any given point, the method maintains an $n_t \times n_t$ distance matrix M between the current *clusters* in the data. The precise methodology for computing and maintaining this distance matrix will be described later. In any given iteration of the algorithm, the (non-diagonal) entry in the distance-matrix with the least distance is selected, and the corresponding clusters are merged. This merging will require the distance matrix to be updated to a smaller $(n_t - 1) \times (n_t - 1)$ matrix. The dimensionality reduces by 1 because the rows and columns for the two merged clusters need to be deleted, and a new row and column of distances, corresponding to the newly created cluster, needs to be added to the matrix. This corresponds to the newly created cluster in the data. The algorithm for determining the values of this newly created row and column depends on the cluster-to-cluster distance computation in the merging procedure and will be described later. The incremental update process of the distance matrix is a more efficient option than that of computing all distances from scratch. It is, of course, assumed that sufficient memory is available to maintain the distance matrix. If this is not the case, then the distance matrix will need to be fully re-computed in each iteration, and such agglomerative methods become less attractive. For termination, either a maximum threshold can be used on the distances between two merged clusters, or a minimum threshold can be used on the number of clusters at termination. The former criterion is designed to automatically determine the natural number of clusters in the data but has the disadvantage of requiring the specification of a quality threshold that is hard to guess intuitively. The latter criterion has the advantage of being intuitively interpretable in terms of the number of clusters in the data. The order of merging naturally creates a hierarchical tree-like structure illustrating the relationship between different clusters, which is referred to as a *dendrogram*. An example of a dendrogram on successive merges on six data points, denoted by A, B, C, D, E, and F, is illustrated in Figure 9.10(a).

The generic agglomerative procedure with an unspecified merging criterion is illustrated in Figure 9.9. The distances are encoded in the $n_t \times n_t$ distance matrix M. This matrix provides the pairwise cluster distances computed with the use of the merging criterion. The different choices for the merging criteria will be described later. The merging of two clusters corresponding to rows (columns) i and j in the matrix M requires the computation of some measure of distances between their constituent objects. For two clusters containing m_i and m_j objects, respectively, there are $m_i \cdot m_j$ pairs of distances between constituent objects. For example, in Figure 9.10(b), there are $2 \times 4 = 8$ pairs of distances between the constituent



(a) Dendrogram (b) Group similarity computation

Figure 9.10: Illustration of hierarchical clustering steps

objects, which are illustrated by the corresponding edges. The overall distance between the two clusters needs to be computed as a function of these $m_i \cdot m_j$ pairs. In the following, different ways of computing the distances will be discussed.

9.3.2.1 Group-Based Statistics

The following discussion assumes that the indices of the two clusters to be merged are denoted by i and j, respectively. In group-based criteria, the distance between two groups of objects is computed as a function of the $m_i \cdot m_j$ pairs of distances among the constituent objects. The different ways of computing distances between two groups of objects are as follows:

1. *Best (single) linkage:* In this case, the distance is equal to the minimum distance between all $m_i \cdot m_j$ pairs of objects. This corresponds to the closest pair of objects between the two groups. After performing the merge, the matrix M of pairwise distances needs to be updated. The ith and jth rows and columns are deleted and replaced with a single row and column representing the merged cluster. The new row (column) can be computed using the minimum of the values in the previously deleted pair of rows (columns) in M. This is because the distance of the other clusters to the merged cluster is the minimum of their distances to the individual clusters in the best-linkage scenario. For any other cluster $k \neq i, j$, this is equal to $\min\{M_{ik}, M_{jk}\}$ (for rows), and $\min\{M_{ki}, M_{kj}\}$ (for columns). The indices of the rows and columns are then updated to account for the deletion of the two clusters and their replacement with a new one. The best linkage approach is one of the instantiations of agglomerative methods that is very good at discovering clusters of arbitrary shape. This is because the data points in clusters of arbitrary shape can be successively merged with chains of data point pairs at small pairwise distances to each other. On the other hand, such chaining may also inappropriately merge distinct clusters when it results from noisy points.

2. *Worst (complete) linkage:* In this case, the distance between two groups of objects is equal to the maximum distance between all $m_i \cdot m_j$ pairs of objects in the two groups. This corresponds to the farthest pair in the two groups. Correspondingly, the matrix M is updated using the maximum values of the rows (columns) in this case. For any value of $k \neq i, j$, this is equal to $\max\{M_{ik}, M_{jk}\}$ (for rows), and $\max\{M_{ki}, M_{kj}\}$ (for columns). The worst-linkage criterion implicitly attempts to minimize the maximum

diameter of a cluster, as defined by the largest distance between any pair of points in the cluster. This method is also referred to as the *complete linkage* method.

3. *Group-average linkage:* In this case, the distance between two groups of objects is equal to the average distance between all $m_i \cdot m_j$ pairs of objects in the groups. To compute the row (column) for the merged cluster in M, a weighted average of the ith and jth rows (columns) in the matrix M is used. For any value of $k \neq i, j$, this is equal to $\frac{m_i \cdot M_{ik} + m_j \cdot M_{jk}}{m_i + m_j}$ (for rows), and $\frac{m_i \cdot M_{ki} + m_j \cdot M_{kj}}{m_i + m_j}$ (for columns).

4. *Closest centroid:* In this case, the closest centroids are merged in each iteration. This approach is not desirable, however, because the centroids lose information about the relative spreads of the different clusters. For example, such a method will not discriminate between merging pairs of clusters of varying sizes, as long as their centroid pairs are at the same distance. Typically, there is a bias towards merging pairs of larger clusters because centroids of larger clusters are statistically more likely to be closer to each other.

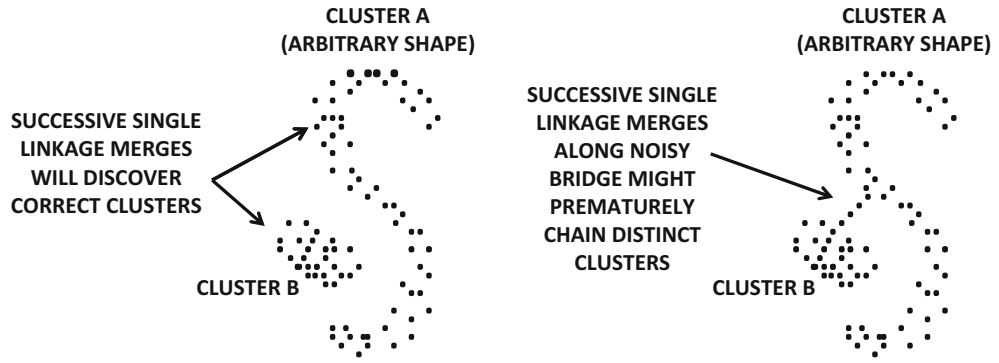
5. *Variance-based criterion:* This criterion minimizes the *change* in the objective function (such as cluster variance) as a result of the merging. Merging always results in a worsening of the clustering objective function value because of loss of granularity. In this case, the clusters are merged, so that the change in the objective function as a result of merging is as little as possible.

6. *Ward's method:* Instead of using the change in variance, one might also use the (unscaled) sum of squared error as the merging criterion. Surprisingly, this approach is a variant of the centroid method. The objective function for merging is obtained by multiplying the (squared) Euclidean distance between centroids with the harmonic mean of the number of points in each of the pair. Because larger clusters are penalized by this additional factor, the approach performs more effectively than the centroid method.

The various criteria have different advantages and disadvantages. For example, the single linkage method is able to successively merge chains of closely related points to discover clusters of arbitrary shape. However, this property can also (inappropriately) merge two unrelated clusters, when the chaining is caused by noisy points between two clusters. Examples of good and bad cases for single-linkage clustering are illustrated in Figures 9.11(a) and (b), respectively. Therefore, the behavior of single-linkage methods depends on the impact and relative presence of noisy data points.

The complete (worst-case) linkage method attempts to minimize the maximum distance between any pair of points in a cluster. This quantification can implicitly be viewed as an approximation of the diameter of a cluster. Because of its focus on minimizing the diameter, it will try to create clusters so that all of them have a similar diameter. However, if some of the natural clusters in the data are larger than others, then the approach will break up the larger clusters. It will also be biased towards creating clusters of spherical shape irrespective of the underlying data distribution. Another problem with the complete linkage method is that it gives too much importance to data points at the noisy fringes of a cluster because of its focus on the maximum distance between any pair of points in the cluster. The group-average, variance, and Ward's methods are more robust to noise due to the use of multiple linkages in the distance computation.

The agglomerative method requires the maintenance of a heap of sorted distances to efficiently determine the minimum distance value in the matrix. The initial distance matrix



(a) Good case with no noise (b) Bad case with noise

Figure 9.11: Good and bad cases for single-linkage clustering

computation requires $O(n^2 \cdot d)$ time, and the maintenance of a sorted heap data structure requires $O(n^2 \cdot \log(n))$ time over the course of the algorithm because there will be a total of $O(n^2)$ additions and deletions into the heap. Therefore, the overall running time is $O(n^2 \cdot d + n^2 \cdot \log(n))$. The required space for the distance matrix is $O(n^2)$. The space-requirement is particularly problematic for large data sets. In such cases, a similarity matrix M cannot be incrementally maintained, and the time complexity of many hierarchical methods will increase dramatically to $O(n^3 \cdot d)$. This increase occurs because the similarity computations between clusters need to be performed explicitly at the time of the merging.

Agglomerative hierarchical methods naturally lead to a binary tree of clusters. It is generally difficult to control the structure of the hierarchical tree with bottom-up methods as compared to top-down methods. Therefore, in cases where a taxonomy of a specific structure is desired, bottom-up methods are less desirable.

A problem with hierarchical methods is that they are sensitive to a small number of mistakes made during the merging process. For example, if an incorrect merging decision is made at some stage because of the presence of noise in the data set, then there is no way to undo it, and the mistake may further propagate in successive merges. In fact, some variants of hierarchical clustering, such as single-linkage methods, are notorious for successively merging neighboring clusters because of the presence of a small number of noisy points. Nevertheless, there are numerous ways to reduce these effects by treating noisy data points specially.

Agglomerative methods can become impractical from a *space- and time-efficiency* perspective for larger data sets. Therefore, these methods are often combined with sampling and other partitioning methods to efficiently provide solutions of high quality.

9.3.3 Top-down Divisive Methods

Although bottom-up agglomerative methods are typically distance-based methods, top-down hierarchical methods can be viewed as general-purpose meta-algorithms that can use almost any clustering algorithm as a subroutine. Because of the top-down approach, greater control is achieved on the global structure of the tree in terms of its degree and balance between different branches.

The overall approach for top-down clustering uses a general-purpose flat clustering algorithm $\mathcal{A}$ as a subroutine. The algorithm initializes the tree at the root-node containing

Algorithm *GenericTopDownClustering*(Data: $\mathcal{D}$, Flat Algorithm: $\mathcal{A}$)
begin
  Initialize tree $\mathcal{T}$ to root containing $\mathcal{D}$;
  repeat
    Select a leaf node L in $\mathcal{T}$ based on pre-defined criterion;
    Use algorithm $\mathcal{A}$ to split L into $L_1 \ldots L_k$;
    Add $L_1 \ldots L_k$ as children of L in $\mathcal{T}$;
  until termination criterion;
end

Figure 9.12: Generic top-down meta-algorithm for clustering

all the data points. In each iteration, the data set at a particular node of the current tree is split into multiple nodes (clusters). By changing the criterion for node selection, one can create trees balanced by height or trees balanced by the number of clusters. If the algorithm $\mathcal{A}$ is randomized, such as the k-means algorithm (with random seeds), it is possible to use multiple trials of the same algorithm at a particular node and select the best one. The generic pseudocode for a top-down divisive strategy is illustrated in Figure 9.12. The algorithm recursively splits nodes with a top-down approach until either a certain height of the tree is achieved or each node contains fewer than a predefined number of data objects. A wide variety of algorithms can be designed with different instantiations of the algorithm $\mathcal{A}$ and growth strategy. Note that the algorithm $\mathcal{A}$ can be any arbitrary clustering algorithm, and not just a distance-based algorithm.

9.3.3.1 Bisecting k-Means

The bisecting k-means algorithm is a top-down hierarchical clustering algorithm in which each node is split into exactly two children with a 2-means algorithm. To split a node into two children, several randomized trial runs of the split are used, and the split that has the best impact on the overall clustering objective is used. Several variants of this approach use different growth strategies for selecting the node to be split. For example, the heaviest node may be split first, or the node with the smallest distance from the root may be split first. These different choices lead to balancing either the cluster weights or the tree height.

9.3.4 Probabilistic Model-based Algorithms

Most clustering algorithms discussed in this book are *hard* clustering algorithms in which each data point is deterministically assigned to a particular cluster. Probabilistic model-based algorithms are *soft* algorithms in which each data point may have a non-zero assignment probability to many (typically all) clusters. A soft solution to a clustering problem may be converted to a hard solution by assigning a data point to a cluster with respect to which it has the largest assignment probability.

The broad principle of a mixture-based *generative* model is to assume that the data was generated from a mixture of k distributions with probability distributions $\mathcal{G}_1 \ldots \mathcal{G}_k$. Each distribution $\mathcal{G}_i$ represents a cluster and is also referred to as a *mixture component*. Each data point $\overline{X_i}$, where $i \in \{1 \ldots n\}$, is generated by this mixture model as follows:

1. Select a mixture component with prior probability $\alpha_i = P(\mathcal{G}_i)$, where $i \in \{1 \ldots k\}$. Assume that the rth one is selected.

2. Generate a data point from $\mathcal{G}_r$.

This generative model will be denoted by $\mathcal{M}$. The different prior probabilities α_i and the parameters of the different distributions $\mathcal{G}_r$ are not known in advance. Each distribution $\mathcal{G}_i$ is often assumed to be the Gaussian, although any arbitrary (and different) family of distributions may be assumed for each $\mathcal{G}_i$. The choice of distribution $\mathcal{G}_i$ is important because it reflects the user's *a priori* understanding about the distribution and shape of the individual clusters (mixture components). The parameters of the distribution of each mixture component, such as its mean and variance, need to be estimated from the data, so that the overall data has the maximum likelihood of being *generated* by the model. This is achieved with the *expectation-maximization (EM)* algorithm. The parameters of the different mixture components can be used to describe the clusters. For example, the estimation of the mean of each Gaussian component is analogous to determining the mean of each cluster center in a k-representatives algorithm. After the parameters of the mixture components have been estimated, the *posterior* generative (or assignment) probabilities of data points with respect to each mixture component (cluster) can be determined.

Assume that the probability density function of mixture component $\mathcal{G}_i$ is denoted by $f^i(\cdot)$. The probability (density function) of the data point $\overline{X}_j$ being generated by the model is given by the weighted sum of the probability densities over different mixture components, where the weight is the prior probability $\alpha_i = P(\mathcal{G}_i)$ of the mixture components:

$$f^{point}(\overline{X}_j|\mathcal{M}) = \sum_{i=1}^{k} \alpha_i \cdot f^i(\overline{X}_j) \tag{9.28}$$

Then, for a data set $\mathcal{D}$ containing n data points, denoted by $\overline{X_1} \ldots \overline{X_n}$, the probability density of the data set being generated by the model $\mathcal{M}$ is the product of all the point-specific probability densities:

$$f^{data}(\mathcal{D}|\mathcal{M}) = \prod_{j=1}^{n} f^{point}(\overline{X}_j|\mathcal{M}) \tag{9.29}$$

The log-likelihood fit $\mathcal{L}(\mathcal{D}|\mathcal{M})$ of the data set $\mathcal{D}$ with respect to model $\mathcal{M}$ is the logarithm of the aforementioned expression and can be (more conveniently) represented as a sum of values over the different data points. The log-likelihood fit is preferred for computational reasons.

$$\mathcal{L}(\mathcal{D}|\mathcal{M}) = \log(\prod_{j=1}^{n} f^{point}(\overline{X}_j|\mathcal{M})) = \sum_{j=1}^{n} \log(\sum_{i=1}^{k} \alpha_i f^i(\overline{X}_j)) \tag{9.30}$$

This log-likelihood fit needs to maximized to determine the model parameters. A salient observation is that if the probabilities of data points being generated from different clusters were known, then it becomes relatively easy to determine the optimal model parameters separately for each component of the mixture. At the same time, the probabilities of data points being generated from different components are dependent on these optimal model parameters. This circularity is reminiscent of a similar circularity in optimizing the objective function of partitioning algorithms in Section 9.3.1. In that case, the knowledge of a *hard* assignment of data points to clusters provides the ability to determine optimal cluster representatives locally for each cluster. In this case, the knowledge of a *soft* assignment provides the ability to estimate the optimal (maximum likelihood) model parameters *locally*

for each cluster. This naturally suggests an iterative EM algorithm, in which the model parameters and probabilistic assignments are iteratively estimated from one another.

Let Θ be a vector, representing the *entire set* of parameters describing all components of the mixture model. For example, in the case of the Gaussian mixture model, Θ contains all the component mixture means, variances, co-variances, and the *prior* generative probabilities $\alpha_1 \ldots \alpha_k$. Then, the EM algorithm starts with an initial set of values of Θ (possibly corresponding to random assignments of data points to mixture components), and proceeds as follows:

1. (E-step) Given the current value of the parameters in Θ, estimate the *posterior* probability $P(\mathcal{G}_i|\overline{X}_j, \Theta)$ of the component $\mathcal{G}_i$ having been selected in the generative process, given that we have observed data point $\overline{X}_j$. The quantity $P(\mathcal{G}_i|\overline{X}_j, \Theta)$ is also the soft cluster assignment probability that we are trying to estimate. This step is executed for each data point $\overline{X}_j$ and mixture component $\mathcal{G}_i$.

2. (M-step) Given the current probabilities of assignments of data points to clusters, use the maximum likelihood approach to determine the values of all the parameters in Θ that maximize the log-likelihood fit on the basis of current assignments.

The two steps are executed repeatedly in order to improve the maximum likelihood criterion. The algorithm is said to converge when the objective function does not improve significantly in a certain number of iterations. The details of the E-step and the M-step will now be explained.

The E-step uses the currently available model parameters to compute the probability density of the data point $\overline{X}_j$ being generated by each component of the mixture. This probability density is used to compute the Bayes probability that the data point $\overline{X}_j$ was generated by component $\mathcal{G}_i$ (with model parameters fixed to the current set of the parameters Θ):

$$P(\mathcal{G}_i|\overline{X}_j, \Theta) = \frac{P(\mathcal{G}_i) \cdot P(\overline{X}_j|\mathcal{G}_i, \Theta)}{\sum_{r=1}^{k} P(\mathcal{G}_r) \cdot P(\overline{X}_j|\mathcal{G}_r, \Theta)} = \frac{\alpha_i \cdot f^{i,\Theta}(\overline{X}_j)}{\sum_{r=1}^{k} \alpha_r \cdot f^{r,\Theta}(\overline{X}_j)} \tag{9.31}$$

A superscript Θ has been added to the probability density functions to denote the fact that they are evaluated for current model parameters Θ.

The M-step requires the optimization of the parameters for each probability distribution under the assumption that the E-step has provided the "correct" soft assignment. To optimize the fit, the partial derivative of the log-likelihood fit with respect to corresponding model parameters needs to be computed and set to zero. Without specifically describing the details of these algebraic steps, the values of the model parameters that are computed as a result of the optimization are described here.

The value of each α_i is estimated as the current weighted fraction of points assigned to cluster i, where a weight of $P(\mathcal{G}_i|\overline{X}_j, \Theta)$ is associated with data point $\overline{X}_j$. Therefore, we have:

$$\alpha_i = P(\mathcal{G}_i) = \frac{\sum_{j=1}^{n} P(\mathcal{G}_i|\overline{X}_j, \Theta)}{n} \tag{9.32}$$

In practice, in order to obtain more robust results for smaller data sets, the expected number of data points belonging to each cluster in the numerator is augmented by 1, and the total number of points in the denominator is $n + k$. Therefore, the estimated value is as follows:

$$\alpha_i = \frac{1 + \sum_{j=1}^{n} P(\mathcal{G}_i | \overline{X}_j, \Theta)}{k + n} \tag{9.33}$$

This approach is also referred to as *Laplacian smoothing*.

To determine the other parameters for component i, the value of $P(\mathcal{G}_i | \overline{X}_j, \Theta)$ is treated as a weight of that data point. Consider a Gaussian mixture model in d dimensions, in which the distribution of the ith component is defined as follows:

$$f^{i,\Theta}(\overline{X}_j) = \frac{1}{\sqrt{|\Sigma_i|}(2 \cdot \pi)^{(d/2)}} e^{-\frac{1}{2}(\overline{X}_j - \overline{\mu_i})\Sigma_i^{-1}(\overline{X}_j - \overline{\mu_i})} \tag{9.34}$$

Here, $\overline{\mu_i}$ is the d-dimensional mean vector of the ith Gaussian component, and Σ_i is the $d \times d$ covariance matrix of the generalized Gaussian distribution of the ith component. The notation $|\Sigma_i|$ denotes the determinant of the covariance matrix. It can be shown[5] that the maximum-likelihood estimation of $\overline{\mu_i}$ and Σ_i yields the (probabilistically weighted) means and covariance matrix of the data points in that component. These probabilistic weights were derived from the assignment probabilities in the E-step. Interestingly, this is exactly how the representatives and covariance matrices of the Mahalanobis k-means approach are derived in Section 9.3.1. The only difference was that the data points were not weighted because hard assignments were used by the deterministic k-means algorithm. Note that the term in the exponent of the Gaussian distribution is the square of the Mahalanobis distance.

The E-step and the M-step can be iteratively executed to convergence to determine the optimal parameter set Θ. At the end of the process, a probabilistic model is obtained that describes the entire data set in terms of a generative model. The model also provides soft assignment probabilities $P(\mathcal{G}_i | \overline{X}_j, \Theta)$ of the data points, on the basis of the final execution of the E-step.

In practice, to minimize the number of estimated parameters, the non-diagonal entries of Σ_i are often set to 0. In such cases, the determinant of Σ_i simplifies to the product of the variances along the individual dimensions. This is equivalent to using the square of the *Minkowski* distance in the exponent. If all diagonal entries are further constrained to have the same value, then it is equivalent to using the Euclidean distance, and all components of the mixture will have spherical clusters. Thus, different choices and complexities of mixture model distributions provide different levels of flexibility in representing the probability distribution of each component.

This two-phase iterative approach is similar to representative-based algorithms. The E-step can be viewed as a soft version of the *assign* step in distance-based partitioning algorithms. The M-step is reminiscent of the *optimize* step, in which optimal component-specific parameters are learned on the basis of the fixed assignment. The distance term in the exponent of the probability distribution provides the natural connection between probabilistic and distance-based algorithms.

The E-step is structurally similar to the *Assign* step, and the M-step is similar to the *Optimize* step in k-representative algorithms. Many mixture component distributions can be expressed in the form $K_1 \cdot e^{-K_2 \cdot Dist(\overline{X_i}, \overline{Y_j})}$, where K_1 and K_2 are regulated by distribution parameters. The log-likelihood of such an exponentiated distribution directly maps to an addi-

[5]This is achieved by setting the partial derivative of $\mathcal{L}(\mathcal{D}|\mathcal{M})$ (see Equation 9.30) with respect to each parameter in $\overline{\mu_i}$ and Σ to 0.

tive distance term $Dist(\overline{X_i}, \overline{Y_j})$ in the M-step objective function, which is structurally identical to the corresponding additive optimization term in k-representative methods. For many EM models with mixture probability distributions of the form $K_1 \cdot e^{-K_2 \cdot Dist(\overline{X_i}, \overline{Y_j})}$, a corresponding k-representative algorithm can be defined with a distance function $Dist(\overline{X_i}, \overline{Y_j})$.

9.3.5 Kohonen Self-Organizing Map

The Kohonen self-organizing map [103] constructs a 1-dimensional or 2-dimensional embedding in which a 1-dimensional string-like or 2-dimensional lattice-like structure is imposed on the neurons. The dimensionality of the embedding is the same as the dimensionality of the lattice. It is also possible to create 3-dimensional embeddings with the appropriate choice of lattice structure. We will consider the case in which a 2-dimensional lattice-like structure is imposed on the neurons. As we will see, this type of lattice structure enables the mapping of all points to 2-dimensional space for visualization. An example of a 2-dimensional lattice structure of 25 neurons arranged in a 5×5 rectangular grid is shown in Figure 9.13(a). A hexagonal lattice containing the same number of neurons is shown in Figure 9.13(b). The shape of the lattice affects the shape of the 2-dimensional regions in which the clusters will be mapped.

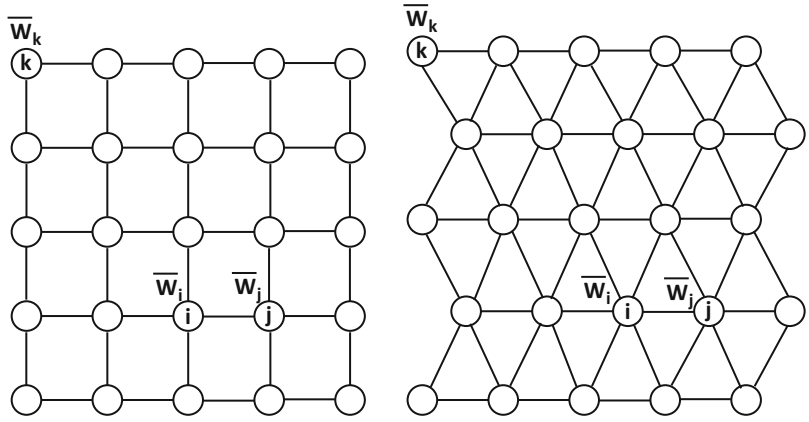
The idea of using the lattice structure is that the values of $\overline{W}_i$ in adjacent lattice neurons tend to be similar. Here, it is important to define separate notations to distinguish between the distance $||\overline{W}_i - \overline{W}_j||$ and the distance on the lattice. The distance between adjacent pairs of neurons on the lattice is exactly one unit. For example, the distance between the neurons i and j based on the lattice structure in Figure 9.13(a) is 1 unit, and the distance between neurons i and k is $\sqrt{2^2 + 3^2} = \sqrt{13}$. The vector-distance in the original input space (e.g., $||\overline{X} - \overline{W}_i||$ or $||\overline{W}_i - \overline{W}_j||$) is denoted by a notation like $Dist(\overline{W}_i, \overline{W}_j)$. On the other hand, the distance between neurons i and j along the lattice structure is denoted by $LDist(i, j)$. Note that the value of $LDist(i, j)$ is dependent only on the indices (i, j), and is independent of the values of the vectors $\overline{W}_i$ and $\overline{W}_j$.

The learning process in the self-organizing map is regulated in such a way that the closeness of neurons i and j (based on lattice distance) will also bias their weight vectors to be more similar. In other words, *the lattice structure of the self-organizing maps acts as a regularizer in the learning process*. As we will see later, imposing this type of 2-dimensional structure on the learned weights is helpful for visualizing the original data points with a 2-dimensional embedding.

The overall self-organizing map training algorithm proceeds in a similar way to competitive learning by sampling $\overline{X}$ from the training data, and finding the winner neuron based on the Euclidean distance. The weights in the winner neuron are updated in a manner similar to the vanilla competitive learning algorithm. However, the main difference is that a damped version of this update is also applied to the lattice-neighbors of the winner neuron. In fact, in soft variations of this method, one can apply this update to all neurons, and the level of damping depends on the lattice distance of that neuron to the winning neuron. The damping function, which always lies in $[0, 1]$, is typically defined by a Gaussian kernel:

$$Damp(i, j) = \exp\left(-\frac{LDist(i, j)^2}{2\sigma^2}\right) \tag{9.35}$$

Here, σ is the bandwidth of the Gaussian kernel. Using extremely small values of σ reverts to pure winner-take-all learning, whereas using larger values of σ leads to greater regularization in which lattice-adjacent units have more similar weights. For small values of σ, the damping



(a) Rectangular (b) Hexagonal

Figure 9.13: An example of a 5×5 lattice structure for the self-organizing map. Since neurons i and j are close in the lattice, the learning process will bias the values of $\overline{W}_i$ and $\overline{W}_j$ to be more similar. The rectangular lattice will lead to rectangular clustered regions in the resulting 2-dimensional representation, whereas the hexagonal lattice will lead to hexagonal clustered regions in the resulting 2-dimensional representation

function will be 1 only for the winner neuron, and it will be 0 for all other neurons. Therefore, the value of σ is one of the parameters available to the user for tuning. Note that many other kernel functions are possible for controlling the regularization and damping. For example, instead of the smooth Gaussian damping function, one can use a thresholded step kernel, which takes on a value of 1 when $LDist(i, j) < \sigma$, and 0, otherwise.

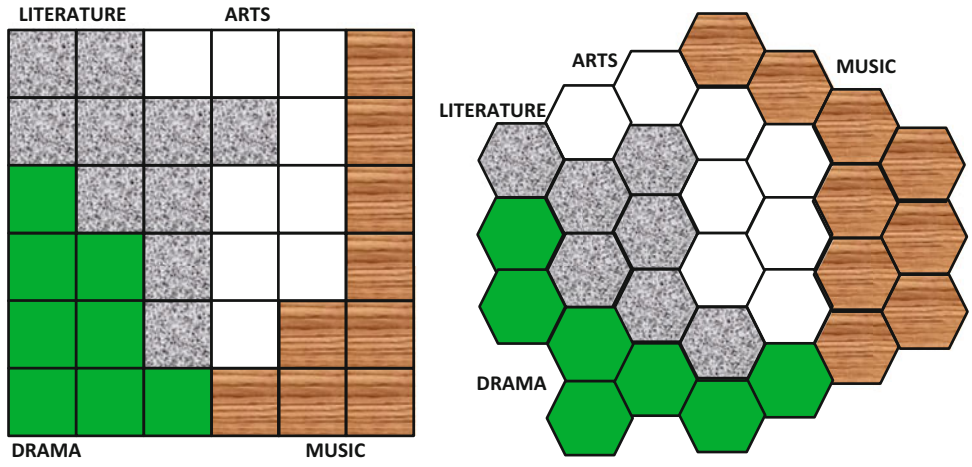
The training algorithm repeatedly samples $\overline{X}$ from the training data, and computes the distances of $\overline{X}$ to each weight $\overline{W}_i$. The index p of the winning neuron is computed. Rather than applying the update only to $\overline{W}_p$ (as in winner-take-all), the following update is applied to each $\overline{W}_i$:

$$\overline{W}_i \Leftarrow \overline{W}_i + \alpha \cdot Damp(i, p) \cdot (\overline{X} - \overline{W}_i) \quad \forall i \tag{9.36}$$

Here, $\alpha > 0$ is the learning rate. It is common to allow the learning rate α to reduce with time. These iterations are continued until convergence is reached. Note that weights that are lattice-adjacent will receive similar updates, and will therefore tend to become more similar over time. *Therefore, the training process forces lattice-adjacent clusters to have similar points, which is useful for visualization.*

Using the Learned Map for 2D Embeddings

The self-organizing map can be used in order to induce a 2-dimensional embedding of the points. For a $k \times k$ grid, all 2-dimensional lattice coordinates will be located in a square in the positive quadrant with vertices $(0, 0)$, $(0, k-1)$, $(k-1, 0)$, and $(k-1, k-1)$. Note that each grid point in the lattice is a vertex with integer coordinates. The simplest 2-dimensional embedding is simply by representing each point $\overline{X}$ with its closest grid point (i.e., winner neuron). However, such an approach will lead to overlapping representations of points. Furthermore, a 2-dimensional representation of the data can be constructed and each coordinate is one of $k \times k$ values from $\{0 \ldots k-1\} \times \{0 \ldots k-1\}$. This is the reason that the self-organizing map is also referred to as a *discretized* dimensionality reduction method. It is



(a) Rectangular lattice (b) Hexagonal lattice

Figure 9.14: Examples of 2-dimensional visualization of documents belonging to four topics

possible to use various heuristics to disambiguate these overlapping points. When applied to high-dimensional document data, a visual inspection often shows documents of a particular topic being mapped to a particular local regions. Furthermore, documents of related topics (e.g., politics and elections) tend to get mapped to adjacent regions. Illustrative examples of how a self-organizing map arranges documents of four topics with rectangular and hexagonal lattices are shown in Figures 9.14(a) and (b), respectively. The regions are colored differently, depending on the majority topic of the documents belonging to the corresponding region.

Self-organizing maps have a strong neurobiological basis in terms of their relationship with how the mammalian brain is structured. In the mammalian brain, various types of sensory inputs (e.g., touch) are mapped onto a number of folded planes of cells, which are referred to as *sheets* [59]. When parts of the body that are close together receive an input (e.g., tactile input), then groups of cells that are physically close together in the brain will also fire together. Therefore, proximity in (sensory) inputs is mapped to proximity in neurons, as in the case of the self-organizing map. This type of neurobiological inspiration is used in many neural architectures, such as *convolutional neural networks* for image data [107, 110].

9.3.6 Spectral Clustering

Spectral clustering [129, 164] combines nonlinear dimensionality reduction with k-means clustering, which allows one to learn clusters of arbitrary shapes. Spectral clustering combines row and column similarities by first using row-wise similarities to create a matrix and then using dimensionality reduction to create a new representation that incorporates column-wise similarities as well. Spectral clustering uses the following steps:

1. **(Breaking inter-cluster links):** Let $S = [s_{ij}]$ be a symmetric $n \times n$ similarity matrix defined over n data points, in which s_{ij} is the similarity between data points i and j. It is not necessary for the data points to be multidimensional. The similarity matrix might be created with the use of a domain-specific similarity function such as a *string subsequence kernel* [118] in the case of sequence or text data. The diagonal

entries of S are set to 0. All pairs (i, j) are identified such that data points i and j are *mutual* κ-nearest neighbors of each other according to the similarity matrix S. Such similarity values, s_{ij}, are retained in S. Otherwise, the value of s_{ij} is set to 0. This step sparsifies the similarity matrix, and intuitively tries to "break" the inter-cluster links, so that the resulting points are less likely to be close to one another in the engineered representation. The number of nearest neighbors, κ, regulates the sparsity of the similarity matrix.

2. **(Normalizing for dense and sparse regions):** For each row i, the sum of each row in the symmetric matrix S is computed as follows:

$$S_i = \sum_j s_{ij}$$

Intuitively, the value of S_i quantifies the "density" in the locality of data point i. Then, each similarity value is normalized using the following relation:

$$s_{ij} \Leftarrow \frac{s_{ij}}{\sqrt{S_i \cdot S_j}} = \frac{s_{ij}}{\text{GEOMETRIC-MEAN}(S_i, S_j)}$$

The basic idea is to normalize the similarities between data points with the geometric mean of the "densities" at their end points. Therefore, the similarity is *relative* to the *local* data distribution. For example, the similarity between two modestly similar data points in a local region belonging to a sparsely populated cluster becomes magnified, whereas the similarity between two data points in a dense region is de-emphasized. This type of adjustment makes the similarity function more adaptive to the statistics of data locality. For example, if a data point is in a very dense region, it facilitates the creation of a larger number of fine-grained clusters in that region. At the same time, it becomes possible to create fewer clusters with more widely separated points in sparse regions. An intuitive way of understanding this (in the context of a spatial application) is that population clusters in sparsely-populated Alaska would be geographically larger than those in densely-populated California.

3. **(Explicit feature engineering):** The resulting similarity matrix S is diagonalized to $S = Q\Delta Q^T$, where the columns of Q contain the eigenvectors, and Δ is a diagonal matrix containing the eigenvalues. Only the largest $r \ll n$ eigenvectors (columns) of Q need to be computed to create a smaller $n \times r$ matrix Q_0. Furthermore, each row of Q_0 is scaled to unit norm, so that all engineered points (i.e., rows of Q_0) lie on the unit sphere. At this point, the k-means algorithm is applied on the normalized and engineered points with the Euclidean distance.

The first two steps change the similarity matrix in a data-dependent way because aggregated statistics from multiple points are used to change the entries. The various adjustments to the engineered representation such as the dropping of lower-order eigenvectors help in improving the representation of the data for clustering. Spectral clustering is able to discover nonlinearly shaped, entangled clusters from the data that are not possible to discover using methods like k-means, which discover only spherically shaped clusters. For example, spectral clustering will be able to discover both clusters in Figure 9.11, since the portions of the similarity matrix corresponding to two clusters will have non-zero entries. Furthermore, the bridge of points shown in Figure 9.11(b) will often be disconnected in the sparsification step with the proper choice of similarity function.

9.4 Why Unsupervised Learning Is Important

Most of human and animal learning is unsupervised learning, which serves as a base for other forms of learning. For example, humans learn the nature of their environment all the time, as they take in sensory inputs and file away useful bits of information all the time. This information is then used for learning more specialized tasks. For example, it is much easier for a person to learn the laws of physics after having experienced how the world works via observation. A similar observation holds true in machine learning, wherein unsupervised learning often makes it easier to perform specialized tasks like classification. Unsupervised methods are used often for feature engineering, pretraining, and semisupervised learning. In the following, we will provide several examples of unsupervised models for supervised learning.

9.4.1 Feature Engineering for Machine Learning

The class of methods, referred to as *kernel methods* use unsupervised feature engineering in order to perform classification. It is noteworthy that feature engineering is also used to change the behavior of unsupervised methods like clustering, when it is desired for the method to show particular types of characteristics. A specific example of feature engineering for clustering is the use of spectral methods, where the diagonalization of a similarity matrix provides a new set of features on which k-means clustering is performed. By changing the nature of the similarity matrix, one can change the behavior of the underlying algorithm.

In kernel methods for classification, a similarity matrix is constructed from a data set with the use of a similarity function that is more sensitive to distances than the dot product. Recall from the discussion earlier in this chapter that if we have an $n \times d$ data set, then the eigenvectors of the $n \times n$ similarity matrix DD^T yield a rotated version of the data set which the same as the representation provided by SVD. The (i, j)th entry of $S = DD^T$ is the dot product similarity between the ith and jth rows of D. In other words, if $S = [s_{ij}]$ is the similarity matrix, and $\overline{X}_i$ is the ith row of D, then we have the following:

$$s_{ij} = \overline{X}_i \cdot \overline{X}_j$$

Singular value decomposition creates the embedding $U = Q\Sigma$ for diagonal matrix Σ, so that the following is satisfied:

$$S = DD^T = Q\Sigma^2 Q^T = \underbrace{(Q\Sigma)}_{U}(Q\Sigma)^T = UU^T$$

Here, Q, Σ, and U are all $n \times n$ matrices. An important property of U in vanilla SVD is that at most d columns are nonzero, and therefore the new embedding is also (at most) d-dimensional. This is not particularly surprising, since SVD simply rotates the data set.

Kernel methods use a nonlinear version of singular value decomposition in which the matrix S is constructed with a more sensitive similarity function (e.g., the Gaussian kernel) rather than with the use of dot products:

$$s_{ij} \propto \exp(-\|\overline{X}_i - \overline{X}_j\|^2/\sigma^2) \tag{9.37}$$

The use of a more sensitive similarity function results in a new embedding $S = UU^T$, in which it is possible for all n columns of U to be nonzero. Therefore, the new embedding might be of much higher dimensionality than vanilla SVD. The larger dimensionality of the data set makes it easier to separate the classes in the new representation with a linear

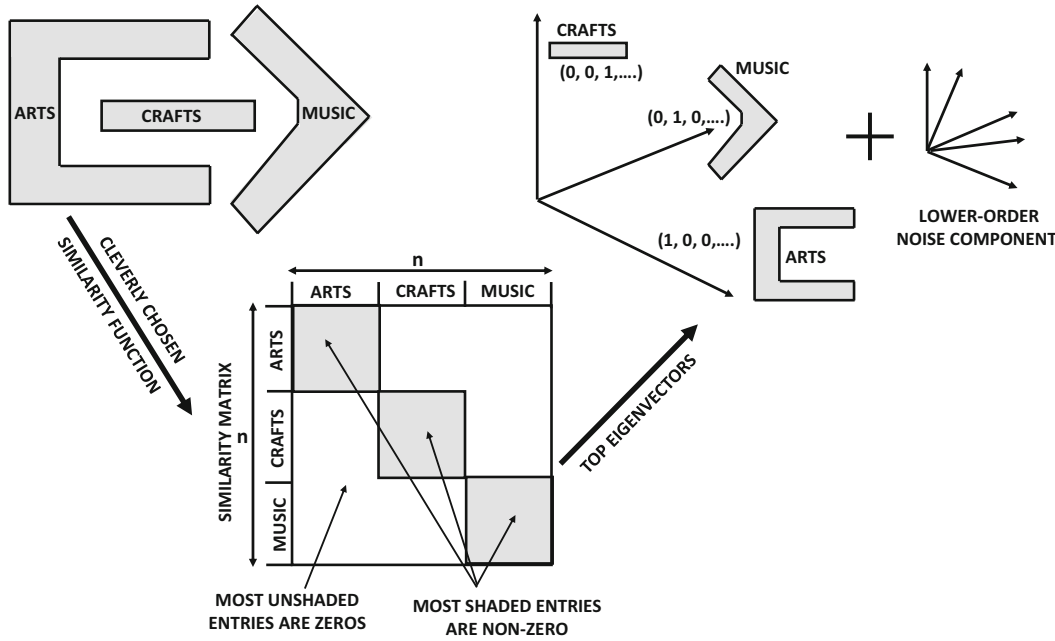


Figure 9.15: Explaining the rationale for nonlinear dimensionality reduction

hyperplane. As a result, one often transforms the data to this new space, before performing classification. In order to understand why nonlinearly separable clusters become linearly separable with the use of a more distance sensitive similarity function, we will use an example.

Consider a setting in which the data matrix D is an $n \times d$ matrix containing the frequencies of d words in each of n documents. The data set contains three classes of related topics corresponding to *Arts*, *Crafts*, and *Music*, and we wish to build a classifier separating these classes. The classes are naturally clustered in the underlying data, as shown in Figure 9.15. Unfortunately, the classes are not linearly separable, and a linear classifier will have difficulty in separating any particular class from the other classes.

Now imagine that we could somehow define a similarity matrix in which most of the similarities between documents of different topics are close to zero, whereas most of the similarities between documents of the same topic are nearly 1s. This can be achieved with the Gaussian kernel of Equation 9.37, provided that the bandwidth σ is chosen appropriately. The resulting similarity matrix S is shown in Figure 9.15 with a natural block structure. What type of embedding U will yield the factorization $S = UU^T$? First let us consider the absolutely perfect similarity function in which the entries in all the shaded blocks are 1s and all the entries outside shaded blocks are 0s. In such a case, it can be shown (after ignoring zero eigenvalues) that every document in *Arts* will receive an embedding of $(1, 0, 0)$, every document in *Music* will receive an embedding of $(0, 1, 0)$, and every document in *Crafts* will receive an embedding of $(0, 0, 1)$. Of course, in practice, we will never have a precise block structure of 1s and 0s, and there will be significant noise/finer trends within the block structure. These variations will be captured by the lower-order eigenvectors shown in Figure 9.15. Even with these additional noise dimensions, this new representation will typically be linearly separable for classification. The key idea here is that dot product similarities are

sometimes not very good at capturing the *detailed* structure of the data, which other similarity functions with sharper locality-centric variations can sometimes capture. The purpose of using the Gaussian kernel is to precisely accentuate these locality-centric variations appropriately. The only supervision here is in choosing the bandwidth of the kernel based on out-of-sample performance. Note that it is not necessary to use all the columns of U; rather one can drop the smaller eigenvectors as noisy dimensions.

Armed with this basic idea, we provide an example of a kernel classification algorithm, based on an $n \times d$ data matrix D that contains *both* the training and the test rows:

Diagonalize $S = Q\Sigma^2 Q^T$;
Extract the n-dimensional embedding in rows of $U = Q\Sigma$;
Partition rows of U into U_{train} and U_{test};
Apply linear SVM on training rows of U_{train} and class labels to learn model $\mathcal{M}$;
Apply $\mathcal{M}$ on each row of U_{test} to yield predictions;

This model is almost the same as the kernel SVM, with the only difference being that a kernel SVM performs the feature engineering with only the training rows, and then fits the test rows into the training space. Generalizing nonlinear tranformations to out-of-sample rows in this way is referred to as the *Nystrom method*. The precise feature engineering analog to the kernel SVM is described in [8]. Furthermore, it is more common to use this type of feature engineering indirectly with the use of the *kernel trick* in traditional machine learning, rather than via feature engineering. Nevertheless, these kernel methods are roughly equivalent to the procedure described above.

9.4.2 Radial Basis Function Networks for Feature Engineering

A traditional feed-forward network contains many layers, and the nonlinearity is typically created by the repeated composition of activation functions. On the other hand, an RBF network typically uses only an input layer, a single hidden layer (with a special type of behavior defined by RBF functions), and an output layer. As in feed-forward networks, the input layer is not really a computational layer, and it only carries the inputs forward. The layers of the RBF network are designed as follows:

1. The input layer simply transmits from the input features to the hidden layers. Therefore, the number of input units is exactly equal to the dimensionality d of the data. As in the case of feed-forward networks, no computation is performed in the input layers. As in all feed-forward networks, the input units are fully connected to the hidden units and carry their input forward.

2. The computations in the hidden layers are based on comparisons with *prototype vectors*. Each hidden unit contains a d-dimensional prototype vector. Let the prototype vector of the ith hidden unit be denoted by $\overline{\mu}_i$. In addition, the ith hidden unit contains a bandwidth denoted by σ_i. Although the prototype vectors are always specific to particular units, the bandwidths of different units σ_i are often set to the same value σ. The prototype vectors and bandwidth(s) are usually learned either in an unsupervised way, or with the use of mild supervision.

   Then, for any input training point $\overline{X}$, the activation $\Phi_i(\overline{X})$ of the ith hidden unit is defined as follows:

$$h_i = \Phi_i(\overline{X}) = \exp\left(-\frac{||\overline{X} - \overline{\mu}_i||^2}{2 \cdot \sigma_i^2}\right) \quad \forall i \in \{1, \ldots, m\} \tag{9.38}$$

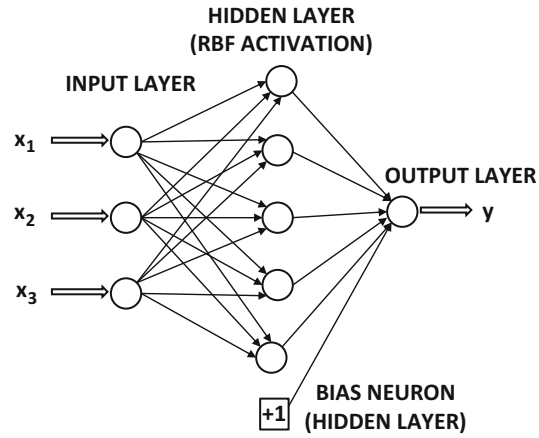


Figure 9.16: An RBF network: Note that the hidden layer is broader than the input layer, which is typical (but not mandatory)

The total number of hidden units is denoted by m. Each of these m units is designed to have a high level of influence on the particular cluster of points that is closest to its prototype vector. Therefore, one can view m as the number of clusters used for modeling, and it represents an important hyper-parameter available to the algorithm. For low-dimensional inputs, it is typical for the value of m to be larger than the input dimensionality d, but smaller than the number of training points n.

3. For any particular training point $\overline{X}$, let h_i be the output of the ith hidden unit, as defined by Equation 9.38. The weights of the connections from the hidden to the output nodes are set to w_i. Then, the prediction $\hat{y}$ of the RBF network in the output layer is defined as follows:

$$\hat{y} = \sum_{i=1}^{m} w_i h_i = \sum_{i=1}^{m} w_i \Phi_i(\overline{X}) = \sum_{i=1}^{m} w_i \exp\left(-\frac{||\overline{X} - \overline{\mu}_i||^2}{2 \cdot \sigma_i^2}\right)$$

The variable $\hat{y}$ has a circumflex on top to indicate the fact that it is a predicted value rather than observed value. If the observed target is real-valued, then one can set up a least-squares loss function, which is much like that in a feed-forward network. The values of the weights $w_1 \ldots w_m$ need to be learned in a supervised way.

An example of an RBF network is illustrated in Figure 9.16.

In the RBF network, there are two sets of computations corresponding to the hidden layer and the output layer. The parameters $\overline{\mu}_i$ of the hidden layer are learned in an unsupervised way, whereas those of the output layer are learned in a supervised way with gradient descent. The latter is similar to the case of the feed-forward network. The prototypes $\overline{\mu}_i$ may either be sampled from the data, or be set to be the m *centroids* of an m-way clustering algorithm. In other words, we can partition the training data into m clusters with an off-the-shelf clustering algorithm, and use the means of the m clusters as the m prototypes. The parameters σ_i are set to the same value of σ, which is often treated as a hyper-parameter. In other words, it is tuned on a portion of the data hat is held out in order to optimize classification accuracy.

An interesting special case is when the prototypes are set to the individual training points (and therefore the value m is the same as the number of training examples). In such cases, RBF networks can be shown to specialize to well-known *kernel methods* in machine learning. However, since RBF networks can choose different prototypes than the training points, it suggests that RBF networks have greater power and flexibility than do kernel methods.

9.4.3 Semisupervised Learning

The process by which most humans and animals learn is most closely reflected in semi-supervised learning. Not all experiences of humans are task focused; rather there is a level of background knowledge that one learns from day-to-day experiences. This background knowledge is often deployed in task-focused learning. The day-to-day experiences (which are not task-focused) can be viewed as unsupervised experiences. These experiences do, however, help in learning more focused tasks in most cases. An important observation is that humans train in an unsupervised way most of the time, whereas task-focused learning is only performed a small part of the time. A similar observation applies to machine learning tasks where one can improve the accuracy of supervised algorithms by using unsupervised data. In most cases, unsupervised data is copious, whereas the supervised data is very limited. The goal is to restrict the model to small parts of the feature space. This is particularly useful when a lot of unlabeled data is available, and only a small part of the data is labeled.

Many generic meta-algorithms, such as self-training, co-training, and pre-training, are often used for learning. The goal of generic meta-algorithms is to use existing classification algorithms to enhance the classification process with unlabeled data. The simplest method is *self-training*, in which the smoothness assumption is used to incrementally expand the labeled portions of the training data. The major drawback of this approach is that it might lead to overfitting. One way of avoiding overfitting is by using *co-training*. Co-training partitions the feature space and independently labels instances using classifiers trained on each of these feature spaces. The labeled instances from one classifier are used as feedback to the other, and vice versa.

9.4.3.1 Self-Training

The self-training procedure can use any existing classification algorithm $\mathcal{A}$ as input. The classifier $\mathcal{A}$ is used to incrementally assign labels to unlabeled examples for which it has the most confident prediction. As input, the self-training procedure uses the initial labeled set L, the unlabeled set U, and a user-defined parameter k that may sometimes be set to 1. The self-training procedure iteratively uses the following steps:

1. Use algorithm $\mathcal{A}$ on the current labeled set L to identify the k instances in the unlabeled data U for which the classifier $\mathcal{A}$ is the most confident.

2. Assign labels to the k most confidently predicted instances and add them to L. Remove these instances from U.

The major drawback of self-training is that the addition of predicted labels to the training data can lead to propagation of errors in the presence of noise. Another procedure, known as *co-training*, is able to avoid such overfitting more effectively.

9.4.3.2 Co-Training

In co-training, it is assumed that the feature set can be partitioned into two *disjoint* groups F_1 and F_2, such that each of them is sufficient to learn the target classification function. It is important select the two feature subsets so that they are as independent from one another as possible. Two classifiers are constructed, such that one classifier is constructed on each of these groups. These classifiers are not allowed to interact with one another directly for prediction of unlabeled examples though they are used to build up training sets for each other. This is the reason that the approach is referred to as *co-training*.

Let L be the labeled training data and U be the unlabeled data. Let L_1 and L_2 be the labeled sets for each of these classifiers. The sets L_1 and L_2 are initialized to the available labeled data L, except that they are represented in terms of disjoint feature sets F_1 and F_2, respectively. Over the course of the co-training process, as different examples from the initially unlabeled set U are added to L_1 and L_2, respectively, the training instances in L_1 and L_2 may vary from one another. Two classifier models $\mathcal{A}_1$ and $\mathcal{A}_2$ are constructed using the training sets L_1 and L_2, respectively. The following steps are then iteratively applied:

1. Train classifier $\mathcal{A}_1$ using labeled set L_1, and add k most confidently predicted instances from unlabeled set $U - L_2$ to training data set L_2 for classifier $\mathcal{A}_2$.

2. Train classifier $\mathcal{A}_2$ using labeled set L_2, and add k most confidently predicted instances from unlabeled set $U - L_1$ to training data set L_1 for classifier $\mathcal{A}_1$.

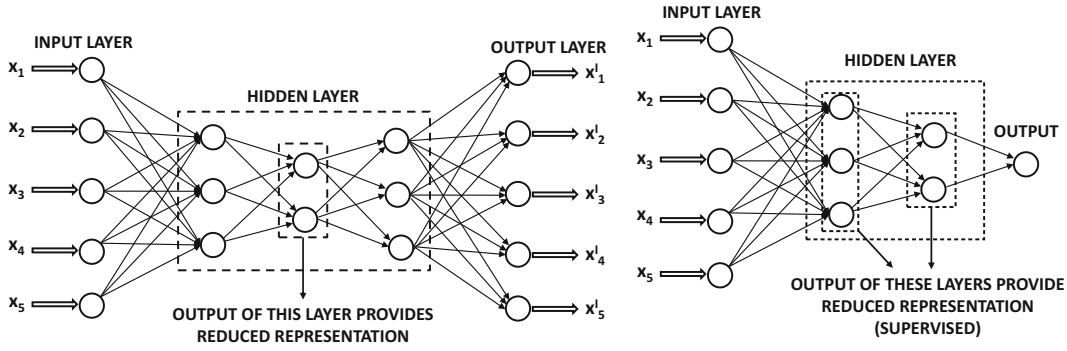
In many implementations of the method, the most confidently labeled examples *for each class* are added to the training sets of the other classifier. This procedure is repeated until all instances are labeled. The two classifiers are then retrained with the expanded training data sets. This approach can be used to label not only the unlabeled data set U, but also unseen test instances. At the end of the procedure, two classifiers are returned. For an unseen test instance, each classifier may be used to determine the class label scores. The score for a test instance is determined by combining the scores of the two classifiers. For example, if the Bayes method is used as the base classifier, then the product of the posterior probabilities returned by the two classifiers may be used.

The co-training approach is more robust to noise because of the disjoint feature sets used by the two algorithms. An important assumption is that of *conditional independence* of the features in the two sets with respect to a particular class. In other words, after the class label is fixed, the features in one subset are conditionally independent of the other. The intuition for this is that instances generated by one classifier appear to be randomly distributed to the other, and vice versa. As a result, the approach will generally be more robust to noise than the self-training method.

9.4.3.3 Unsupervised Pretraining in Multilayer Neural Networks

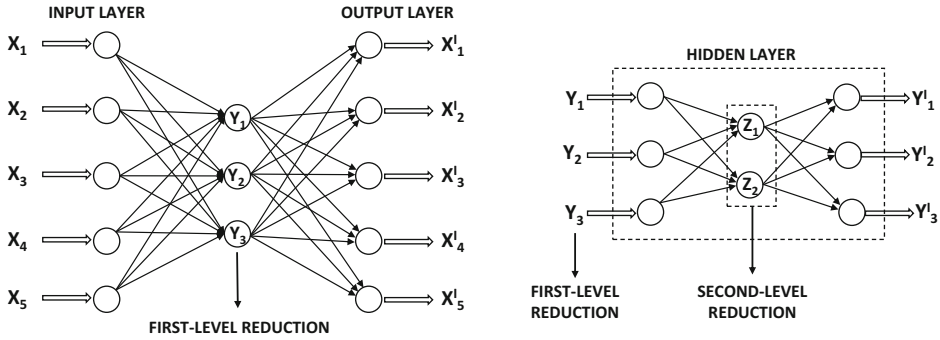
A neural network with multiple layers is referred to as a *deep network*. The early layers learn features that are used by later layers. Deep networks are inherently hard to train because the magnitudes of the gradients in various layers are often quite different. As a result, the different layers of the neural network do not get trained at the same rate. The multiple layers of the neural network cause distortions in the gradient, which make them hard to train.

Although the depth of the neural network causes challenges, the problems associated with depth are also heavily dependent on how the network is initialized. A good initialization point can often solve many of the problems associated with reaching good solutions. A



(a) Multilayer autoencoder (b) Multilayer classifier

Figure 9.17: Both the multilayer classifier and the multilayer autoencoder use a similar pretraining procedure



(a) Pretraining first-level reduction and outer weights (b) Pretraining second-level reduction and inner weights

Figure 9.18: Pretraining a neural network

ground-breaking break-through in this context was the use of unsupervised pretraining in order to provide robust initializations [80]. This initialization is achieved by training the network greedily in layer-wise fashion. The approach was originally proposed in the context of deep belief networks, but it was later extended to other types of models such as autoencoders [146, 189]. In this chapter, we will study the autoencoder approach because of its simplicity. First, we will start with the dimensionality reduction application, because the application is unsupervised and it is easy to show how to use unsupervised pretraining in this case. However, unsupervised pretraining can also be used for supervised applications like classification with minor modifications.

In pretraining, a greedy approach is used to train the network one layer at a time by learning the weights of the outer hidden layers first and then learning the weights of the inner hidden layers. The resulting weights are used as starting points for a final phase of traditional neural network backpropagation in order to fine-tune them.

Consider the autoencoder and classifier architectures shown in Figure 9.17. Since these architectures have multiple layers, randomized initialization can sometimes cause challenges. However, it is possible to create a good initialization by setting the initial weights layer by

layer in a greedy fashion. First, we describe the process in the context of the autoencoder shown in Figure 9.17(a), although an almost identical procedure is relevant to the classifier of Figure 9.17(b). We have intentionally chosen neural architectures in the two cases so that the hidden layers have similar numbers of nodes.

The pretraining process is shown in Figure 9.18. The basic idea is to assume that the two (symmetric) outer hidden layers contain a first-level reduced representation of larger dimensionality, and the inner hidden layer contains a second-level reduced representation of smaller dimensionality. Therefore, the first step is to learn the first-level reduced representation and the corresponding weights associated with the outer hidden layers using the simplified network of Figure 9.18(a). In this network, the middle hidden layer is missing and the two outer hidden layers are collapsed into a single hidden layer. The assumption is that the two outer hidden layers are related to one another in a symmetric way like a smaller autoencoder. In the second step, the reduced representation in the first step is used to learn the second-level reduced representation (and weights) of the inner hidden layers. Therefore, the inner portion of the neural network is treated as a smaller autoencoder in its own right. Since each of these pretrained subnetworks is much smaller, the weights can be learned more easily. This initial set of weights is then used to train the entire neural network with backpropagation. Note that this process can be performed in layerwise fashion for a deep neural network containing any number of hidden layers.

So far, we have only discussed how we can use unsupervised pretraining for unsupervised applications. A natural question arises as to how one can use pretraining for supervised applications. Consider a multilayer classification architecture with a single output layer and k hidden layers. During the pretraining stage, the output layer is removed, and the representation of the final hidden layer is learned in an unsupervised way. This is achieved by creating an autoencoder with $2 \cdot k - 1$ hidden layers, where the middle layer is the final hidden layer of the supervised setting. For example, the relevant autoencoder for Figure 9.17(b) is shown in Figure 9.17(a). Therefore, an additional $(k-1)$ hidden layers are added, each of which has a symmetric counterpart in the original network. This network is trained in exactly the same layer-wise fashion as discussed above for the autoencoder architecture. The weights of only the encoder portion of this autoencoder are used for initialization of the weights entering into all hidden layers. The weights between the final hidden layer and the output layer can also be initialized by treating the final hidden layer and output nodes as a single-layer network. This single-layer network is fed with the reduced representations of the final hidden layer (based on the autoencoder learned in pretraining). After the weights of all the layers have been learned, the output nodes are re-attached to the final hidden layer. The backpropagation algorithm is applied to this initialized network in order to fine-tune the weights from the pretrained stage. Note that this approach learns all the initial hidden representations in an unsupervised way, and only the weights entering into the output layer are initialized using the labels. Therefore, the pretraining can still be considered to be largely unsupervised.

Unsupervised pretraining helps even in cases where the amount of training data is very large. It is likely that this behavior is caused by the fact that pretraining helps in issues beyond model generalization. One evidence of this fact is that in larger data sets, even the error on the training data seems to be high, when methods like pretraining are not used. In these cases, the weights of the early layers often do not change much from their initializations, and one is using only a small number of later layers on a random transformation of the data (defined by the random initialization of the early layers). As a result, the trained portion of the network is rather shallow, with some additional loss caused by the random transformation. In such cases, pretraining also helps a model realize the full benefits of depth, thereby facilitating the improvement of prediction accuracy on larger data sets.

Another way of understanding pretraining is that it provides insights into the repeated patterns in the data, which are the features learned from digits by putting together these frequent shapes. However, these shapes also have discriminative power with respect to recognizing digits. Expressing the data in terms of a few features then helps in recognizing how these features are related to the class labels. This is at the heart of the idea of unsupervised learning, which uses copiously available labeled data to recognize the frequent patterns. This principle is summarized by Geoff Hinton [79] in the context of image classification as follows: "*To recognize shapes, first learn to generate images.*" This type of regularization preconditions the training process in a semantically relevant region of the parameter space, where several important features have already been learned, and further training can fine-tune and combine them for prediction.

9.5 Summary

Unsupervised learning methods use a variety of techniques to develop compressed representations from data. This compressed representation could take the form of a data set with reduced dimensionality, or it could take the form of the representatives of the various clusters in the data set. Linear dimensionality reduction is a form of matrix factorization, in which a data matrix can be represented as a product of two matrices. Some forms of linear dimensionality reduction (such as nonnegative matrix factorization) are closely related to clustering.

Unsupervised methods are used for various types of feature engineering algorithms. The core idea of feature engineering is to create a new representation of the data on which existing supervised algorithms work effectively. Examples include kernel methods and radial basis function networks. One can also use unsupervised methods in order to improve the accuracy of supervised methods. Unsupervised methods are able to learn the manifolds on which the data points lie. The knowledge of the structure of the manifold reduces the amount of labeled data that is then required for classification.

9.6 Further Reading

Methods for dimensionality reduction and matrix factorization are discussed in detail in [8]. A detailed book on data clustering may be found in [10]. A discussion of the use of autoencoders for dimensionality reduction may be found in [6]. Discussions of feature engineering and semisupervised learning may be found in [8, 10]. The Kohonen self-organizing map is discussed in detail in [103]. Detailed discussions of pretraining methods for supervised and unsupervised learning may be found in [6].

9.7 Exercises

1. Use singular value decomposition to show the *push-through identity* for any $n \times d$ matrix D:

$$(\lambda I_d + D^T D)^{-1} D^T = D^T (\lambda I_n + D D^T)^{-1}$$

2. Let D be an $n \times d$ data matrix, and $\overline{y}$ be an n-dimensional column vector containing the dependent variables of linear regression. The regularized solution to linear regression predicts the dependent variables of a test instance $\overline{Z}$ using the following equation:

$$\text{Prediction}(\overline{Z}) = \overline{Z}\,\overline{W} = \overline{Z}(D^T D + \lambda I)^{-1} D^T \overline{y}$$

Here, the vectors $\overline{Z}$ and $\overline{W}$ are treated as $1 \times d$ and $d \times 1$ matrices, respectively. Show using the result of Exercise 1, how you can write the above prediction purely in terms of similarities between training points or between $\overline{Z}$ and training points.

3. Suppose that you are given a truncated SVD $D \approx Q\Sigma P^T$ of rank-k. Show how you can use this solution to derive an alternative rank-k decomposition $Q'\Sigma' P'^T$ in which the unit columns of Q (or/and P) might not be mutually orthogonal and the truncation error is the same.

4. **Recommender systems:** Let D be an $n \times d$ matrix in which only a small subset of the entries are specified. This is commonly the case with recommender systems. Show how you can adapt the algorithm for unconstrained matrix factorization to this case, so that only observed entries are used to create the factors. How would you change the matrix-based updates of Equation 9.6 to this case.

5. **Biased matrix factorization:** Consider the factorization of an incomplete $n \times d$ matrix D into an $n \times k$ matrix U and a $d \times k$ matrix V:

$$D \approx UV^T$$

Suppose you add the constraint that all entries of the penultimate column of U and the final column of V are fixed to 1. Discuss the similarity of this model to that of the addition of bias to classification models. How is gradient descent modified?

6. The text of the book discusses gradient descent updates (cf. Equation 9.6) for unconstrained matrix factorization $D \approx UV^T$. Suppose that the matrix D is symmetric, and we want to perform the symmetric matrix factorization $D \approx UU^T$. Formulate the objective function and gradient descent steps of symmetric matrix factorization in a manner similar to the asymmetric case.

7. Discuss why the following integer matrix factorization is equivalent to the objective function of the k-means algorithm for an $n \times d$ matrix D, in which the rows contain the data points:

$$\begin{aligned} \text{Minimize}_{U,V} \; & \|D - UV^T\|_F^2 \\ & \text{subject to:} \\ & \text{Columns of } U \text{ are mutually orthogonal} \\ & u_{ij} \in \{0, 1\} \end{aligned}$$

8. What is the maximum number of possible clusterings of a data set of n points into k groups? What does this imply about the convergence behavior of algorithms whose objective function is guaranteed not to worsen from one iteration to the next?

9. Suppose that you represent your data set as a graph in which each data point is a node, and the weight of the edge between a pair of nodes is equal to the Gaussian kernel similarity between them. Edges with weight less than a particular threshold are dropped. Interpret the single-linkage clustering algorithm in terms of this similarity graph.

10. The text of the chapter shows how one can transform any linear classifier into recognizing nonlinear decision boundaries by using a feature engineering phase in which the eigenvectors of an appropriately chosen similarity matrix are used to create new features. Discuss the impact of this type of preprocessing on the nature of the clusters found by the k-means algorithm.