Statistical NLP: course notes

Çağrı Çöltekin — SfS / University of Tübingen

2020-05-27

These notes are prepared for the class *Statistical Natural Language Processing* taught in Seminar für Sprachwissenschaft, University of Tübingen.

8 Unsupervised machine learning

Unsupervised learning refers to a set of machine learning methods that do not require labeled examples. Unlike in supervised learning, there is no correct or incorrect solution. The motivation to is to find some structure in the data. As a result, the unsupervised methods are often used for exploratory methods for discovering structure in the given data. Unsupervised methods may also used as a substitute for their supervised counterparts when no labeled data is available, or to supplement supervised methods in case labeled data is small and unlabeled data is abundant. Considering the cost of creating labeled (annotated) data, unsupervised methods are attractive. However, unsurprisingly, they are rarely as accurate as their supervised counterparts.

Since we do not have labels, unsupervised learning by itself is an ill-defined problem. We do not have a 'ground truth', which makes measuring the success difficult. The question is, then, what drives learning in an unsupervised method? The short answer is the similarity or differences of the objects of interest. As a result the way we define similarity or differences between the objects is important, and it is the way we can affect the 'learning' of an unsupervised method.

Traditional unsupervised methods include, *clustering*, *dimensionality reduction*, and *density estimation*. Clustering aims to group the data into sensible clusters. Dimensionality reduction aims at reducing the redundancy in the data such that a small number of informative dimensions are retained. Density estimation assumes that the data at hand is sampled from a number of groups whose members are distributed according to some probability distributions. The aim is, then, to characterize the underlying distributions, with which we can determine the likelihood of each data point coming from one of the distributions estimated from the data.

Another way of looking at the unsupervised methods is as probabilistic models with latent (hidden or unobserved) variables. In fact, all of the classical methods introduced in this chapter can be viewed as different instantiations of probabilistic models. For the remainder of this chapter, we will go through the classical methods listed above.

8.1 Clustering

Clustering aims to solve a problem similar to *classification*. Both methods assign the objects of interest into a number of groups. However, in case of clustering, we do not know the true group labels. What we hope to do is to discover is 'natural groupings' of the objects in our data set.

Figure 8.1 shows an example data set for clustering. Note that unlike in classification, the data points do not have labels. Humans are often good at clustering two (and maybe three) dimensional data similar to the one presented in Figure 8.1. For the data in Figure 8.1, most people are likely to suggest a clustering similar to the one presented in Figure 8.2. However, most real world problems require dealing with (very) high-dimensional feature spaces, and it is difficult for humans to visualize and cluster data expressed in feature spaces larger than 3 dimensions.

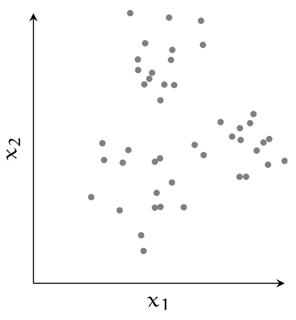


Figure 8.1: An example data set in $\mathbb{R}^2$ for clustering. Note that unlike classification data (e.g., in Figure **??**), the data points do not have associated labels.

Example uses of clustering in CL include clustering languages or dialects for determining their relationships; clustering texts to discover authorship or topics; clustering words in such a way that semantically similar words fall into the same cluster.

In this chapter, we will discuss a few common methods used for clustering. Before introducing the methods, however, we first discuss formalizing similarity or distances between two objects and some other related concepts which are crucial for any of the methods we will discuss below.

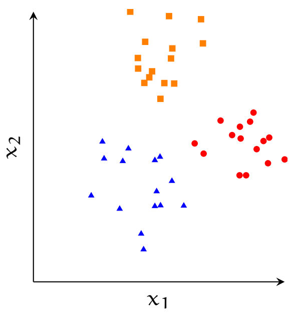


Figure 8.2: A likely clustering of the data in Figure 8.1, into three clusters. In fact, the data is sampled from three bivariate normal distributions with different means, and the labeling indicates the original clusters (distributions).

8.1.1 *Similarity and distance*

Since we do not have real labels for each item, we rely on a distance (or similarity) measure in all clustering methods. If the objects of interest are expressed as feature vectors in $\mathbb{R}^n$, we can use one of the well-known distance measures. Typical choices include the *Euclidean distance* and *Manhattan distance*. Euclidean distance between two vectors is the L2 norm of their difference.

$$\|\mathbf{a}-\mathbf{b}\| = \sqrt{\sum_{i=1}^{k}(a_i-b_i)^2}$$

where, $\mathbf{a}$ and $\mathbf{b}$ are d-dimensional vectors. The Manhattan (or taxicab) distance between two feature vectors is the L1 norm of their difference.

$$\|\mathbf{a}-\mathbf{b}\|_1 = \sum_{i=1}^{k}|a_i-b_i|$$

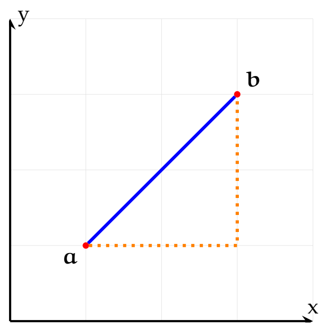


Figure 8.3: Visualization of Euclidean (solid blue line) and Manhattan (dotted orange path) between two vectors $\mathbf{a}$ and $\mathbf{b}$ in $\mathbb{R}^2$.

These are only two well-known distance measures defined on Euclidean (feature) space. Choice of distance measure (besides the choice of features) depends on the application. Different distance measures lead to different clusters.

Some clustering algorithms only operate on distances. For these algorithms, we do not even need the explicit features. Distance measures can also be defined on objects without explicit definition of real-valued feature vectors. In linguistic applications it is common to define distance metrics over stings (e.g., words) and trees (e.g., syntactic representations) directly.

Formally, a *distance metric* D has to satisfy the following three properties:

1. $D(a, b) = D(b, a)$ (symmetry)
2. $D(a, b) \geqslant 0$ for all a, b (non-negativity)
3. $D(a, b) + D(b, c) \geqslant D(a, c)$ (triangle inequality)

Once a distance measure is defined on the objects of interest, a natural approach to clustering is minimizing the *within-cluster scatter*, which is defined as

$$\sum_{k=1}^{K} \sum_{a \in C_k} \sum_{b \in C_k} D(a, b)$$

where K is the number of clusters and C_k represents the set of data points that belong to the cluster k. Within cluster scatter measures the total distance between all data points which share the cluster assignment. A related measure is the *between cluster scatter*, which is the sum of all distances between all data points that are not assigned to the same cluster. Formally, Between cluster scatter is defined as

$$\sum_{k=1}^{K} \sum_{a \in C_k} \sum_{b \notin C_k} D(a, b).$$

Between- and within-cluster scatter are visualized in Figure 8.4 using Euclidean distances on a 2-dimensional feature space ($\mathbb{R}^2$).

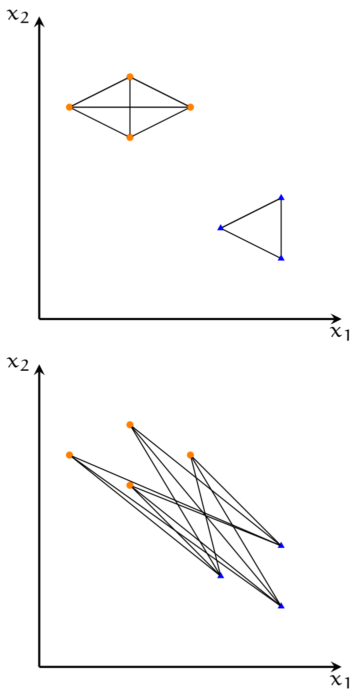


Figure 8.4: Two example clusters (one shown as blue triangles, the other with orange dots), and their within-cluster (above) and between cluster (below) distances.

Intuitively, we want the data points that belong to the same cluster to be closer to each other, while the clusters to be farther from each other. As a result, minimizing within-cluster scatter, and maximizing between-cluster scatter makes sense. You may have already noticed that minimizing one also means maximizing the other. They sum up to the total scatter in the data set, which is constant for a given data set. Hence, by minimizing within-cluster scatter, we automatically maximize the between-cluster scatter.

There are two problems with a direct approach to minimizing within-cluster scatter for obtaining the best clustering configuration for a given data set. First, the lowest within-cluster scatter is obtained when each data point forms its own cluster. As a result, without a predefined number K of clusters, the optimum solution (which puts each data point in its own cluster) is not useful.[1]

[1] We will get back to the choice of number of clusters later.

The second problem is computational complexity. Enumerating all possible clustering configurations, and finding the configuration with the lowest within-cluster scatter is intractable for most realistic data sets.

Given the high computational complexity, all practical clustering methods opt for an approximate solution. There are two major approaches to clustering. One approach is to divide the features space into areas for each cluster. The other is to organize the objects hierarchically such that items that are most similar to each other are grouped together, and repeating the process recursively to obtain a hierarchical grouping. We will discuss the *k-means* as an example of the non-hierarchical clustering method, and hierarchical clustering in the remainder of this section.

8.1.2 K-means

The k-means algorithm is a simple but effective clustering algorithm. The algorithm requires real-valued feature vectors in $\mathbb{R}^d$ as its input.[2] The number of clusters, K, is specified in advance. The algorithm finds the *centroids* (the mid-point of a set of data points in $\mathbb{R}^d$) of each proposed cluster. A new data point is assigned to the cluster with the closest centroid.

[2] Where, d is the dimension of the feature vectors.

The k-means algorithm is an iterative algorithm. An informal description of the algorithm is given below.[3]

[3] K-means algorithm is related to the *expectation-maximization algorithm* that we will discuss later in this lecture.

1. Randomly choose *centroids*, $m_1, \dots, m_K$, representing K clusters
2. Repeat until convergence
   (a) Assign each data point to the cluster of the nearest centroid
   (b) Re-calculate the centroid locations based on the assignments

Typically, the algorithm terminates (converges) when no cluster re-assignments are done in step 2a, or the centroids does not change in two successive iterations. Both criteria can be 'softened' by allowing a small threshold for the relevant difference to prevent long convergence times.

The k-means algorithm finds a *local minimum* of the sum of squared Euclidean distance between the cluster center and points assigned to the cluster within each cluster

$$\sum_{k=1}^{K} \sum_{a \in C_k} \|a - \mu_k\|^2 \tag{8.1}$$

Note the resemblance of the inner sum to formulation of the variance from Section 2.2.7. Effectively, we are looking for clusters with minimum variance. Furthermore, minimizing the distances of points in a cluster from their mean will also minimize the within-cluster scatter, distances between the data points within the same cluster. It should again be stressed that the k-means finds a local minimum, since the above objective is non-convex.

Figure 8.5 presents a demonstration of the k-means algorithm on the data set we have seen earlier in Figure 8.1 and 8.2. Note that the algorithm starts with a random initialization of cluster centroids, as shown in the upper left panel of Figure 8.5. The initialization affects the final outcome, different initializations may result in finding different cluster configurations. A way to reduce the effects of initialization is to run the algorithm multiple times with different initializations, and pick the solution with the lowest squared error. Rather than randomly initializing the centroids as in our example,

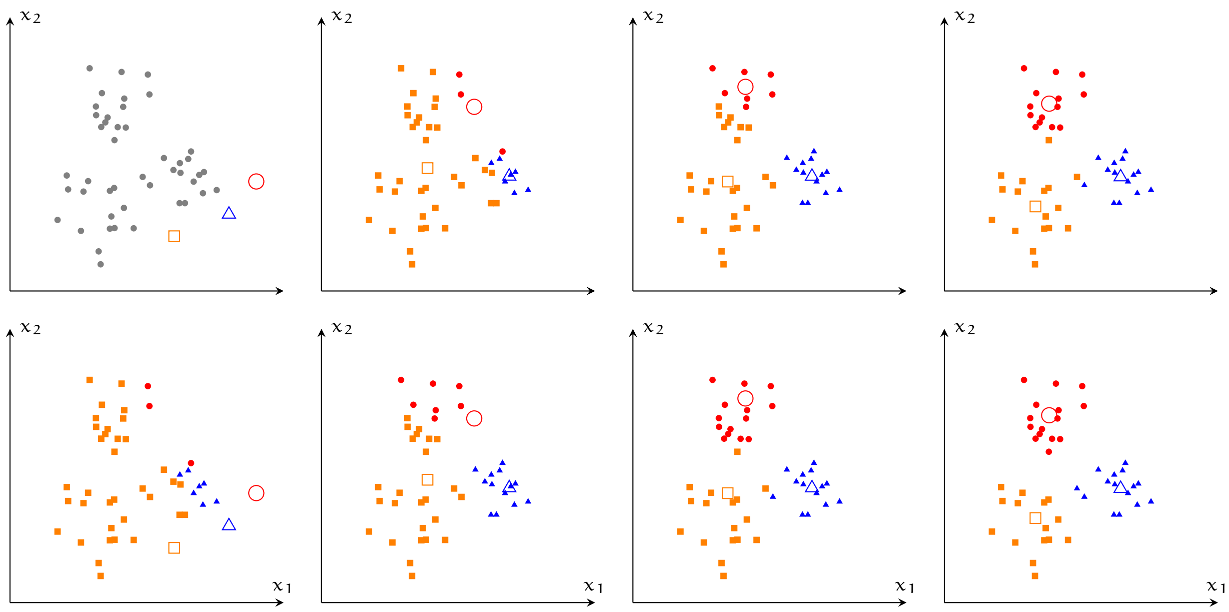


Figure 8.5: Demonstration of the k-means algorithm on the data presented in Figure 8.1. The solid marks on the graphs represents the data points, and large shapes represents the centroids of the clusters shown with the same shape and color. The first row corresponds to the step (a) in the algorithm, the second row corresponds to the step (b). Each column represents an iteration, where the first column shows the random initialization of the centroids, followed by the first cluster assignments.

there are also 'more informed' methods to assign initial cluster centroids. However, there is no single initialization method that works best in all applications.

As seen in Figure 8.5, by successive assignment of the points to the clusters, and re-computing the cluster centroids, the algorithm converges to the intuitive solution in Figure 8.1. The steps after the fourth iteration (not shown in the figure) does not change the cluster assignments. Note, however, the cluster labels do not match, with our 'gold-standard' labels in Figure 8.2 since the labels are arbitrarily assigned. A different run of the algorithm may produce the same solution with different label assignments. Although this is obvious and expected from an unsupervised method, it is worth to note that it affects the evaluation of the model.

One final note on the example in Figure 8.5 is that the type of data here is almost ideal for the k-means algorithm. K-means is known to perform well when clusters with approximately round-shaped (hyperspherical) and equal-sized clusters.[4]

[4] Why?

One issue with k-means is that we need to specify the number of clusters K, in advance. For some problems this is not much of an issue, as the number of clusters are fixed in advance (for example, clustering news articles into a fixed set of topics). However, in most cases the best K is not obvious. K-means objective (or within-cluster scatter) does not help us here since the objective is minimum when K is equal to the number of data points. A helpful method is to run the clustering multiple times with increasing K, and choose the point where the objective stops improving drastically. Figure 8.6 shows a plot, known as the *scree plot*, where the values of the objective is plotted against the number of clusters. The reduction of the error function improves significantly until $K = 3$, after which improvement is very small compared to earlier steps. Although it may not be as clear as in Figure 8.6 in real data sets, the point that the error line forms an 'elbow' in a scree plot is an indication of a good K value.

The k-means algorithm we describe in this section is used in many fields. It also has a large number of variations. One variation that is worth mentioning here is the *k-medoids* algorithm, which reduces the within-cluster distances between the data points and the cluster *medoid*, rather than cluster centroid. The cluster medoid, analogous to *median*, is always one of the data points, whose average distance to the other points in the cluster is minimal. One of the advantages of k-medoids is that it can use any distance function (unlike Euclidean distance used by k-means). However, it is computationally more expensive than k-means, and as a result, it is not as popular.

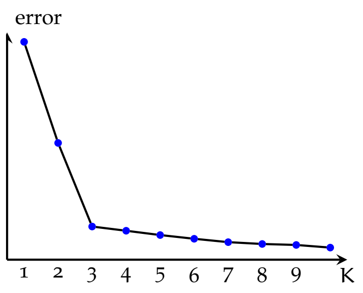


Figure 8.6: The scree plot for the data set in Figure 8.5. The graph shows squared error (Equation 8.1) after k-means algorithm has converged for K values between 1 and 10.

8.1.3 *Hierarchical clustering*

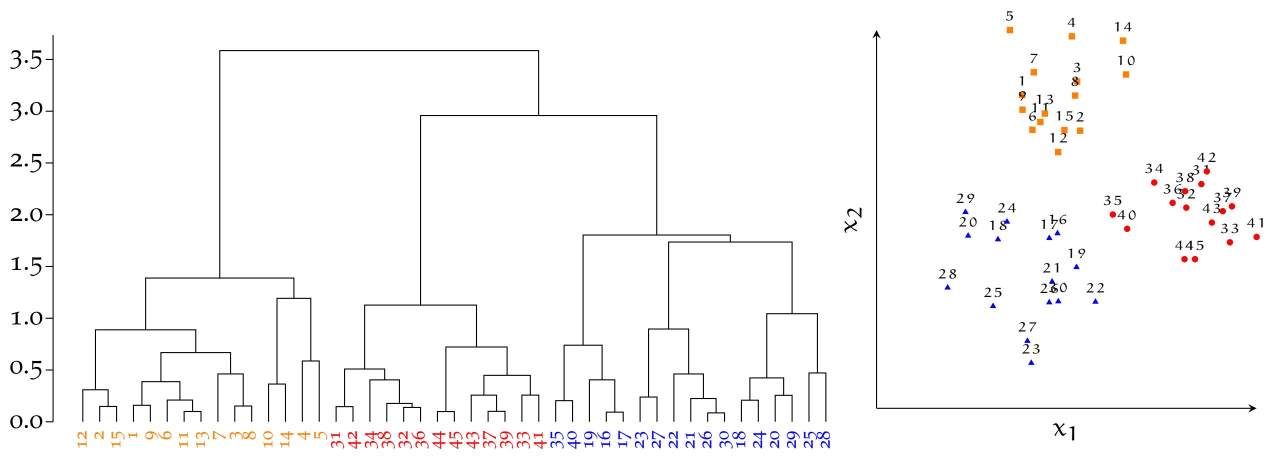


Figure 8.7: Example dendrogram (left) produced by hierarchical clustering of the points on the left pane. The data is the one we have used in demonstration of k-means (first introduced in Figure 8.2). We use the colors to represent 'true' clusters in the scatter plot on the right panel (we know the true clusters since data is artificially generated). The colors on the dendrogram reflect the 3-way clustering suggested by the dendrogram. We use numeric ids for cross-referencing the data in two-dimensional feature space and the dendrogram. The dendrogram is obtained by hierarchical agglomerative clustering using Euclidean distances, and complete linkage.

The k-means clustering discussed above forms groups in a 'flat' manner. Another approach to clustering is based on forming a hierarchical organization, such that the objects that are linked at a lower level and the objects (or clusters) with lower similarities are linked at higher levels of the hierarchy. The resulting hierarchy is most commonly shown as a tree called *dendrogram* as in the left panel of Figure 8.7. A dendrogram shows the cluster hierarchy, and the height at which two items or clusters merge shows the distance between them.

The clusters are determined after cutting 'cutting' the dendrogram at a particular height. For examples, cutting the dendrogram in Figure 8.7 at height 3.5 would yield two clusters, while cutting it at height 2.5 would yield a clustering close to the original three-clusters solution shown in the right panel of Figure 8.7. You are recommended to study this figure carefully.

The hierarchical clustering can be done bottom up, starting with each data point assigned to its own cluster, and merging the most-similar ones to obtain larger clusters at each step. This type of clustering is called *agglomerative* clustering. The other obvious method of hierarchical clustering, *divisive* clustering, starts with a single cluster containing all data points, and splits the clusters until each data point is in its own cluster. The optimum solution, considering all possible splits/merges, is intractable for both divisive and agglomerative clustering. As a result, the methods used in practice are greedy methods that often find a good solution, rather than the best solution. The methods known for agglomerative clustering are computationally more efficient and they are more popular.

A typical agglomerative clustering algorithm can be described as follows:

1. Compute the similarity/distance matrix
2. Assign each data point to its own cluster
3. Repeat until no clusters left to merge
   - Pick two clusters that are most similar to each other
   - Merge them into a single cluster

Figure 8.8 shows a step-by-step demonstration of agglomerative clustering of five points in two-dimensional Euclidean space. We start with assigning each point to its own cluster. Then we find the closest two points, which turns out to be the data points labeled 4 and 5, then we go on merging 2 and 3, then merge 1 with the cluster formed by 2 and 3, and finally merge remaining two clusters together. It is also important to note that most hierarchical clustering algorithms operate on a distance matrix, rather than the feature vectors. This is sometimes convenient when there is a well-known distance function that does not depend on explicit features, such as Levenshtein distance between two strings. Since a distance matrix is symmetric (more specifically positive semidefinite) with all 0 values at the main diagonal, the algorithm only uses part of it, typically a lower triangular matrix as in Figure 8.9.

We discussed various options for distance measures between two objects. However, note that determining the distance between a cluster and a data point (step 3 of Figure 8.8), and determining the distance between two clusters (step 4 of Figure 8.8) are non-trivial decisions.

There is a relatively large number of choices for measuring inter-cluster distances. The choice affects the computational complexity as well as the final clustering yielded by the clustering algorithm. The methods of measuring the distances between the clusters are known as *linkage* in the literature. The common linkage methods include the following.

Single linkage uses the minimal distance between two clusters as their distance. This method is related to the *minimum spanning tree* algorithm, and tends to form long linkage of 'thin' clusters. Successively linked objects are closer to each other. However, objects at the opposite ends of the link may be farther apart from each other compared to the objects in other clusters.

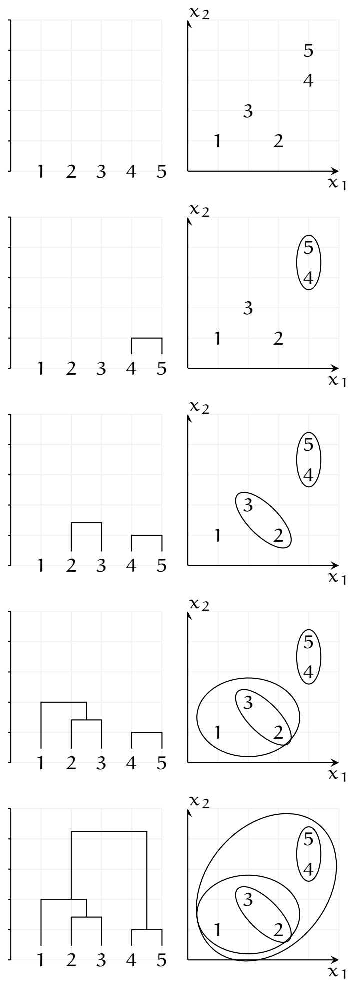


Figure 8.8: Step-by-step demonstration of agglomerative clustering. Each row shows a successive step in the procedure. The right side shows the dendrogram at each step, while the left side shows the data points in $\mathbb{R}^2$ alongside the clusters formed at each step.

Complete linkage uses the maximal distance between two clusters as their distance. Complete linkage typically finds clusters of similar size, avoiding the long chains of similarities of single linkage.

Average linkage uses the average of all inter-cluster distances.

Centroid method uses the distance between the cluster centroids.

Ward's method is another popular method for determining cluster differences. The general idea with the Ward's method is to choose the merge with the lowest within-cluster scatter (or variance). Remember that the clustering configuration with the lowest within-cluster scatter is one where each object is assigned its own cluster. Each merge during agglomerative clustering increases the within-cluster scatter. Then, we use the increase in the within-cluster variance incurred by merging two cluster as the distance between the clusters. This method, tends to yield more spherical equal-sized clusters (like k-means). Also note that although Ward's method often refers to an objective based on within-cluster variance, in practice one can use any other objective function that may improve clustering and/or that may be computationally convenient.

	1	2	3	4
1				
2	2.00			
3	1.41	1.41		
4	3.61	2.24	2.24	
5	4.24	3.16	2.83	1.00

Figure 8.9: The distance matrix used as input to the demonstration in Figure 8.8 (using Euclidean distances). We do not need to specify the upper triangular part as the matrix is symmetric, and the diagonal entries are all 0.

A schematic description of the basic linkage methods are demonstrated in Figure 8.10. There are more linkage options than the ones listed above. Most other linkage methods result in cluster configurations in-between single and complete linkage methods. In general, however, there is no single-best option. The choice of the linkage method is ultimately data and application dependent.

Note that single, complete and average linkage methods (as well as most variations of Ward's method) would work on distances, without need to access feature vectors in $\mathbb{R}^n$. This may be useful in tasks where some intuitive distance (or difference) can be defined between the objects, but the features are not accessible or definition of a centroid does not make sense. In computational/quantitative linguistics research, it is commonplace to use such distances. For example using Levenshtein distance between words. Another interesting property of these three linkage methods is that the distances in higher levels of cluster hierarchy never decrease. As a result the dendrograms obtained are proper dendrograms.

8.1.4 *Evaluating clustering results*

Evaluation is often problematic for unsupervised methods. Unlike their supervised counterparts, we do not typically have gold-standard labels (if we had them we would use classification). However, we often want to have an objective way for comparing clustering results. Although none of them are without problems, there are a number of major methods for obtaining evaluation metrics for clustering.

INTERNAL EVALUATION METRICS calculate a statistic, a measure, based only on the clustering result. Although there exists a number of different formulations, all of the internal evaluation metrics indicate some sort of clustering consistency. That is, we want cluster configurations where the distances within the clusters are low, and the distances between the clusters are high. If you realized the similarity of this statement with the similarity of the clustering objectives (e.g., of k-means) repeated multiple times above, you have also realized the problem. This type of evaluation will be biased towards the clustering methods that use similar objectives.

There are a number of measures that are the variations of the same idea above, including *Davies–Bouldin index*, *Dunn index*, *silhouette coefficient*, and *gap statistics*. These internal evaluation measures are also useful for selecting the ideal number of clusters. That is, we choose the k value that maximizes the evaluation metric of choice. Unlike within-cluster scatter, these metrics do not always increase (or decrease) as k is increased. Hence, to determine the best k value, we may apply the clustering algorithm multiple times with varying k, and pick the one with the best evaluation metric (e.g., gap statistic). However, maximizing/minimizing the statistic does not always result in the best clustering, and one often needs to resort visual aids like scree plots (Figure 8.6) for making the optimal choice.

For the sake of a concrete example, we will discuss the silhouette coefficient briefly here. The silhouette of a data point is defined as

$$s_i = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

where $a(i)$ is the average distance between the object i to the other objects in the same cluster, and $b(i)$ is the average distance between the object i and the objects in the closest cluster. The resulting index value ranges in the interval $[-1, 1]$. A negative value means the data point is closer to a neighboring cluster rather than its own cluster, zero indicates a data point on the border, while a positive value indicates a data point closer to its own cluster compared to others. The average silhouette value over all data points indicate the quality of clustering, where larger numbers indicate better clustering configurations. Although, the preferred clustering configurations depends on the distance metric (and the way the objects are represented), in general, this evaluation metric prefers compact clusters that are well-separated from the neighboring clusters. Sometimes silhouette score is used for selecting the optimum number of clusters. A configuration (number of clusters) that result in no (or least number of) negative silhouette scores is preferred over the other configurations. Also note that, like any other internal metric, we could use average s_i as the objective to maximize during clustering. The only reason for not having a wide-spread method based on maximizing the average silhouette score is that there is no known way to maximize it efficiently.

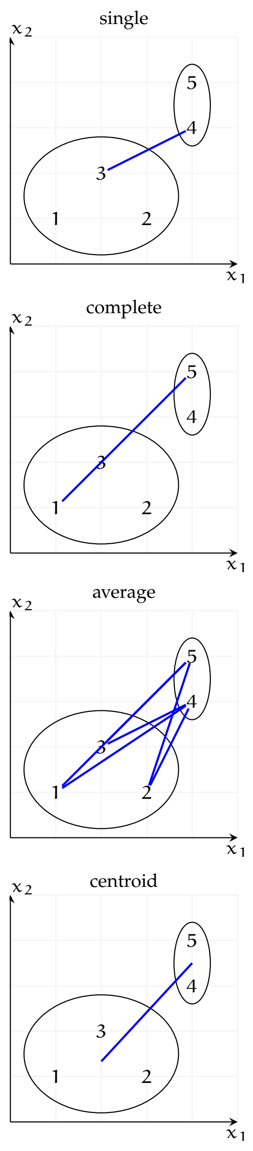


Figure 8.10: Single, complete, average and centroid linkage options.

EXTERNAL EVALUATION METHODS depend on a gold-standard test set. We may not have enough labeled data for training a classifier. However, if we have a (small) labeled test data, an external evaluation metric gives a direct comparison with the intended classes. The comparison of the labels, however, is not straightforward. Since the label assignments to clustering results are arbitrary, we cannot match them directly with gold-standard class labels.

Almost all external cluster evaluation metrics rely on a version of two properties.First, we want each cluster to be composed of members of a single gold-standard class. Second, we want all members of a gold-standard class to be assigned to a single cluster. The first property, *homogeneity*, has the trivial solution of assigning all data points to their own clusters. Similarly, the second property, *completeness*, can trivially be obtained by assigning all data points to a single cluster. Homogeneity is similar to precision, and completeness is similar to recall.[5] However, we do not need to have matching labels in the clustering solution to calculate the homogeneity and completeness scores. Also note that, similar to precision and recall, increasing one will likely to decrease the other.

Homogeneity and completeness can be quantified in a number of different ways. One popular way to define these quantities is based on entropy. Given a clustering solution K, and gold standard classes C, we can quantify homogeneity (h) and completeness (c) as

$$h = \begin{cases} 1 & \text{if } H(C,K) = 0 \\ 1 - \frac{H(C\mid K)}{H(C)} & \text{otherwise} \end{cases} \qquad c = \begin{cases} 1 & \text{if } H(C,K) = 0 \\ 1 - \frac{H(K\mid C)}{H(K)} & \text{otherwise} \end{cases}$$

Remember that conditional entropy, $H(x\mid y)$, is equal to the entropy of x if y does not provide any information about x. Otherwise, $H(x|y)$ is smaller than $H(x)$. Hence, both h and c range between $[0,1]$. High values of h can be obtained when we can predict the gold-standard classes from the clusters, which is possible when each cluster is dominated by the members of a single class. So, once we know the cluster, we can tell with high certainty which class it belongs to. High values of c are obtained when gold-standard classes can be mapped to clustering solution without loss. Having multiple clusters for a single class will reduce the c value.

If we treat the clustering solution and the gold standard classes as categorical distributions over the clustered objects, estimate the necessary probability values using MLE, the calculation of the (conditional) entropy values above are simple.

Now we can define a measure analogous to F-measure, which is called *V-measure* as,

$$\text{V-measure} = \frac{2 \times h \times c}{h + c}$$

There are quite a few other measures of external cluster evaluation, that we will not discuss here. Interested readers can find a few pointers to relevant literature at the end of the chapter.

EVALUATION BASED ON AN EXTERNAL TASK is another option. The clustering is often used as a means to improve a particular system. For example, if we are using clusters of words as features for a classifier, e.g., parser actions during parsing, the improvement in parsing,

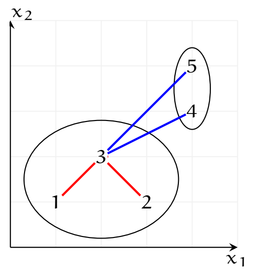


Figure 8.11: A schematic description of the silhouette metric. The metric tries to maximize the average distance of each data point (the data point labeled 3 in this example) to nearest cluster that it does not belong to (blue lines), and maximize the average distance of the data point to the others in the same cluster (red lines).

[5] Not surprisingly, the term completeness is used for the same quantity as recall in some disciplines.

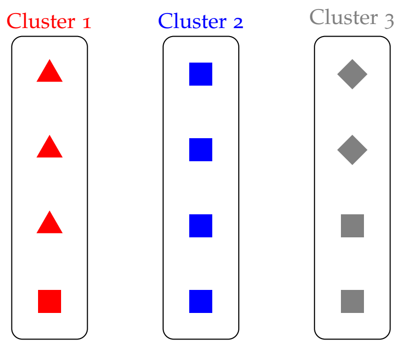


Figure 8.12: An example clustering solution with gold-standard labels (represented as shapes). Cluster 2 is pure, or fully homogeneous. The label represented by triangle is completely contained in cluster 1. The actual calculations of homogeneity, completeness and V-measure scores are left as an exercise.

then, can be used as the evaluation metric for the clustering results. Although this is not always applicable, if our aim is to improve a task using clustering, it is the naturally the best evaluation metric.

HUMAN JUDGMENTS, although subjective, are sometimes the only option, and can be useful even if one uses one of the more objective approaches outlined above. We should not dismiss the use of human evaluation. Aided with visualizations, human evaluation is often useful (or insightful, since getting insights about data is one of the reason for using unsupervised methods).

8.2 *Density estimation*

The clustering algorithms we discussed above assume that each data point belongs to only one cluster. For many problems this is appropriate and/or desirable. For others, it makes more sense to assign objects to multiple clusters. For example, while clustering news articles into topics, we may want to let the system to label an article belonging both 'economics', 'politics' and maybe more groups. Furthermore, even if the ultimate aim is to assign each object to a single group, we may want to have a more 'softer' membership decision, so that more typical members of a group get higher membership score/status.

There are a number of *soft* or *fuzzy* versions of the clustering algorithms we discussed above. A typical approach is to weigh the membership of each object based on their distance from the centroid. For hierarchical clustering methods, a statistical/soft clustering can be obtained using bootstrap, or doing multiple clustering experiments with added noise. Rather than reviewing these methods, we will go through a more basic approach, *density estimation*, below.

In density estimation, we assume that the data at hand is generated from a number of (parametric) probability density functions. The task is, then, finding the parameters of the probability distribution(s) given the data. Once the parameters of the underlying probability distributions are estimated, likelihood of the parameters given each data point can be calculated based on the estimated probability density or probability mass function.[6] As a result, we can quantify the degree of membership to one of the distributions (or clusters). A hard cluster membership decision can be made by simply choosing distribution which assigns the highest likelihood to the each point in question.

[6] The likelihood of parameter(s), θ, of a model, defined by such as a probability distribution, given the data x is,

$$\mathcal{L}(\theta \mid x) = p(x \mid \theta)$$

where $p(\cdot)$ is the value of *probability mass function* PMF for a discrete distribution, and *probability density function* (PDF) for a continuous distribution.

Remember that a discrete probability distribution can be characterized by a PMF that returns probabilities. Continuous variables are characterized by a PDF whose values are arbitrary non-negative real numbers (not probability values).

A popular instance of mixture models assume that the underlying distributions are normal (Gaussian) distributions, and resulting models are known as *mixtures of Gaussians* or *Gaussian mixtures*. In principle, however, there is no reason to limit the density estimation to Gaussian distributions, and examples of other distributions, such as categorical or multinomial distributions are common in computational linguistics literature.

As in clustering, finding the optimum parameters is intractable for most applications of density estimation. As a result, we often opt for a solution that finds a local optimum. Below, we will briefly describe the *expectation-maximization* (EM) algorithm. The EM algorithm is a very well-studied and popular iterative algorithm for solving learning problems involving latent, or unobserved, variables. In density estimation, the latent variables are the parameters of the hypothesized probability densities.

A general sketch of the EM algorithm is given below. As described below, the EM is a family of algorithms. A concrete algorithm can be formed based on the particular application, and the choice of the model (the number and type of the distributions).

1. Initialize the parameters (e.g., randomly) of K multivariate normal distributions $(\mu, \boldsymbol{\Sigma})$

2. Iterate until convergence:

E-step Given the parameters, compute the membership 'weights' (sometimes called *responsibilities*), the likelihood of each data point belonging to each hypothesized distribution

M-step Re-estimate the mixture density parameters using the calculated membership weights in the E-step

Note that the algorithm is very similar to the k-means algorithm described above. In fact, if we choose our mixture densities to be K Gaussian distributions, we get a soft version of the k-means algorithm.

Below, we will go through a very simple example for the purposes of understanding the density estimation and the EM algorithm. We will assume that our data comes from 2 normal distributions with a known variance but different means μ_1 and μ_2. Hence, estimating μ_1 and μ_2 allows us to characterize the underlying distributions. However, we need one more parameter, α, for specifying the ratio of the data points generated from the first distribution (the distribution with mean μ_1).[7] From a Bayesian point of view, the parameter α can be thought as the prior probability of a point belonging to the first distribution.

[7] Note, again, that we are making these simplifications to aid understanding. The EM algorithm can be used for more complex problems. The method we describe here can easily be adapted to more than 2 distributions. This requires estimating parameters (e.g., μ) for more distributions, and instead of using a scalar α, we employ a vector, $\boldsymbol{\alpha}$, whose components specifying the proportion of data points generated from each distribution (hence, $\sum_i \alpha_i = 1$). Similarly, we can generalize the algorithm to multivariate Gaussian distributions by allowing μ_i to be vectors in a higher dimensional space. There is also no reason (other than simplicity) for having a fixed variance, we can estimate the variances of each distribution, as well as modeling covariances.

In this simplified example, the parameters we want to estimate are μ_1, μ_2, and α. We will collectively call these parameters θ. In other words, the parameters of interest is the vector $\theta = (\mu_1, \mu_2, \alpha)$.

The EM algorithm starts with initializing the parameters. Typically, the initialization is done randomly, but as in k-means, 'more informed' initialization methods exist.

In the E-step, for every data point x_i, we calculate the membership weights, that is $p(k = 1 \,|\, x_i\theta)$ and $p(k = 2 \,|\, x_i\theta)$, where we use the notation $k = 1$ to indicate that the data point belongs to the distribution with mean μ_1. For n data points, the membership weights can be represented by a $n \times 2$ matrix A, such that $a_{i,k}$ indicates the membership weight of i^{th} data point being generated from the k^{th} distribution. We can calculate $a_{i,1}$, using

$$a_{i,1} = p(k = 1 \mid x_i, \theta) = \frac{\alpha p(x_i \mid k = 1, \mu_1)}{\alpha p(x_i \mid k = 1, \mu_1) + (1 - \alpha) p(x_i \mid k = 2, \mu_2)}$$

where $p(x_i \mid k = 1, \mu_1)$ is simply calculated using the normal PDF.[8] The membership weights for the distribution with mean μ_2, follows from the same equation. However, since we have only two classes, we can simply calculate it by subtracting the corresponding weight of the first distribution from 1 ($a_{i,2} = 1 - a_{i,1}$).

[8] With known variance σ,

$$p(x_i \mid k = 1, \mu_1) = \frac{e^{-\frac{(x_i - \mu_1)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}}$$

In the M-step, assuming that the membership assignments are correct, we re-estimate the parameters based on the membership weights calculated in the E-step. The mixture parameter α can be calculated by dividing the weighted counts (n_1) of the first distribution membership weights to the total number of data points.

$$\alpha^{\text{new}} = \frac{n_1}{n}$$

where $n_1 = \sum_{i=1}^{n} a_{i,1}$. The weighted count for the second distribution can be calculated similarly, or by simply $n_2 = n - n_1$. The means of the distributions are also estimated similarly, as a weighted average of each data point.

$$\mu_1^{\text{new}} = \frac{1}{n_1} \sum_{i=1}^{n} a_{i,1} x_i$$

Now, we have new values for each member of $\boldsymbol{\theta}$, we can return to E-step recalculating the membership weight matrix $\mathbf{A}$. The algorithm alternates between E and M steps until a defined convergence criterion is reached. The convergence criterion is typically based on log-likelihood of the data. We observe the likelihood at each iteration, and stop when it does not improve (more than a defined threshold). The log-likelihood is the logarithm of the likelihood of the data, under the model assumptions. Assuming that the data points are conditionally independent given the model, likelihood of the model parameters can be computed as

$$\mathcal{L}(\theta \mid x) = p(x \mid \theta) \sum_{i=1}^{n} \alpha p(x_i \mid k = 1, \mu_1) + (1 - \alpha) p(x_i \mid k = 2, \mu_2)$$

as before, $p(x_i \mid k = 1, \mu_1)$ and $p(x_i \mid k = 2, \mu_2)$ is calculated by plugging relevant x, μ and the fixed-known σ into the normal PDF function.[9]

[9] Remember that because of computational reasons, we work with logarithm of the likelihood in practice.

As noted above, the EM algorithm is a popular, well-studied algorithm for many problems involving latent variables. Probably one of the reasons for its popularity is that it is guaranteed to converge. However, it converges to a local optimum value. As a result, similar to the k-means algorithm, one way to look for better solutions is to re-estimate the parameters with different (random) initializations. Similarly, as in k-means, the number of components (or distributions) has to be specified. We do not discuss the extensions of mixture models for unknown number of components, but we will revisit the problem briefly in the next chapter.

8.3 *Dimensionality reduction*

Another family of well-known and popular methods in unsupervised learning is *dimensionality reduction*. In many problems, we have to deal with a (very) high dimensional data. However, most of these dimensions are highly correlated, and 'meaningful' dimensions of the data is often much lower. A typical example in computational linguistics is characterization of a collection of documents based on (a measure of) the words that occur in each document. In this type of representation, each document is represented with a vector whose size is equal to the size of the vocabulary. Arguably, however, the differences or similarities between the documents can be attributed to a much lower number of latent dimensions (for example, related to the topic or style). Finding these underlying/latent dimensions has a number of benefits, including the following.

- It is easier to visualize and understand lower-dimensional data
- Reducing the dimensionality helps reduce the computational complexity of the other methods applied to the data
- Related to above, one can use such a dimensionality reduction as a lossy compression method
- The lower-dimensional representation removes the noise, potentially resulting in better generalizations

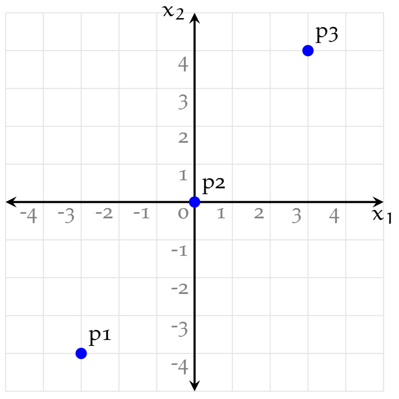


Figure 8.13: Three data points in $\mathbb{R}^2$.

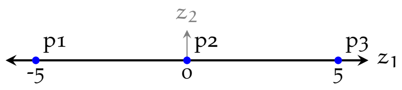
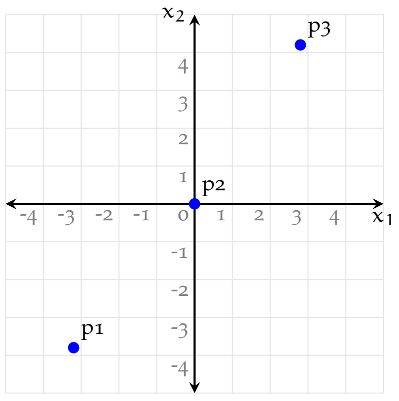


Figure 8.14: A transformation (rotation) applied to the data in Figure 8.13.



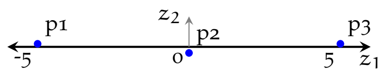


Figure 8.15: A slight variation of the 2D data in Figure 8.13 without perfect correlation (top), and its rotated version in Figure 8.14 applied to it (bottom).

We motivate the dimensionality reduction with a simple artificial example. Figure 8.13 presents three data points in two-dimensional Euclidean space ($\mathbb{R}^2$). Although our data points are represented by two features, a careful look at the figure shows that all of our data points perfectly lines up on a single line. Intuitively, the two-dimensional representation is unnecessary. This can be seen in Figure 8.14 where the data points are mapped to another coordinate system by a simple linear transformation (rotation). Note that in Figure 8.14, the dimension z_2 is not necessary.

In real-world data sets, it is not usual to come accross perfectly correlated variables as in Figure 8.13. However, in many real data sets, some features/dimensions in the data have very little information given the others. Figure 8.15 shows the result of applying the same transformation to a similar data set without perfect correlation (but with very high correlation). The z_2 dimension in the transformed version of this data set shows some variation. Hence, it is not completely redundant. However, it is also clear that if we remove z_2 (set it to 0 for all data points), we do not lose much information. If we do the reverse transformation, or 'reconstruct' the original data from the bottom part of Figure 8.15, the result will be as in Figure 8.13, which is not equal to the top panel of Figure 8.15, but it is not too far from it, i.e., the reconstruction error is small.

Dimensionality reduction is a general term used for a number of methods that reduce the dimensionality of the data by exploiting the dependencies in a high-dimensional feature space. There are a number of different methods for dimensionality reduction. Here, we will discuss a linear method, *principal component analysis* (PCA). The PCA is simplest, and probably most popular, of the dimensionality reduction techniques. Most methods can be considered as extensions of PCA.

Our toy example above, in fact, demonstrates the PCA. Similar to this informal example, the PCA performs a linear transformation (rotation), of the original data set into another linear space such a way that the variables/dimensions are no more correlated. In other words, the new features, the principal components, are a linear combination of the original features. For example, in our example above, $z_1 = \frac{3}{5}x_1 + \frac{4}{5}x_2$ and $z_2 = -\frac{4}{5}x_1 + \frac{3}{5}x_2$. The transformation by itself does not reduce the dimensionality of the data (we still have two components for $\boldsymbol{z}$ in the example above). However, we can discard the latent dimensions with low variance as they do not contain a lot of information, and discarding these dimensions will have little effect on the reconstruction error.

The discussion above hints at a few different ways to look at the PCA, and, in general, dimensionality reduction. First, we can view it as a procedure to find the direction of the highest variance, the first *principal component*, then finding another direction with the highest variance which is (linearly) independent from, as a result perpendicular to, the first one, and find yet the direction of the largest variance that is perpendicular to previously found components, and so on. Figure 8.16 demonstrates this idea. Since we have only two dimensions in Figure 8.16, the second principal component is simply perpendicular to the first one.

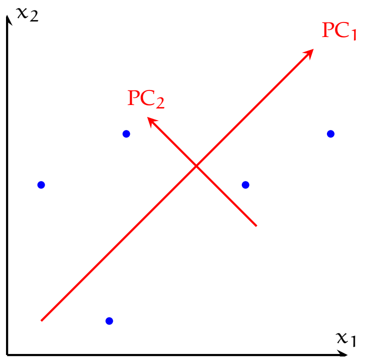


Figure 8.16: Interpretation of the PCA as finding the direction of the highest variance.

A second way to view PCA is finding a lower dimensional representation such that the reconstruction error is minimum. Figure 8.17 demonstrates this in two dimensions. If we reduce the data to a single dimension, we would like to find the line (the lower dimensional space) with the minimum reconstruction error, which turns out to be the line that has the minimal average distance from all data points. The PCA in this example would map original data points (blue) to the points on the first principal component (red points). The mapping with the least error is the mapping with the minimum perpendicular distance between the data points and their image under the transformation. Considering this is a linear transformation, the idea translates to higher dimensional spaces trivially.

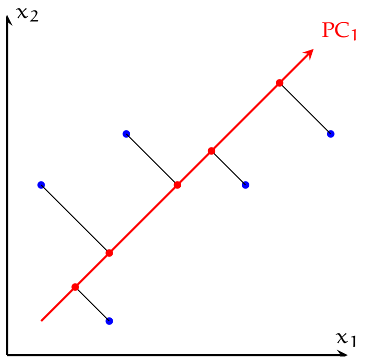


Figure 8.17: Interpretation of the PCA as finding a lower dimensional representation with minimum reconstruction error.

Yet another view is to assume that the data is generated by a lower dimensional hidden (or latent) variable, then it is mapped to a higher dimensional space with some added noise. This view is similar to our discussion of clustering and mixture densities. However, the latent variable in this case is a continuous one, while clustering and density estimation assumes a categorical latent variable. In particular, the PCA, according to this interpretation, assumes a multi-variate latent Gaussian variable.

Depending on the view, there are different ways to formulate a procedure for performing the PCA. Following the variance maximization approach, we can use a few well-established procedures from linear algebra. For reducing the reconstruction error, we can write an appropriate objective function, and find the principal components that minimize it. If we take the latent-variable view, we can, for example, use the EM algorithm to estimate the parameters of the lower-dimensional latent variable, similar to estimating the mixture models. Here, we will only discuss the first approach in a bit more detail.

The first approach for obtaining principal components is a well-known method from linear algebra called *eigenvalue decomposition*. Here we will limit ourselves to a special case of eigenvalue decomposition, where the matrix we want to decompose is symmetric. For a symmetric matrix $\mathbf{A}$, the eigenvalue decomposition results in

$$\mathbf{A} = \mathbf{U}\Lambda\mathbf{U}^T$$

where columns of $\mathbf{U}$ are orthogonal to each other, and they are the eigenvectors of $\mathbf{A}$. As a result, the columns of $\mathbf{U}$ are not correlated.[10] Λ is a diagonal matrix with corresponding eigenvalues.

Remember that for the PCA, we are interested in a linear transformation where the resulting features are decorrelated, and we also want to be able to tell the features that are associated with the large variance. When applied to a covariance matrix, the decomposition above gives us both: the eigenvectors are decorrelated (orthogonal to each other), and the larger the eigenvalues, the larger the amount of variance. For the rest of the discussion here, we will assume that our variables are standardized.[11]

To perform PCA using eigenvalue decomposition, we simply apply it to the covariance matrix ($\boldsymbol{\Sigma} = \mathbf{U}\Lambda\mathbf{U}^T$).[12] The columns of $\mathbf{U}$ is now the directions of the principal components, and the diagonal values of Λ indicate the magnitude of the variance in the corresponding direction. The matrix of eigenvectors, $\mathbf{U}$, transforms the vectors from the original input space to the space of the principal components. Any (column) vector $\mathbf{x}$ can be transformed by $\mathbf{z} = \mathbf{U}^T\mathbf{x}$.[13] If we want to transform $\mathbf{z}$ back to the original feature space, all we need to do is to use the inverse transformation. A nice property of the orthogonal matrices is that their inverse is their transpose. As a result, the inverse transformation is simply $\mathbf{x} = \mathbf{U}\mathbf{z}$.

Note, however, that we have not done any dimensionality reduction yet. If our original feature space has m dimensions, the dimension of the covariance matrix $\boldsymbol{\Sigma}$, and hence the dimensions of the matrix $\mathbf{U}$, will be $m \times m$, multiplying $\mathbf{U}$ with any m-dimensional vector $\mathbf{x}$ results in another m-dimensional vector $\mathbf{z}$. Instead of using the complete matrix $\mathbf{U}$ we can pick only the eigenvectors (columns) that correspond to the largest d eigenvalues. This will result in a matrix $\mathbf{U}_L$ with $m \times d$ dimensions whose transpose can be used to transform m-dimensional input to the lower d-dimensional space.

Although the eigenvalue decomposition explained above demonstrate the PCA conceptually as decorrelation and maximizing variance, another well-known method from linear algebra, *singular value*

[10] Remember the relation between orthogonality and independence.

[11] Remember that to standardize a variable, we *center* it by subtracting the mean of the variable, and *scale* it with it standard deviation. The standardized version of the variable x with mean μ and standard deviation σ is

$$z = \frac{x - \mu}{\sigma}.$$

Note that this is a linear (more correctly, an affine) mapping between x and z. Any linear operation on z can be expressed in terms of x by the inverse transformation.

[12] Note that if our data matrix $\mathbf{X}$ is arranged such that columns represent the features, and rows represent the instances (data points), and if the vectors representing each data point are standardized,

$$\boldsymbol{\Sigma} = \mathbf{X}^T\mathbf{X}.$$

[13] We can obtain a transformed version of the original data matrix, by $\mathbf{Z} = \mathbf{X}\mathbf{U}$. Note that this holds because the data vectors are rows of $\mathbf{X}$, and the transformed vectors will be the rows of $\mathbf{Z}$.

decomposition (SVD), is often more convenient for performing the PCA. The SVD factorizes the data matrix $\mathbf{X}$ directly such that

$$\mathbf{X} = \mathbf{V}\mathbf{D}\mathbf{U}^T$$

where $\mathbf{V}$ and $\mathbf{U}^T$ are orthogonal matrices, and $\mathbf{D}$ is a diagonal matrix of singular values. As in eigenvalue decomposition, rows of $\mathbf{U}^T$ contain the eigenvectors. The diagonal matrix $\mathbf{D}$ is related to eigenvalues ($\mathbf{D}^2 = \mathbf{\Lambda}$). The result, hence, is equivalent. The SVD has some other applications we will get back later in this course.

8.4 Unsupervised learning with neural networks

Typical neural networks are trained using backpropagation, which requires supervision, an error signal to be backpropagated thorough the network. However, there are a number of ways to use neural networks as unsupervised learners. Here we will introduce a typical unsupervised method to train neural networks, *autoencoders*, and briefly note two, a well-known generative network (RBMs), and a more recent use of both generative and discrimantive methods (adverserial networks).

8.4.1 Restricted Boltzmann machines

As in other unsupervised approaches we discussed earlier, one way to formulate unsupervised learning through a neural network is to use hidden or latent variables. *Restricted Boltzmann Machines* (RBMs) is such a network architecture consisting of one hidden layer (the latent variable) and an input layer (depicted in Figure 8.18). The layers of the RBM are fully connected, but there are no links within the layers. Although it is not a requirement, typically the hidden layer has a smaller dimension than the input layer. Hence, the idea is very similar to PCA, or other dimensionality reduction methods. We present an input to the network, and obtain a useful (lower dimensional) representation of the input at the hidden layer.

In an RBM, we do not have outputs (or inputs, the RBM models the data as is, without an input–output distincition). Hence, we cannot just define a loss function based on the expected output, and use backpropagation to train the network. RBMs are generative models that model the joint probability of the hidden variable and the input. The joint probability distribution defined by an RBM is[14]

$$p(\mathbf{h}, \mathbf{x}) = \frac{e^{\mathbf{h}^T \mathbf{W} \mathbf{x}}}{Z}.$$

where Z a normalizing constant (a sum over all possible configurations) that makes sure that the result is a proper probability distribution. Learning in an RBM, then, becomes finding the parameters that maximizes this joint probability. Exact solution to this learning problem is intractable. In practice often an approximate solution is found using *contrastive divergence* algorithm, which is an iterative algorithm similar to the EM algorithm discussed earlier.

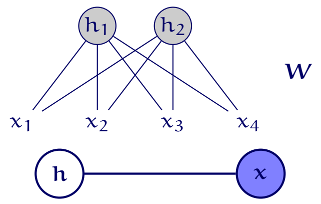


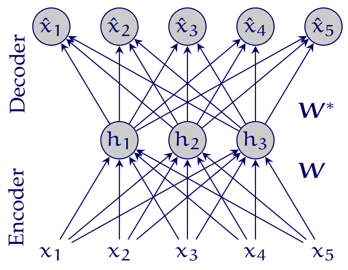
Figure 8.18: A schematic description of a restricted Boltzmann machine (top), and equivalent representation as a (undirected) graphical model (bottom).

[14] Notice the similarity of this formula with the multinomial logistic regression discussed earlier.

8.4.2 Autoencoders

RBMs are theoretically interesting models. However, training RBMs is computationally expensive, and since training them requires a special algorithm, it does not benefit from many developments in the (supervised) neural network literature. *Autoencoders* are a practical solution to elevate some of these problems.

An autoencoder is a standard feed-forward network trained to predict its own input. Figure 8.19 presents a typical autoencoder. We can consider an autoencoder in two parts, an *encoder* that takes the input and encodes in a hidden layer, and a *decoder* that takes the hidden representations and reconstructs the output.[15] The weights of the encoder and decoder may be *shared*. That is, the weights of the decoder ($\boldsymbol{W}$) are the transpose of the weights of the encoder ($\boldsymbol{W}^*$).[16]



$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$
Decoder
$\boldsymbol{W}^*$
h_1 h_2 h_3
$\boldsymbol{W}$
Encoder
x_1 x_2 x_3 x_4 x_5

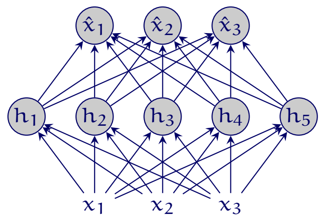
Figure 8.19: An autoencoder, a standard feed-forward network predicting its own input.

[15] The encoder–decoder architecture is not restricted to 'auto' encoding. They are also used in solving problems where input–output pairs are different, e.g., machine translation.

[16] Weight sharing is a common concept in neural networks, especially in more complex models. When it makes sense, weight sharing reduces the model complexity, and may result in learning better models.

Since the network has to predict its own input, through a lower dimensional representation in the hidden layer, it needs to learn a hidden representation with minimum reconstruction error. Again, the idea is similar to PCA. An autoencoder with a single layer, in fact, approximates the PCA. Autoencoders with multiple hidden layers (deep autoencoders), can learn non-linear relationships between the input variables. Besides the potential benefit of discovering non-linear relationships, another practical benefit of autoencoders is their memory efficiency. The matrix factorization techniques used for fitting a PCA model may become inefficient on large data sets. Since autoencoders are trained like a standard feed-forward network, they can be efficiently trained using small batches.

Autoencoders are typically used to learn a lower dimensional hidden representation. However, it is also possible to use a larger hidden representation as shown in Figure 8.20. Such autoencoders are called *over-complete* autoencoders.[17] A particular use of such autoencoders is to train them with L1 regularization, forcing them to learn sparse features form more complex input features. The hidden representation in this case 'disentangles' some of the complex features to simpler ones – which are hopefully more interpretable or useful for a downstream task.

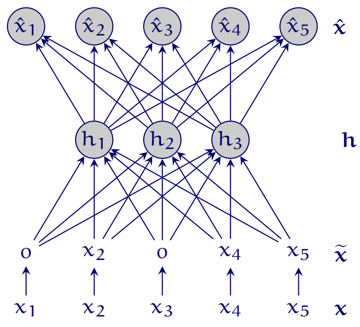


$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$
h_1 h_2 h_3 h_4 h_5
x_1 x_2 x_3

Figure 8.20: An over-complete autoencoder.

[17] Not surprising, the autoencoders with a lower dimensional hidden representation, like the one in Figure 8.19, are called *under-complete* autoencoders.

Another variation of autoencoders is *denoising* autoencoders. As shown in Figure 8.21, the input of a denoising autoencoder is corrupted with noise. When trained, the autoencoder learns to eliminate the type of noise introduced. Hence, possibly being useful for removing noise, e.g., from images or sound signals. Note that unlike the typical use of autoencoders where we are interested in the hidden representations rather than the output of the network, in this use, we are interested in full reconstruction.



$\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{\boldsymbol{x}}$
h_1 h_2 h_3 $\boldsymbol{h}$
0 x_2 0 x_4 x_5 $\tilde{\boldsymbol{x}}$
x_1 x_2 x_3 x_4 x_5 $\boldsymbol{x}$

Figure 8.21: A denoising autoencoder.

8.4.3 Generative adversarial networks

Another, rather recent but highly influential method is *generative adversarial networks* (GANs). Although it is introduced, and mainly used with (deep) neural networks, the interesting part of GANs is the training regime of the system. In a GAN, there are two classifiers; a generative model (similar to RBMs) that tries to generate objects of interest (e.g., paintings by an artist, or novels of an author), while a discriminative model tries to distinguished the true instances of the objects from the objects generated by the generative model. The interesting part of the approach is that, the success (or failure) of the discriminative network is used as the supervision signal for the generative one. The task of the generative model is to fool the discriminative model. As it becomes more successful in doing so, it learns a better representation for the data.

We will not discuss GANs in detail here, but they are one of the relatively recent developments that became popular in a number of fields, even with some impact on popular culture.[18]

[18] Just to exemplify, a painting generated by a GAN was recently sold for $432 500.

Summary

The methods covered in this lecture consists of unsupervised methods. We covered three traditional unsupervised methods. Namly, clustering, density estimation and dimensionality reduction. On the (deep) neural network site, we described autoencoders, which are simply feed-forward networks predicting their own input, and noted two interesting models briefl, RBMs and adverserial netowrks.

More information traditional methods can be found in almost any machine learning text book (e.g., in our usual references, Hastie, Tibshirani, and Friedman 2009; MacKay 2003; Bishop 2006). You can also find discussion of autoencoders in recent textbooks on neural network (**marsland2015**; e.g., Goodfellow, Bengio, and Courville 2016).

Bibliography

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer. ISBN: 978-0387-31073-2.

Goodfellow, Ian J, Yoshua Bengio, and Aaron Courville (2016). *Deep learning*. MIT Press.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second. Springer series in statistics. Springer-Verlag New York. ISBN: 9780387848587. URL: `http://web.stanford.edu/~hastie/ElemStatLearn/`.

MacKay, David J. C. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press. ISBN: 978-05-2164-298-9. URL: `http://www.inference.phy.cam.ac.uk/itprnn/book.html`.