Clustering: Models of High Dimensional Data

12

High-dimensional data comes with problems. Data points tend not to be where you think; they can scattered quite far apart, and can be quite far from the mean. There is an important rule of thumb for coping with high dimensional data: **Use simple models**. One very good, very simple, model for high dimensional data is to assume that it consists of multiple blobs. To build models like this, we must determine which datapoints belong to which blob by collecting together data points that are close and forming blobs out of them. This process is known as **clustering**. It is so useful that there is a very wide range of clustering algorithms

One important application for clustering methods is building features. If we need to classify signals (eg audio, images, video, accelerometer data) that have repeated structures in them, clustering pieces of signal in the training set will expose what structures are repeated commonly. A new signal can be described by recording how often each cluster center appears in the signal. This process yields a convenient and effective feature description of the signal that can be fed into the classifiers from the previous chapter.

Clustering is a somewhat puzzling activity. It is extremely useful to cluster data, and it seems to be quite important to do it reasonably well. But it surprisingly hard to give crisp criteria for a good (resp. bad) clustering of a dataset. Usually, clustering is part of building a model, and the main way to know that the clustering algorithm is good is that the resulting model is useful.

12.1 The Curse of Dimension

High dimensional models display uninituitive behavior (or, rather, it can take years to make your intuition see the true behavior of high-dimensional models as natural). In these models, most data lies in places you don't expect. A very simple example dataset will illustrate these problems. The dataset is an IID sample from a uniform probability density in a cube, with edge length two, centered on the origin. At every point in the cube, this density $P(x) = \frac{1}{2^d}$ (and it is zero elsewhere). The mean of this density is at the origin, which we write as $\mathbf{0}$. Each component of every $\mathbf{x}_i$ must lie in the range $[-1, 1]$.

12.1.1 Minor Banes of Dimension

It is difficult to build histogram representations for high dimensional datasets, because you end up with too many boxes. In the case of our cube, imagine we wish to divide each dimension in half (i.e. between $[-1, 0]$ and between $[0, 1]$), which would produce a very crude histogram. Then we must have 2^d boxes. This presents two problems. First, we will have difficulty representing this number of boxes. Second, unless we are exceptionally lucky, most boxes must be empty because we will not have 2^d data items. Splitting each dimension into a larger number of pieces just makes things a great deal worse.

Covariance matrices are hard to work with because the number of entries in the matrix grows as the square of the dimension. This means the matrix can get big and difficult to store. More important, the amount of data we need to get an accurate estimate of all the entries in the matrix grows fast. As we are estimating more numbers, we need more data to be confident that our estimates are reasonable. There are a variety of straightforward work-arounds for this effect. In some cases, we have so much data there is no need to worry. In other cases, we assume that the covariance matrix has a particular

D. Forsyth, *Probability and Statistics for Computer Science*,
https://doi.org/10.1007/978-3-319-64410-3_12

parametric form, and just estimate those parameters. There are two strategies that are usual. In one, we assume that the covariance matrix is diagonal, and estimate only the diagonal entries. In the other, we assume that the covariance matrix is a scaled version of the identity, and just estimate this scale. You should see these strategies as acts of desperation, to be used only when computing the full covariance matrix seems to produce more problems than using these approaches.

12.1.2 The Curse: Data Isn't Where You Think It Is

The first surprising fact about high dimensional data is that most of the data can lie quite far away from the mean. For example, we can divide our cube into two pieces. $\mathcal{A}(\epsilon)$ consists of all data items where *every* component of the data has a value in the range $[-(1-\epsilon), (1-\epsilon)]$. $\mathcal{B}(\epsilon)$ consists of all the rest of the data. If you think of the data set as forming a cubical orange, then $\mathcal{B}(\epsilon)$ is the rind (which has thickness ϵ) and $\mathcal{A}(\epsilon)$ is the fruit.

Your intuition will tell you that there is more fruit than rind. It is wrong, as a simple calculation shows. We can compute $P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\})$ and $P(\{\mathbf{x} \in \mathcal{B}(\epsilon)\})$. These probabilities tell us the probability a data item lies in the fruit (resp. rind). $P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\})$ is easy to compute as

$$P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\}) = (2(1-\epsilon)))^d \left(\frac{1}{2^d}\right) = (1-\epsilon)^d$$

and

$$P(\{\mathbf{x} \in \mathcal{B}(\epsilon)\}) = 1 - P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\}) = 1 - (1-\epsilon)^d.$$

But notice that, as $d \to \infty$,

$$P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\}) \to 0.$$

This means that, for large d, we expect most of the data to be in $\mathcal{B}(\epsilon)$ (the rind is bigger than the fruit). Equivalently, for large d, we expect that at least one component of each data item is close to either 1 or -1. In high dimensions, volume doesn't behave like you think, and so you do not expect the rind to dominate. The fact that the dataset is a cube, rather than a sphere, makes the calculations easier (most people don't remember the expressions for volume of high dimensional spheres). But the shape has nothing to do with the real problem.

The fact that $P(\{\mathbf{x} \in \mathcal{A}(\epsilon)\})$ is small for large d suggests that much data is quite far from the origin, because most of it lies in $\mathcal{B}(\epsilon)$. This turns out to be true. It is easy to compute the average of the squared distance of data from the origin. We want

$$\begin{aligned}\mathbb{E}\left[\mathbf{x}^T\mathbf{x}\right] &= \mathbb{E}\left[\sum_i x_i^2\right] = \sum_i \mathbb{E}\left[x_i^2\right] \\ &= \sum_i \int_{\text{cube}} x_i^2 P(\mathbf{x}) d\mathbf{x}.\end{aligned}$$

Now each component of $\mathbf{x}$ is independent, so that $P(\mathbf{x}) = P(x_1)P(x_2)\ldots P(x_d)$. Now we substitute, to get

$$\begin{aligned}\mathbb{E}\left[\mathbf{x}^T\mathbf{x}\right] &= \sum_i \int_{-1}^{1} x_i^2 P(x_i) dx_i = \sum_i \frac{1}{2}\int_{-1}^{1} x_i^2 dx_i \\ &= \frac{d}{3},\end{aligned}$$

so as d gets bigger, most data points will be further and further from the origin. Worse, as d gets bigger, data points tend to get further and further from one another. We can see this by computing the average of the squared distance of data points from one another. Write $\mathbf{u}$ for one data point and $\mathbf{v}$ for another; we can compute

$$\mathbb{E}\left[d(\mathbf{u},\mathbf{v})^2\right] = \mathbb{E}\left[(\mathbf{u}-\mathbf{v})^T(\mathbf{u}-\mathbf{v})\right] = \mathbb{E}\left[\mathbf{u}^T\mathbf{u}\right] + \mathbb{E}\left[\mathbf{v}^T\mathbf{v}\right] - 2\mathbb{E}\left[\mathbf{u}^T\mathbf{v}\right]$$

but since $\mathbf{u}$ and $\mathbf{v}$ are independent, we have $\mathbb{E}\left[\mathbf{u}^T\mathbf{v}\right] = \mathbb{E}[\mathbf{u}]^T\mathbb{E}[\mathbf{v}] = 0$. This means

$$\mathbb{E}\left[d(\mathbf{u},\mathbf{v})^2\right] = 2\frac{d}{3}.$$

This means that, for large d, we expect our data points to be quite far apart, too.

Remember this: *High dimensional data does not behave in a way that is consistent with most people's intuition. Points are always close to the boundary and further apart than you think. This property makes a nuisance of itself in a variety of ways. The most important is that only the simplest models work well in high dimensions.*

12.2 Clustering Data

In high dimensional spaces, there is "too much" space for any reasonable amount of data to fill it up (that's what the curse of dimension is about). It is ineffective to cut space up into boxes and see how much data lies in each box: too many boxes, and not enough data. An alternative is to break up the dataset, rather than the space. We form **clusters**—coherent blobs of datapoints that are near one another. A **cluster center** is a summary of an entire cluster. One natural summary is the average of the elements of the cluster. Another natural summary is a data item that is close to all the items in the cluster.

Clusters have a variety of uses. For example, we could form a representation that will function very like a histogram by reporting the center of each cluster and the number of data items in each cluster. As another example, chunks of data that are similar should appear in the same cluster, so cluster centers can be used to build a dictionary of patterns that repeat in a dataset (Sect. 12.4).

12.2.1 Agglomerative and Divisive Clustering

There are two natural recipes to produce clustering algorithms. In **agglomerative clustering**, you start with each data item being a cluster, and then merge clusters recursively to yield a good clustering (Procedure 12.1). The difficulty here is that we need to know a good way to measure the distance between clusters, which can be somewhat harder than the distance between points. There are three standard choices. In **single-link clustering**, the distance between the two closest elements is the inter-cluster distance. This tends to produce "long" clusters. In **complete-link clustering**, the maximum distance between an element of the first cluster and one of the second is the inter-cluster distance. This tends to yield rounded clusters. In **group average clustering**, an average of distances between elements in the clusters is the distance. This also tends to yield rounded clusters.

Procedure 12.1 (Agglomerative Clustering) Choose an inter-cluster distance. Make each point a separate cluster. Now, until the clustering is satisfactory,

- Merge the two clusters with the smallest inter-cluster distance.

In **divisive clustering**, you start with the entire data set being a cluster, and then split clusters recursively to yield a good clustering (Procedure 12.2). The difficulty here is we need to know some criterion for splitting clusters. This tends to be something that follows from the logic of the application, because the ideal is an efficient method to find a natural split in a large dataset. We won't pursue divisive clustering further.

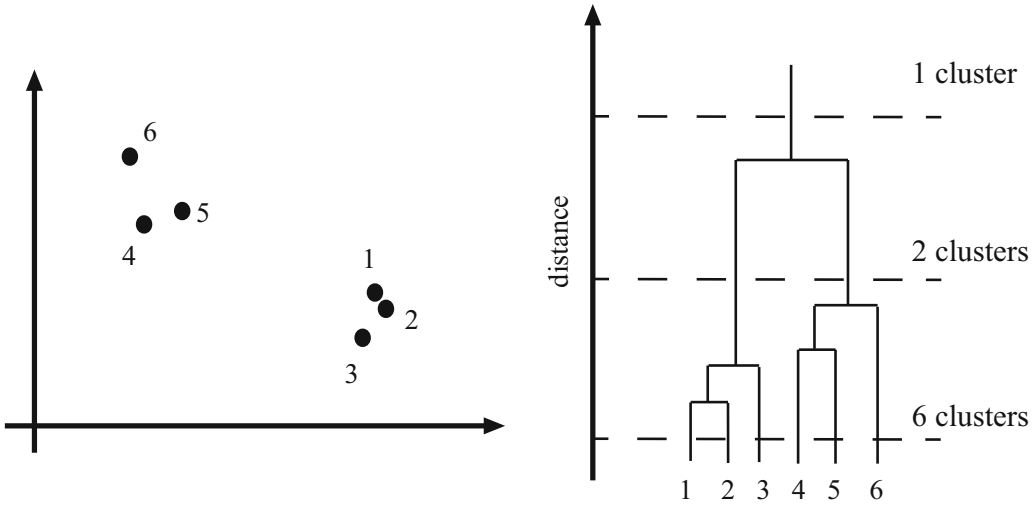

Fig. 12.1 *Left*, a data set; *right*, a dendrogram obtained by agglomerative clustering using single-link clustering. This representation makes it possible to guess how many clusters there are and to get some insight into how good the clusters are. Select a particular value of distance (vertical axis); now draw a a horizontal line at that distance. This will break the dendrogram into separate connected pieces, each of which is a cluster. I have marked some example distances on the dendrogram. Notice there is quite a large range of distances that yield two clusters. This is evidence that the two clusters are quite far apart, compared to their size, and is usually taken to mean that there is quite a large range of scales over which two clusters is a good clustering of the dataset

Procedure 12.2 (Divisive Clustering) Choose a splitting criterion. Regard the entire dataset as a single cluster. Now, until the clustering is satisfactory,

- choose a cluster to split;
- then split this cluster into two parts.

We need to know when to stop either algorithm. This is an intrinsically difficult task if there is no model for the process that generated the clusters. Both agglomerative and divisive clustering produce a hierarchy of clusters. Usually, this hierarchy is displayed to a user in the form of a **dendrogram**—a representation of the structure of the hierarchy of clusters that displays inter-cluster distances—and an appropriate choice of clusters is made from the dendrogram (see the example in Fig. 12.1).

Figure 12.1 illustrates some important properties of clustering, which many people find frustrating. There isn't a single right answer. Instead, different clusterings of a dataset might be acceptable; so, for example, in Fig. 12.1, point 6 might belong to a cluster with points 4 and 5, or it might reasonably be on its own. Whether an answer is good or not depends on appropriate scales of variation in the data. For example, in Fig. 12.1, if the distance between points 1 and 2 is "large", there are likely six different clusters; but if the distance between points 2 and 6 is "small", then there is likely one cluster. Whether a distance is "large" or "small" depends entirely on the application the data came from.

Worked example 12.1 (Agglomerative Clustering) Cluster the seed dataset from the UC Irvine Machine Learning Dataset Repository (you can find it at http://archive.ics.uci.edu/ml/datasets/seeds).

Solution This dataset consists of seven geometric parameters measured for wheat kernels of three types of wheat. It was donated by M. Charytanowicz, J. Niewczas, P. Kulczycki, P. A. Kowalski, S. Lukasik and S. Zak. You can find more information on the webpage. For this example, I used Matlab, but many programming environments will provide tools that are useful for agglomerative clustering. I show a dendrogram in Fig. 12.2). I deliberately forced Matlab to plot the whole dendrogram, which accounts for the crowded look of the figure (you can allow it to merge small leaves, etc.). As you can see from the dendrogram and from Fig. 12.3, this data clusters rather well. There isn't any choice of

(continued)

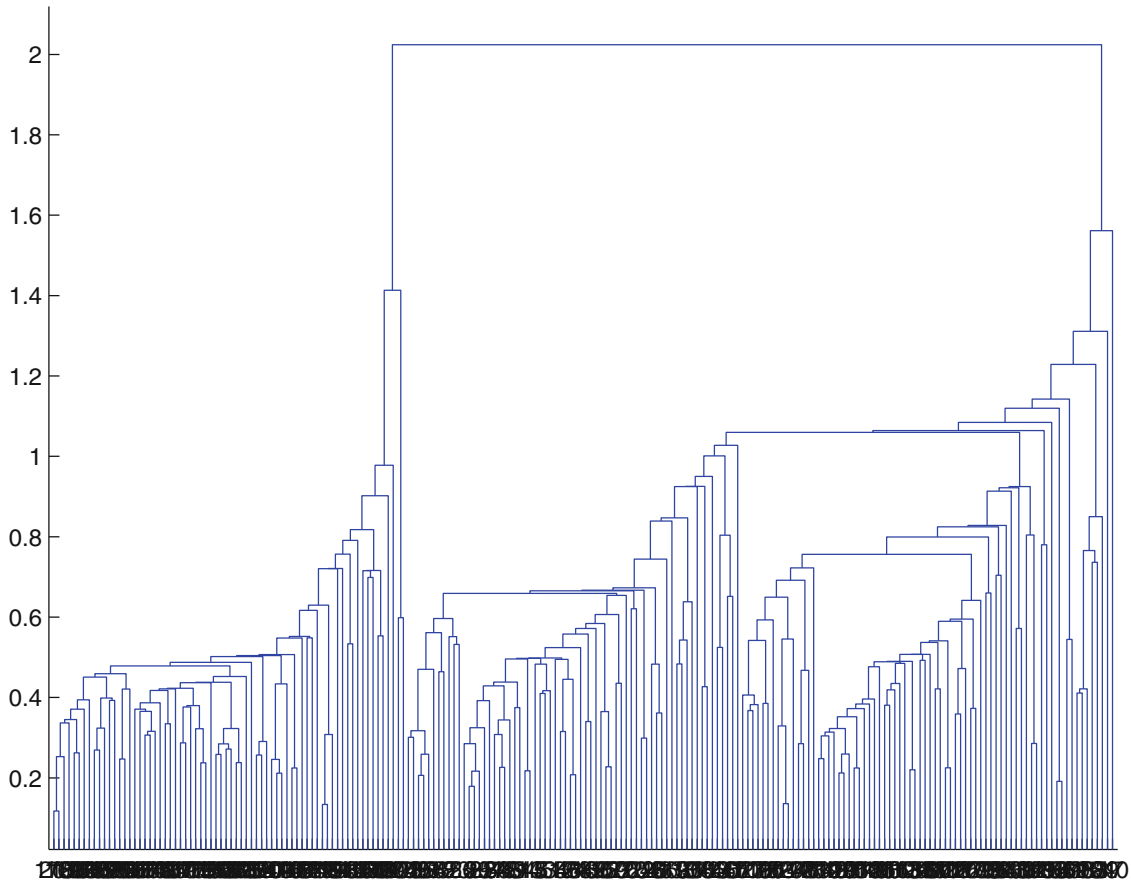

Fig. 12.2 A dendrogram obtained from the seed dataset, using single link clustering. Recall that the data points are on the horizontal axis, and that the vertical axis is distance; there is a horizontal line linking two clusters that get merged, established at the height at which they're merged. I have plotted the entire dendrogram, despite the fact it's a bit crowded at the bottom, because you can now see how clearly the data set clusters into a small set of clusters—there are a small number of vertical "runs"

distance that yields three cleanly separated clusters, even though there are three types of wheat here. At the same time, there is a fair range of distances that will yield three rather big clusters with a number of other small or even single point clusters. You can interpret this in terms of the feature space. The blobs of data corresponding to each type of wheat overlap somewhat, so some data items might be close to more than one blob. Furthermore, because the blobs are scattered, there are some points on the fringes of the blobs that are far from all others. This is pretty typical of real data. It's optimistic to expect that, because there are three types of wheat, there will be three clear and distinct clusters.

12.2.2 Clustering and Distance

You may have noticed the following, occasionally useful, property of agglomerative clustering. There is no need for a feature vector for any of the objects you wish to cluster; it is enough to have a table of distances between all pairs of objects. For example, you could collect data giving the distances between cities, without knowing where the cities are (as in Sect. 10.4.3, particularly Fig. 10.16), then try and cluster using this data. As another example, you could collect data giving similarities between breakfast items as in Sect. 10.4.3. These will be in the range [0, 1], where 0 is completely dissimilar and 1 is exactly the same. It is straightforward to turn the similarities into distances by taking the negative logarithm. This gives a useable table of distances. So it is possible to build a dendrogram and a clustering for the breakfast items of Sect. 10.4.3 without ever knowing a feature vector for any item. In this case, the distance is meaningful because you collected the distance information.

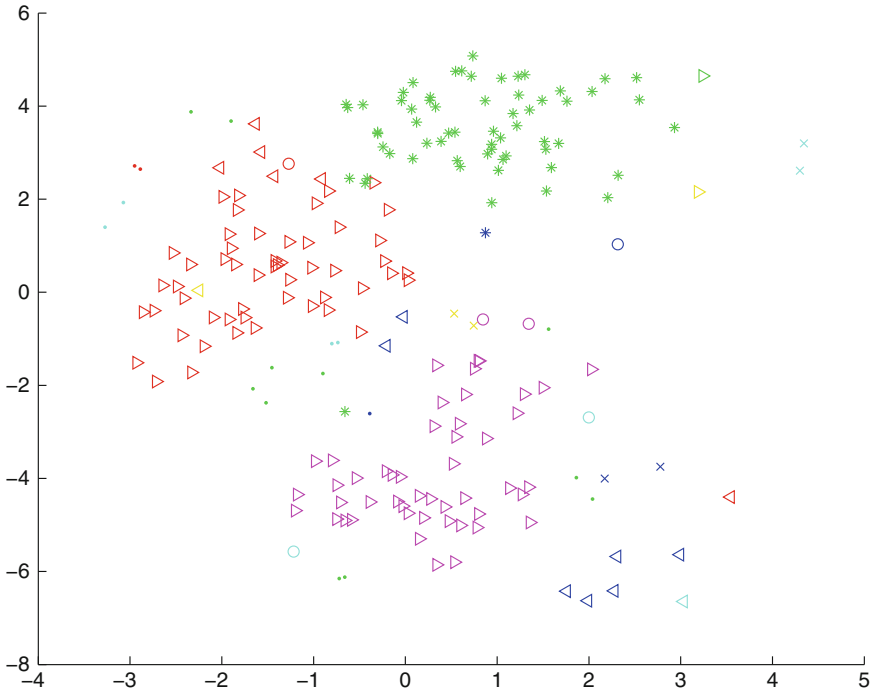

Fig. 12.3 A clustering of the seed dataset, using agglomerative clustering, single link distance, and requiring a maximum of 30 clusters. I have plotted each cluster with a distinct marker (though some markers differ only by color). Notice that there are a set of fairly natural isolated clusters. The original data is seven dimensional, which presents plotting problems; I show a scatter plot on the first two principal components (though I computed distances for clustering in the original seven dimensional space)

In the more usual case, we have a feature vector for each object. This doesn't necessarily mean that the distance between feature vectors is a good guide to the difference between objects. If the features are poorly scaled, distances (measured in the usual way) between data points may not be a good representation of their similarity. This is quite an important point. For example, imagine we are clustering data representing brick walls. The features might contain several distances: the spacing between the bricks, the length of the wall, the height of the wall, and so on. If these distances are given in the same set of units, we could have real trouble. For example, assume that the units are centimeters. Then the spacing between bricks is of the order of one or two centimeters, but the heights of the walls will be in the hundreds of centimeters. In turn, this means that the distance between two datapoints is likely to be completely dominated by the height and length data. This could be what we want, but it might also not be a good thing.

There are some ways to manage this issue. One is to know what the features measure, and know how they should be scaled. Usually, this happens because you have a deep understanding of your data. If you don't (which happens!), then it is often a good idea to try and normalize the scale of the data set. There are two good strategies. The simplest is to translate the data so that it has zero mean (this is just for neatness—translation doesn't change distances), then scale each direction so that it has unit variance. Another possibility, as in nearest neighbors (Sect. 11.2.1), is to transform the features so that the covariance matrix is the identity (this is sometimes known as **whitening**; the method follows from the ideas of Chap. 10).

Remember this: *High dimensional datasets can be represented by a collection of clusters. Each cluster is a blob of data points that are near one another, and is summarized by a cluster center. The choice of distance between data points has a significant effect on clustering. Agglomerative clustering starts with each data point a cluster, then recursively merges. There are three main ways to compute the distance between clusters. Divisive clustering starts with all in one cluster, then recursively splits clusters. The choice of splitting method depends quite strongly on application. Either method yields a dendrogram, which is a helpful summary of distances between points and clusters. For datasets that are small enough to plot the dendrogram, a look at a dendrogram can yield some helpful information about the data and good clusterings.*

12.3 The K-Means Algorithm and Variants

Assume we have a dataset that, we believe, forms many clusters that look like blobs. We would like to choose a clustering so that points are "close" to their cluster centers. It is natural to want to minimize the sum of squared distances from each point to its cluster center. We can even guess an algorithm for doing this. If we knew where the center of each of the clusters was, it would be easy to tell which cluster each data item belonged to—it would belong to the cluster with the closest center. Similarly, if we knew which cluster each data item belonged to, it would be easy to tell where the cluster centers were—they'd be the mean of the data items in the cluster. This is the point closest to every point in the cluster.

We can formalize this fairly easily by writing an expression for the squared distance between data points and their cluster centers. Assume that we know how many clusters there are in the data, and write k for this number. There are N data items. The ith data item to be clustered is described by a feature vector $\mathbf{x}_i$. We write $\mathbf{c}_j$ for the center of the jth cluster. We write $\delta_{i,j}$ for a discrete variable that records which cluster a data item belongs to, so

$$\delta_{i,j} = \begin{cases} 1 \text{ if } \mathbf{x}_i \text{ belongs to cluster } j \\ 0 \qquad\quad \text{otherwise} \end{cases}$$

We require that every data item belongs to exactly one cluster, so that $\sum_j \delta_{i,j} = 1$. We require that every cluster contain at least one point, because we assumed we knew how many clusters there were, so we must have that $\sum_i \delta_{i,j} > 0$ for every j. We can now write the sum of squared distances from data points to cluster centers as

$$\Phi(\delta, \mathbf{c}) = \sum_{i,j} \delta_{i,j} \left[(\mathbf{x}_i - \mathbf{c}_j)^T (\mathbf{x}_i - \mathbf{c}_j) \right].$$

Notice how the $\delta_{i,j}$ are acting as "switches". For the i'th data point, there is only one non-zero $\delta_{i,j}$ which selects the distance from that data point to the appropriate cluster center. It is natural to want to cluster the data by choosing the δ and $\mathbf{c}$ that minimizes $\Phi(\delta, \mathbf{c})$. This would yield the set of k clusters and their cluster centers such that the sum of distances from points to their cluster centers is minimized.

There is no known algorithm that can minimize Φ exactly in reasonable time. The $\delta_{i,j}$ are the problem: it turns out to be hard to choose the best allocation of points to clusters. The algorithm we guessed above is a remarkably effective approximate solution. Notice that if we know the $\mathbf{c}$'s, getting the δ's is easy—for the i'th data point, set the $\delta_{i,j}$ corresponding to the closest $\mathbf{c}_j$ to one and the others to zer. Similarly, if the $\delta_{i,j}$ are known, it is easy to compute the best center for each cluster—just average the points in the cluster. So we iterate:

- Assume the cluster centers are known and allocate each point to the closest cluster center.
- Replace each center with the mean of the points allocated to that cluster.

We choose a start point by randomly choosing cluster centers, and then iterate these stages alternately. This process eventually converges to a local minimum of the objective function (the value either goes down or is fixed at each step, and it is bounded below). It is not guaranteed to converge to the global minimum of the objective function, however. It is also not guaranteed to produce k clusters, unless we modify the allocation phase to ensure that each cluster has some nonzero number of points. This algorithm is usually referred to as **k-means** (summarized in Algorithm 12.3).

Procedure 12.3 (K-Means Clustering) Choose k. Now choose k data points $\mathbf{c}_j$ to act as cluster centers. Until the cluster centers change very little

- Allocate each data point to the cluster whose center is nearest.
- Now ensure that every cluster has at least one data point; one way to do this is by supplying empty clusters with a point chosen at random from points far from their cluster center.
- Replace the cluster centers with the mean of the elements in their clusters.

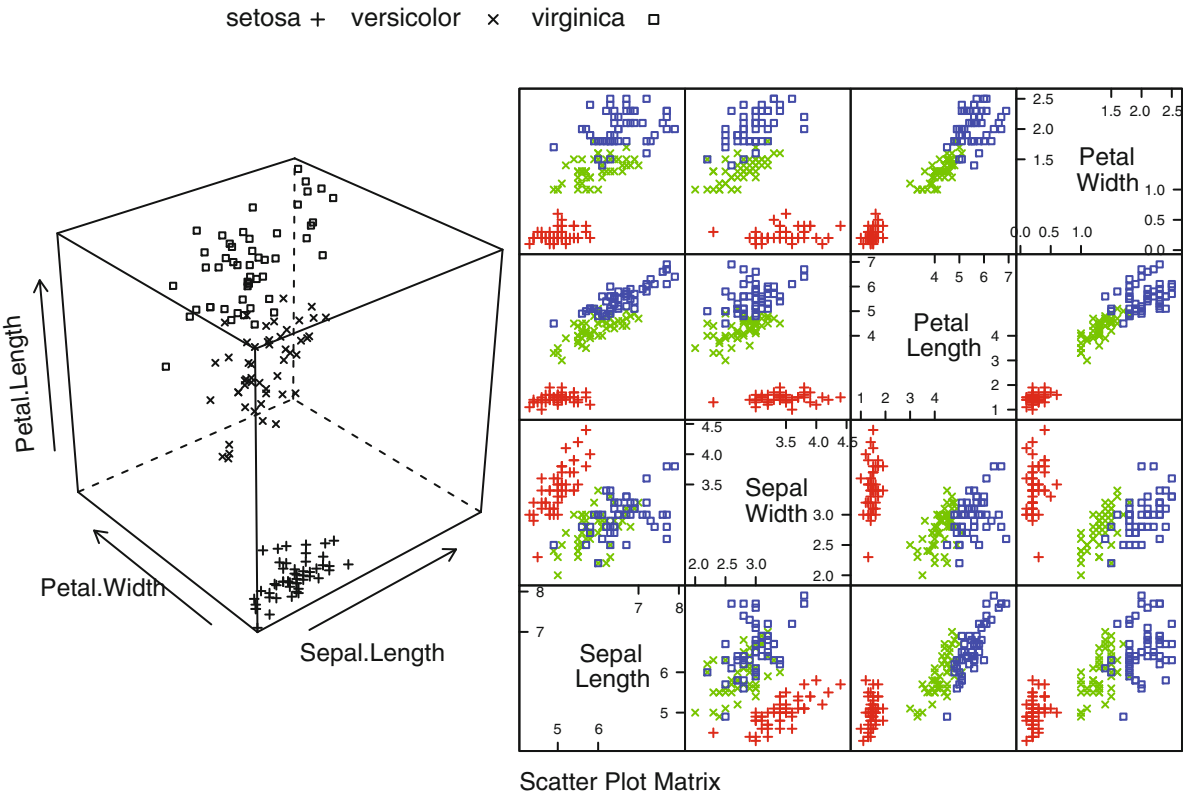

Fig. 12.4 *Left:* a 3D scatterplot for the famous Iris data, collected by Edgar Anderson in 1936, and made popular amongst statisticians by Ronald Fisher in that year. I have chosen three variables from the four, and have plotted each species with a different marker. You can see from the plot that the species cluster quite tightly, and are different from one another. *Right:* a scatterplot matrix for the Iris data. There are four variables, measured for each of three species of iris. I have plotted each species with a different marker. You can see from the plot that the species cluster quite tightly, and are different from one another

Usually, we are clustering high dimensional data, so that visualizing clusters can present a challenge. If the dimension isn't too high, then we can use panel plots. An alternative is to project the data onto two principal components, and plot the clusters there; the process for plotting 2D covariance ellipses from Sect. 12.5.2 comes in useful here. A natural dataset to use to explore k-means is the iris data, where we know that the data should form three clusters (because there are three species). Recall this dataset from Sect. 10.1.2. I reproduce Fig. 10.3 from that section as Fig. 12.4, for comparison. Figure 12.5 shows four different k-means clusterings of the data. By comparison with Fig. 12.4, notice how $k = 2$ clustering appears to merge the *versicolor* and *verginica* clusters. The $k = 3$ case appears to reproduce the species correctly. The $k = 4$ case appears to have broken *setosa* into two groups, but left *versicolor* and *verginica* as predicted by the species. The $k = 5$ case appears to have broken *setosa* into two groups, and *versicolor* and *verginica* into a total of three groups.

12.3.1 How to Choose K

The iris data is just a simple example. We know that the data forms clean clusters, and we know there should be three of them. Usually, we don't know how many clusters there should be, and we need to choose this by experiment. One strategy is to cluster for a variety of different values of k, then look at the value of the cost function for each. If there are more centers, each data point can find a center that is close to it, so we expect the value to go down as k goes up. This means that looking for the k that gives the smallest value of the cost function is not helpful, because that k is always the same as the number of data points (and the value is then zero). However, it can be very helpful to plot the value as a function of k, then look at the "knee" of the curve. Figure 12.6 shows this plot for the iris data. Notice that $k = 3$—the "true" answer—doesn't look particularly special, but $k = 2$, $k = 3$, or $k = 4$ all seem like reasonable choices. It is possible to come up with a procedure that makes a more precise recommendation by penalizing clusterings that use a large k, because they may represent inefficient encodings of the data. However, this is often not worth the bother.

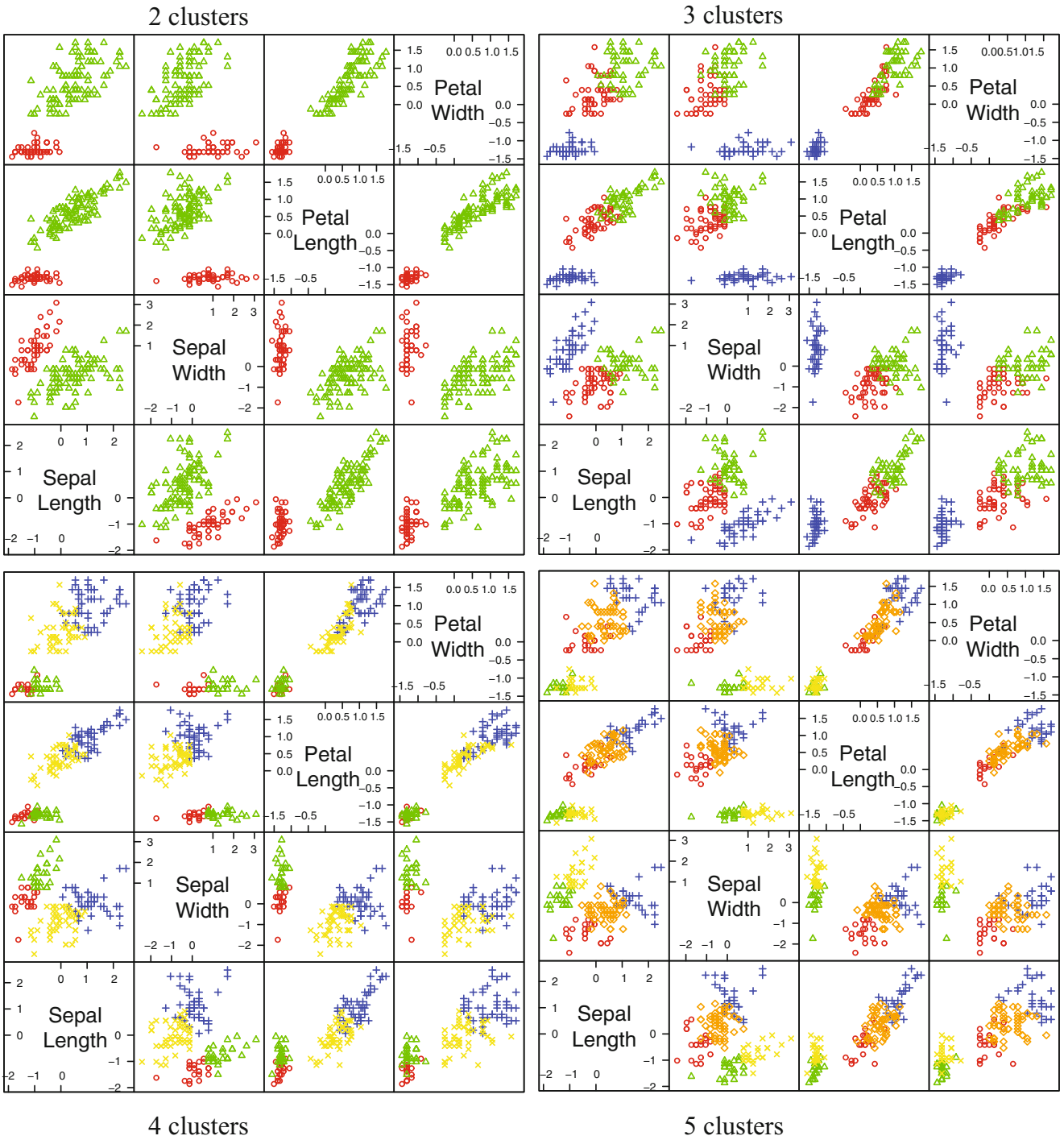

Fig. 12.5 Four panel plots of the iris data, clustered with k-means to different numbers of clusters

In some special cases (like the iris example), we might know the right answer to check our clustering against. In such cases, one can evaluate the clustering by looking at the number of different labels in a cluster (sometimes called the purity), and the number of clusters. A good solution will have few clusters, all of which have high purity. Mostly, we don't have a right answer to check against. An alternative strategy, which might seem crude to you, for choosing k is extremely important in practice. Usually, one clusters data to use the clusters in an application (one of the most important, vector quantization, is described in Sect. 12.4). There are usually natural ways to evaluate this application. For example, vector quantization is often used as an early step in texture recognition or in image matching; here one can evaluate the error rate of the recognizer, or

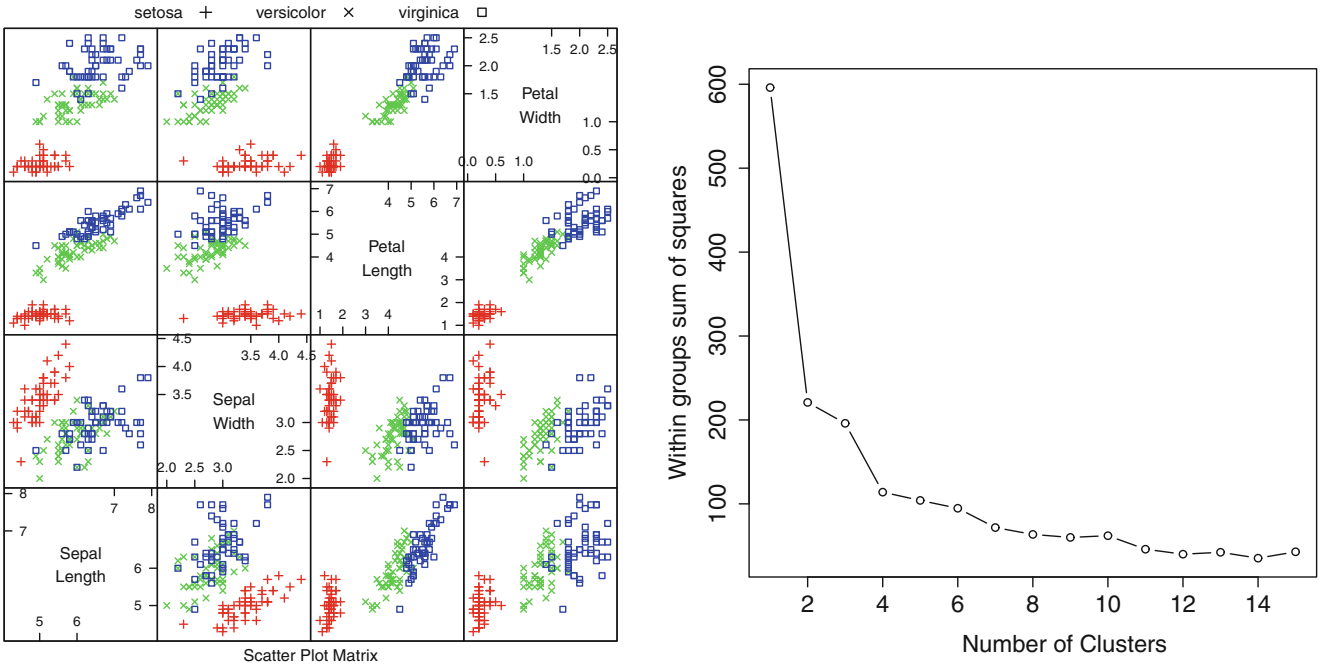

Fig. 12.6 On the *left*, the scatterplot matrix for the Iris data, for reference. On the *right*, a plot of the value of the cost function for each of several different values of k. Notice how there is a sharp drop in cost going from $k = 1$ to $k = 2$, and again at $k = 4$; after that, the cost falls off slowly. This suggests using $k = 2$, $k = 3$, or $k = 4$, depending on the precise application

the accuracy of the image matcher. One then chooses the k that gets the best evaluation score on validation data. In this view, the issue is not how good the clustering is; it's how well the system that uses the clustering works.

12.3.2 Soft Assignment

One difficulty with k-means is that each point must belong to exactly one cluster. But, given we don't know how many clusters there are, this seems wrong. If a point is close to more than one cluster, why should it be forced to choose? This reasoning suggests we assign points to cluster centers with weights. These weights are different from the original $\delta_{i,j}$ because they are not forced to be either zero or one, however. Write $w_{i,j}$ for the weight connecting point i to cluster center j. Weights should be non-negative (i.e. $w_{i,j} \geq 0$), and each point should carry a total weight of 1 (i.e. $\sum_j w_{i,j} = 1$), so that it if the i'th point contributes more to one cluster center, it is forced to contribute less to all others. You should see $w_{i,j}$ as a simplification of the $\delta_{i,j}$ in the original cost function. We can write a new cost function

$$\Phi(w, \mathbf{c}) = \sum_{i,j} w_{i,j} \left[(\mathbf{x}_i - \mathbf{c}_j)^T (\mathbf{x}_i - \mathbf{c}_j) \right],$$

which we would like to minimize by choice of w and $\mathbf{c}$. There isn't any improvement in the problem, because for any choice of $\mathbf{c}$, the best choice of w is to allocate each point to its closest cluster center. This is because we have not specified any relationship between w and $\mathbf{c}$.

But w and $\mathbf{c}$ should be coupled. We would like $w_{i,j}$ to be large when $\mathbf{x}_i$ is close to $\mathbf{c}_j$, and small otherwise. Write $d_{i,j}$ for the distance $\|\mathbf{x}_i - \mathbf{c}_j\|$, choose a scaling parameter $\sigma > 0$, and write

$$s_{i,j} = e^{\frac{-d_{i,j}^2}{2\sigma^2}}.$$

This $s_{i,j}$ is often called the **affinity** between the point i and the center j; it is large when they are close in σ units, and small when they are far apart. Now a natural choice of weights is

$$w_{i,j} = \frac{s_{i,j}}{\sum_{l=1}^{k} s_{i,l}}.$$

All these weights are non-negative, they sum to one. The weight linking a point and a cluster center is large if the point is much closer to one center than to any other. The scaling parameter σ sets the meaning of "much closer"—we measure distance in units of σ.

Once we have weights, re-estimating the cluster centers is easy. We use the weights to compute a weighted average of the points. In particular, we re-estimate the j'th cluster center by

$$\frac{\sum_i w_{i,j}\mathbf{x}_i}{\sum_i w_{i,j}}.$$

Notice that k-means is a special case of this algorithm where σ limits to zero. In this case, each point has a weight of one for some cluster, and zero for all others, and the weighted mean becomes an ordinary mean. I have collected the description into a box (Procedure 12.4) for convenience.

Notice one other feature of this procedure. As long as you use sufficient precision for the arithmetic (which might be a problem), $w_{i,j}$ is *always* greater than zero. This means that no cluster is empty. In practice, if σ is small compared to the distances between points, you can end up with empty clusters. You can tell if this is happening by looking at $\sum_i w_{i,j}$; if this is very small or zero, you have a problem.

Procedure 12.4 (K-Means with Soft Weights) Choose k. Choose k data points $\mathbf{c}_j$ to act as initial cluster centers. Until the cluster centers change very little:

- First, we estimate the weights. For each pair of a data point $\mathbf{x}_i$ and a cluster $\mathbf{c}_j$, compute the affinity

$$s_{i,j} = e^{\frac{-\|\mathbf{x}_i - \mathbf{c}_j\|}{2\sigma^2}}.$$

- Now for each pair of a data point $\mathbf{x}_i$ and a cluster $\mathbf{c}_j$ compute the soft weight linking the data point to the center

$$w_{i,j} = s_{i,j} / \sum_{l=1}^{k} s_{i,l}.$$

- For each cluster, compute $\sum_i w_{i,j}$. If this is too small, then this cluster's new center is a point chosen at random from points far from their cluster center. Otherwise, the new center is

$$\mathbf{c}_j = \frac{\sum_i w_{i,j}\mathbf{x}_i}{\sum_i w_{i,j}}$$

12.3.3 Efficient Clustering and Hierarchical K Means

One important difficulty occurs in applications. We might need to have an enormous dataset (millions of items is a real possibility), and so a very large k. In this case, k-means clustering becomes difficult because identifying which cluster center is closest to a particular data point scales linearly with k (and we have to do this for every data point at every iteration). There are two useful strategies for dealing with this problem.

The first is to notice that, if we can be reasonably confident that each cluster contains many data points, some of the data is redundant. We could randomly subsample the data, cluster that, then keep the cluster centers. This works, but doesn't scale particularly well.

A more effective strategy is to build a hierarchy of k-means clusters. We randomly subsample the data (typically quite aggressively), then cluster this with a small value of k. Each data item is then allocated to the closest cluster center, and the data in each cluster is clustered again with k-means. We now have something that looks like a two-level tree of clusters. Of course, this process can be repeated to produce a multi-level tree of clusters.

12.3.4 K-Mediods

In some cases, we want to cluster objects that can't be averaged. One case where this happens is when you have a table of distances between objects, but do not know vectors representing the objects. For example, you could collect data giving the distances between cities, without knowing where the cities are (as in Sect. 10.4.3, particularly Fig. 10.16), then try and cluster using this data. As another example, you could collect data giving similarities between breakfast items as in Sect. 10.4.3, then turn the similarities into distances by taking the negative logarithm. This gives a useable table of distances. You still can't average kippers with oatmeal, so you couldn't use k-means to cluster this data.

A variant of k-means, known as k-medoids, applies to this case. In k-medoids, the cluster centers are data items rather than averages, and so are called "mediods". The rest of the algorithm has a familiar form. We assume k, the number of cluster centers, is known. We initialize the cluster centers by choosing examples at random. We then iterate two procedures. In the first, we allocate each data point to the closest mediod. In the second, we choose the best medoid for each cluster by finding the data point that minimizes the sum of distances of points in the cluster to that medoid. This point can be found by simply searching all points.

12.3.5 Example: Groceries in Portugal

Clustering can be used to expose structure in datasets that isn't visible with simple tools. Here is an example. At http://archive.ics.uci.edu/ml/datasets/Wholesale+customers, you will find a dataset giving sums of money spent annually on different commodities by customers in Portugal. The commodities are divided into a set of categories (fresh; milk; grocery; frozen; detergents and paper; and delicatessen) relevant for the study. These customers are divided by channel (two channels, corresponding to different types of shop) and by region (three regions). You can think of the data as being divided into six groups (one for each pair of channel and region). There are 440 customer records, and there are many customers in each group. The data was provided by M. G. M. S. Cardoso.

Figure 12.7 shows a panel plot of the customer data; the data has been clustered, and I gave each of 10 clusters its own marker. You (or at least, I) can't see any evidence of the six groups here. This is due to the form of the visualization, rather than a true property of the data. People tend to like to live near people who are "like" them, so you could expect people in a region to be somewhat similar; you could reasonably expect differences between groups (regional preferences; differences in wealth; and so on). Retailers have different channels to appeal to different people, so you could expect people using different channels to be different. But you don't see this in the plot of clusters. In fact, the plot doesn't really show much structure at all, and is basically unhelpful.

Here is a way to think about structure in the data. There are likely to be different "types" of customer. For example, customers who prepare food at home might spend more money on fresh or on grocery, and those who mainly buy prepared food might spend more money on delicatessen; similarly, coffee drinkers with cats or with children might spend more on milk than the lactose-intolerant, and so on. So we can expect customers to cluster in types. An effect like this is hard to see on a panel plot of the clustered data (Fig. 12.7). The plot for this dataset is hard to read, because the dimension is fairly high for a panel plot and the data is squashed together in the bottom left corner. However, you can see the effect when you cluster the data and look at the cost function in representing the data with different values of k—quite a small set of clusters gives quite a good representation of the customers (Fig. 12.8). The panel plot of cluster membership (also in that figure) isn't particularly informative. The dimension is quite high, and clusters get squashed together.

There is an important effect which isn't apparent in the panel plots. Some of what cause customers to cluster in types are driven by things like wealth and the tendency of people to have neighbors who are similar to them. This means that different groups should have different fractions of each type of customer. There might be more deli-spenders in wealthier regions; more milk-spenders and detergent-spenders in regions where it is customary to have many children; and so on. This sort of structure will not be apparent in a panel plot. A group of a few milk-spenders and many detergent-spenders will have a few data points with high milk expenditure values (and low other values) and also many data points with high detergent expenditure values (and low other values). In a panel plot, this will look like two blobs; but if there is a second group with many milk-spenders and few detergent-spenders will also look like two blobs, lying roughly on top of the first set of blobs. It will be hard to spot the difference between the groups.

An easy way to see this difference is to look at histograms of the types of customer *within each group*. I described each group of data by the histogram of customer types that appeared in that group (Fig. 12.9). Notice how the distinction between the groups is now apparent—the groups do appear to contain quite different distributions of customer type. It looks as though

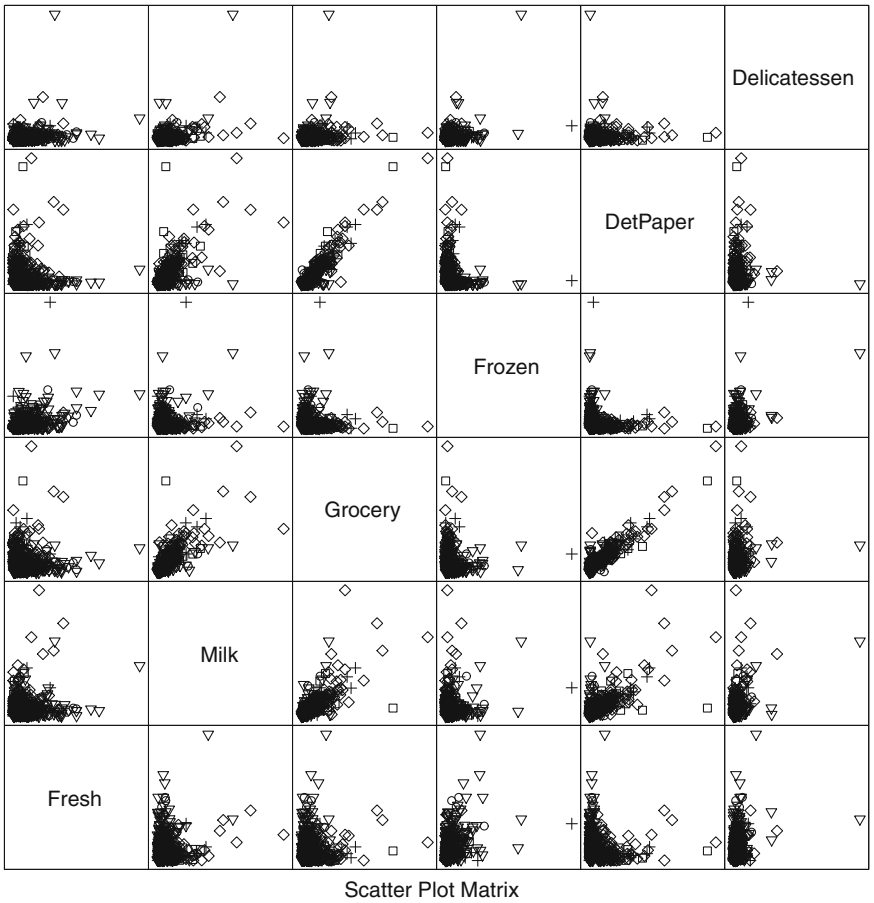

Fig. 12.7 A panel plot of the wholesale customer data of http://archive.ics.uci.edu/ml/datasets/Wholesale+customers, which records sums of money spent annually on different commodities by customers in Portugal. This data is recorded for six different groups (two channels each within three regions). I have plotted each group with a different marker, but you can't really see much structure here, for reasons explained in the text

the channels (rows in this figure) are more different than the regions (columns in this figure). To be more confident in this analysis, we would need to be sure that different types of customer really are different. We could do this by repeating the analysis for fewer clusters, or by looking at the similarity of customer types.

12.3.6 General Comments on K-Means

If you experiment with k-means, you will notice one irritating habit of the algorithm. It almost always produces either some rather spread out clusters, or some single element clusters. Most clusters are usually rather tight and blobby clusters, but there is usually one or more bad cluster. This is fairly easily explained. Because every data point must belong to some cluster, data points that are far from all others (a) belong to some cluster and (b) very likely "drag" the cluster center into a poor location. This applies even if you use soft assignment, because every point must have total weight one. If the point is far from all others, then it will be assigned to the closest with a weight very close to one, and so may drag it into a poor location, or it will be in a cluster on its own.

There are ways to deal with this. If k is very big, the problem is often not significant, because then you simply have many single element clusters that you can ignore. It isn't always a good idea to have too large a k, because then some larger clusters might break up. An alternative is to have a junk cluster. Any point that is too far from the closest true cluster center is assigned to the junk cluster, and the center of the junk cluster is not estimated. Notice that points should not be assigned to the junk cluster permanently; they should be able to move in and out of the junk cluster as the cluster centers move.

Remember this: *K-means clustering is the "go-to" clustering algorithm. You should see it as a basic recipe from which many algorithms can be concocted. The recipe is: iterate: allocate each data point to the closest cluster center; re-estimate cluster centers from their data points. There are many variations, improvements, etc. that are possible on*

(continued)

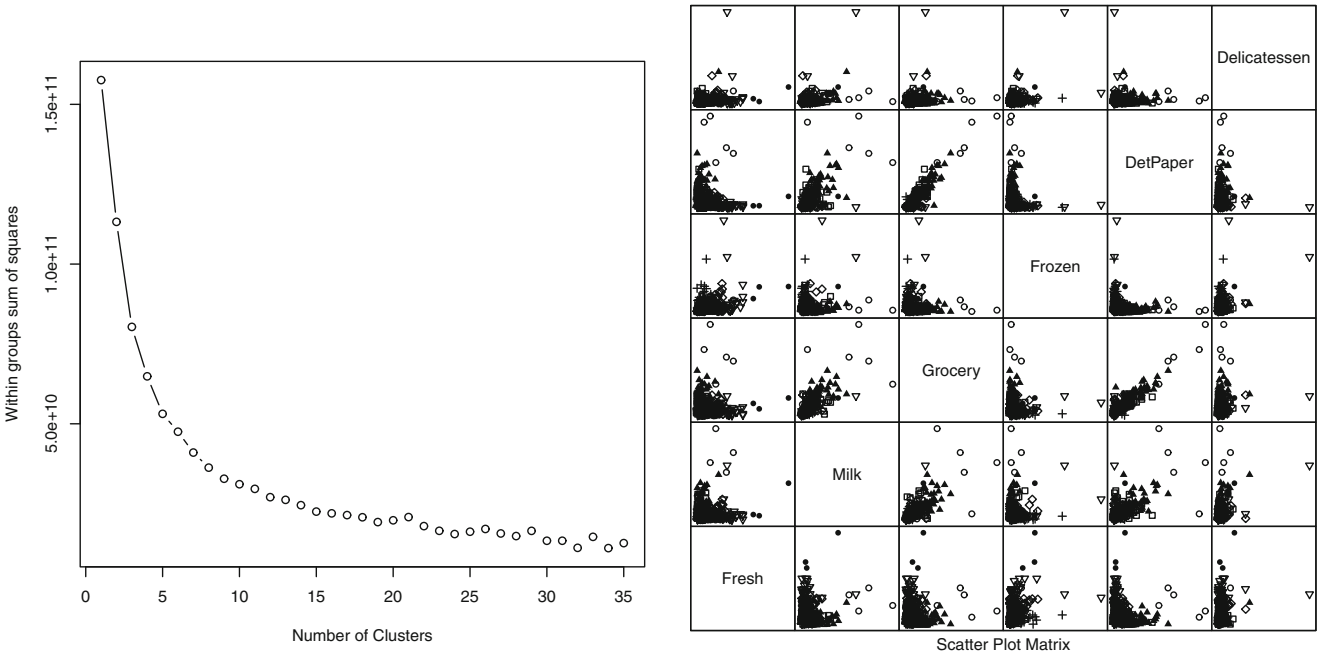

Fig. 12.8 On the *left*, the cost function (of Sect. 12.3) for clusterings of the customer data with k-means for k running from 2 to 35. This suggests using a k somewhere in the range 10–30; I chose 10. On the *right*, I have clustered this data to 10 cluster centers with k-means. The clusters do seem to be squashed together, but the plot on the left suggests that clusters do capture some important information. Using too few clusters will clearly lead to problems. Notice that I did not scale the data, because each of the measurements is in a comparable unit. For example, it wouldn't make sense to scale expenditures on fresh and expenditures on grocery with a different scale

this recipe. We have seen soft weights and k-mediods. K-means is not usually best implemented with the method I described (which isn't particularly efficient, but gets to the heart of what is going on). Implementations of k-means differ in important ways from my rather high-level description of the algorithm; for any but tiny problems, you should use a package, and you should look for a package that uses the Lloyd-Hartigan method.

12.4 Describing Repetition with Vector Quantization

The classifiers in Chap. 11 can be applied to simple images (the MNIST exercises at the end of the chapter, for example), but they will annoy you if you try to apply them as described to more complicated signals. All the methods described apply to feature vectors of fixed length. But typical of signals like speech, images, video, or accelerometer outputs is that different versions of the same thing have different lengths. For example, pictures appear at different resolutions, and it seems clumsy to insist that every image be 28×28 before it can be classified. As another example, some speakers are slow, and others are fast, but it's hard to see much future for a speech understanding system that insisted that everyone speak at the same speed so the classifier could operate. We need a construction that will take a signal and produce a useful feature vector of fixed length. This section shows one of the most useful such constructions (but be aware, this is an enormous topic).

Repetition is an important feature of many interesting signals. For example, images contain *textures*, which are orderly patterns that look like large numbers of small structures that are repeated. Examples include the spots of animals such as leopards or cheetahs; the stripes of animals such as tigers or zebras; the patterns on bark, wood, and skin. Similarly, speech signals contain *phonemes*—characteristic, stylised sounds that people assemble together to produce speech (for example, the "ka" sound followed by the "tuh" sound leading to "cat"). Another example comes from accelerometers. If a subject wears an accelerometer while moving around, the signals record the accelerations during their movements. So, for example, brushing one's teeth involves a lot of repeated twisting movements at the wrist, and walking involves swinging the hand back and forth.

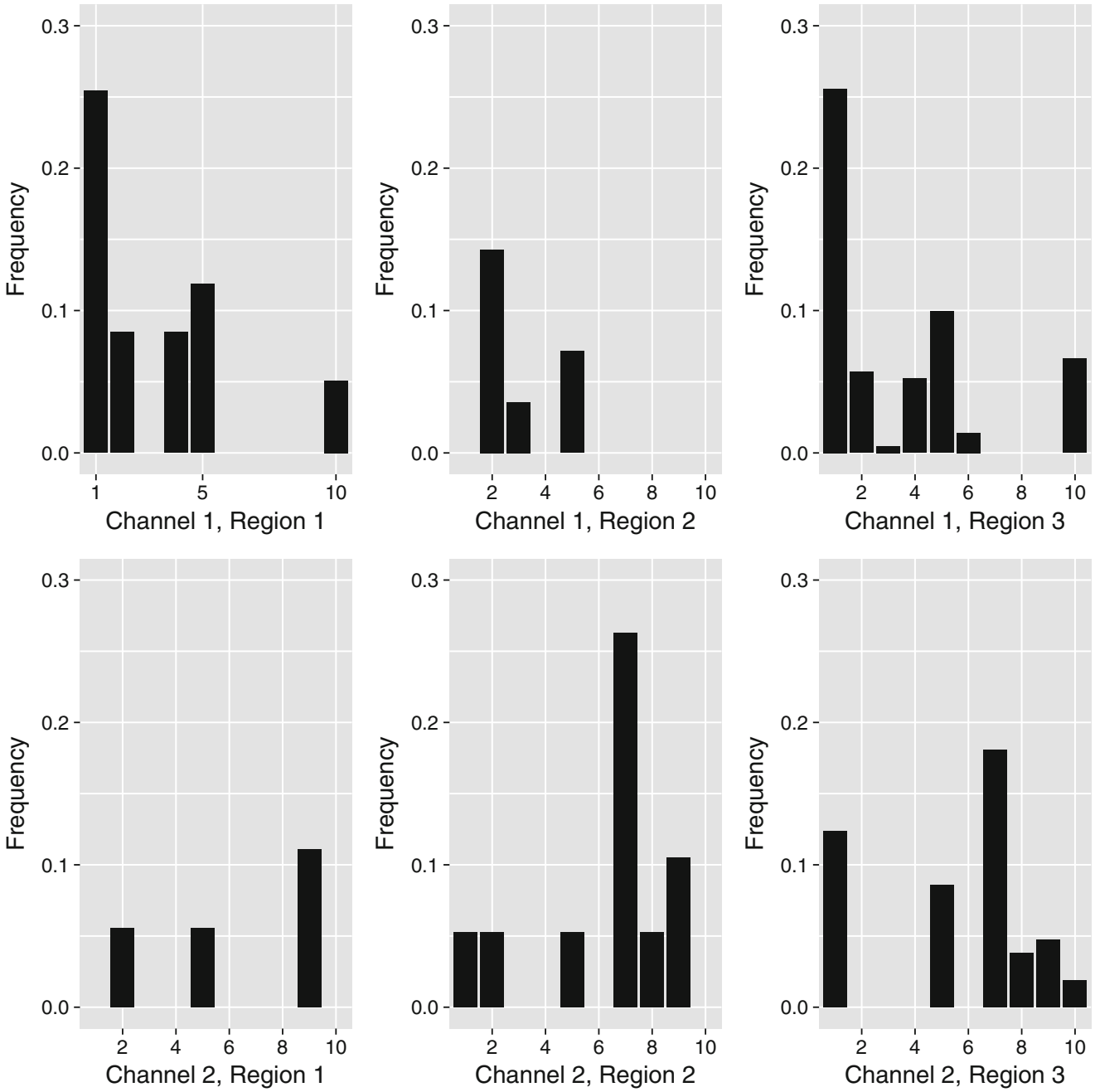

Fig. 12.9 The histogram of different types of customer, by group, for the customer data. Notice how the distinction between the groups is now apparent—the groups do appear to contain quite different distributions of customer type. It looks as though the channels (rows in this figure) are more different than the regions (columns in this figure)

Repetition occurs in subtle forms. The essence is that a small number of local patterns can be used to represent a large number of examples. You see this effect in pictures of scenes. If you collect many pictures of, say, a beach scene, you will expect most to contain some waves, some sky, and some sand. The individual patches of wave, sky or sand can be surprisingly similar. However, it's fair to model this by saying different images are made by selecting some patches from a vocabulary of patches, then placing them down to form an image. Similarly, pictures of living rooms contain chair patches, TV patches, and carpet patches. Many different living rooms can be made from small vocabularies of patches; but you won't often see wave patches in living rooms, or carpet patches in beach scenes. This suggests that the patches that are used to make an image reveal something about what is in the image. This observation works for speech, for video, and for accelerometer signals too.

An important part of representing signals that repeat is building a vocabulary of patterns that repeat, then describing the signal in terms of those patterns. For many problems, knowing what vocabulary elements appear and how often is much more important than knowing where they appear. For example, if you want to tell the difference between zebras and leopards, you need to know whether stripes or spots are more common, but you don't particularly need to know where they appear. As another example, if you want to tell the difference between brushing teeth and walking using accelerometer signals, knowing that there are lots of (or few) twisting movements is important, but knowing how the movements are linked together in time may not be. As a general rule, one can do quite a good job of classifying video just by knowing what patterns are there (i.e. without knowing where or when the patterns appear). However, this doesn't apply to speech, where it really matters what sound follows what sound.

12.4.1 Vector Quantization

It is natural to try and find patterns by looking for small pieces of signal of fixed size that appear often. In an image, a piece of signal might be a 10×10 patch, which can be reshaped into a vector. In a sound file, which is likely represented as a vector, it might be a subvector of fixed size. A 3-axis accelerometer signal is usually represented as a $3 \times r$ dimensional array (where r is the number of samples); in this case, a piece might be a 3×10 subarray, which can be reshaped into a vector. But finding patterns that appear often is hard to do, because the signal is continuous—each pattern will be slightly different, so we cannot simply count how many times a particular pattern occurs.

Here is a strategy. We take a training set of signals, and cut each signal into pieces of fixed size and reshape them into d dimensional vectors. We then build a set of clusters out of these pieces. This set of clusters is often thought of as a dictionary, because we expect many or most cluster centers to look like pieces that occur often in the signals and so are repeated.

We can now describe any new piece of signal with the cluster center closest to that piece. This means that a piece of signal is described with a number in the range $[1, \ldots, k]$ (where you get to choose k), and two pieces that are close should be described by the same number. This strategy is known as **vector quantization**.

This strategy applies to any kind of signal, and is surprisingly robust to details. We could use d dimensional vectors for a sound file; $\sqrt{d} \times \sqrt{d}$ dimensional patches for an image; or $3 \times (d/3)$ dimensional subarrays for an accelerometer signal. In each case, it is easy to compute the distance between two pieces using sum of squared distances. It seems not to matter much if the signals are cut into overlapping or non-overlapping pieces when forming the dictionary, as long as there are enough pieces.

Procedure 12.5 (Vector Quantization—Building a Dictionary) Take a training set of signals, and cut each signal into pieces of fixed size. The size of the piece will affect how well your method works, and is usually chosen by experiment. It does not seem to matter much if the pieces overlap. Cluster all the example pieces, and record the k cluster centers. It is usual, but not required, to use k-means clustering.

We can now build features that represent important repeated structure in signals. We take a signal, and cut it up into vectors of length d. These might overlap, or be disjoint. We then take each vector, and compute the number that describes it (i.e. the number of the closest cluster center, as above). We then compute a histogram of the numbers we obtained for all the vectors in the signal. This histogram describes the signal.

Procedure 12.6 (Vector Quantization—Representing a Signal) Take your signal, and cut it into pieces of fixed size. The size of the piece will affect how well your method works, and is usually chosen by experiment. It does not seem to matter much if the pieces overlap. For each piece, record the closest cluster center in the dictionary. Represent the signal with a histogram of these numbers, which will be a k dimensional vector.

Notice several nice features to this construction. First, it can be applied to anything that can be thought of in terms of fixed size pieces, so it will work for speech signals, sound signals, accelerometer signals, images, and so on. Another nice feature is the construction can accept signals of different length, and produce a description of fixed length. One accelerometer signal

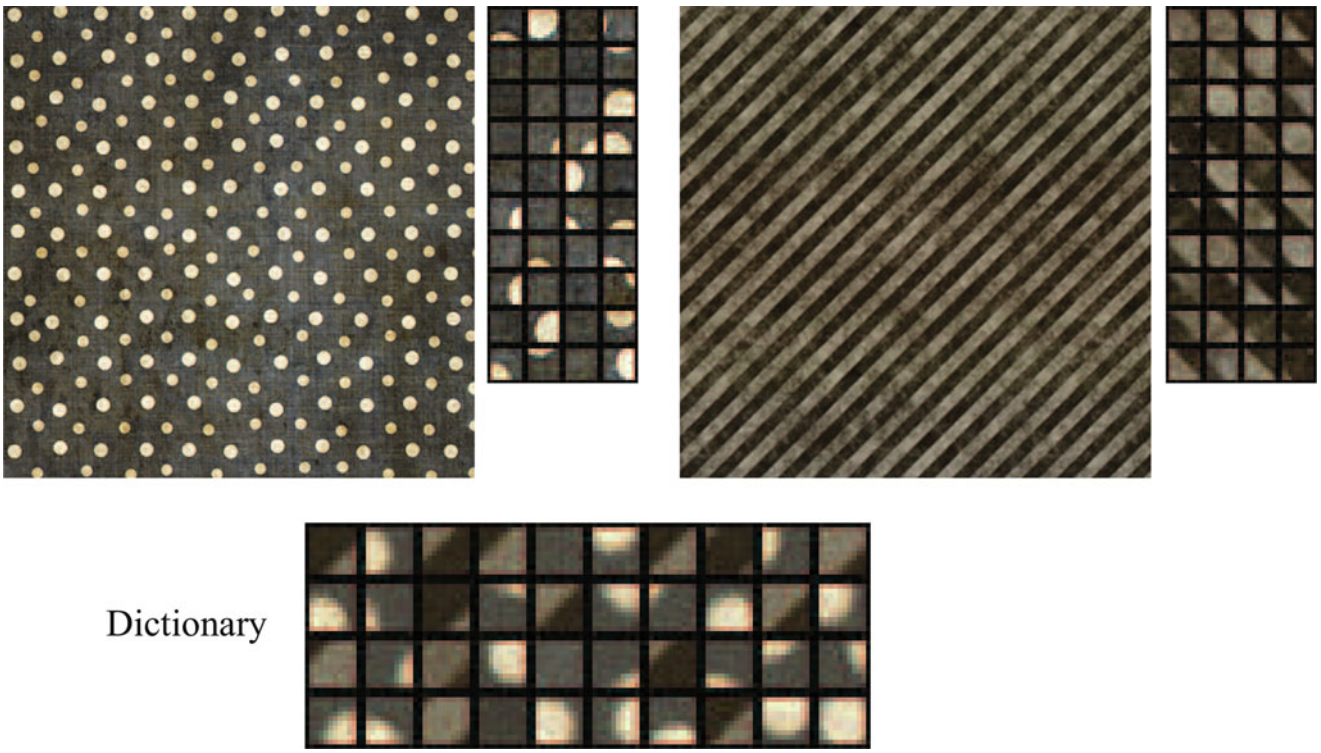

Fig. 12.10 *Top:* two images with rather exaggerated repetition, published on flickr.com with a creative commons license by `webtreats`. Next to these images, I have placed zoomed sampled 10×10 patches from those images; although the spots (resp. stripes) aren't necessarily centered in the patches, it's pretty clear which image each patch comes from. *Bottom:* a 40 patch dictionary computed using k-means from 4000 samples from each image. If you look closely, you'll see that some dictionary entries are clearly stripe entries, others clearly spot entries. Stripe images will have patches represented by stripe entries in the dictionary and spot images by spot entries

might cover 100 time intervals; another might cover 200; but the description is always a histogram with k buckets, so it's always a vector of length k.

Yet another nice feature is that we don't need to be all that careful how we cut the signal into fixed length vectors. This is because it is hard to hide repetition. This point is easier to make with a figure than in text, so look at Fig. 12.10.

The number of pieces of signal (and so k), might be very big indeed. It is quite reasonable to want to build a dictionary for a million items and use tens to hundreds of thousands of cluster centers. In this case, it is a good idea to use hierarchical k-means, as in Sect. 12.3.3. Hierarchical k-means produces a tree of cluster centers. It is easy to use this tree to vector quantize a query data item. We vector quantize at the first level. Doing so chooses a branch of the tree, and we pass the data item to this branch. It is either a leaf, in which case we report the number of the leaf, or it is a set of clusters, in which case we vector quantize, and pass the data item down. This procedure is efficient both when one clusters and at run time.

Representing a signal as a histogram of cluster centers loses information in two important ways. First, the histogram has little or no information about how the pieces of signal are arranged. So, for example, the representation can tell whether an image has stripy or spotty patches in it, but not where those patches lie. You should not rely on your intuition to tell you whether this lost information is important or not. For many kinds of image classification task, histograms of cluster centers are much better than you might guess, despite not encoding where patches lie (though still better results are now obtained with convolutional neural networks).

Second, replacing a piece of signal with a cluster center must lose some detail, which might be important, and likely results in some classification errors. There is a surprisingly simple construction that can alleviate these problems. Build three (or more) dictionaries, rather than one, using different sets of training pieces. For example, you could cut the same signals into pieces on a different grid. Now use each dictionary to produce a histogram of cluster centers, and classify with those. Finally, use a voting scheme to decide the class of each test signal. In many problems, this approach yields small but useful improvements.

12.4.2 Example: Activity from Accelerometer Data

A complex example dataset appears at https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer. This dataset consists of examples of the signal from a wrist mounted accelerometer, produced as different subjects engaged in different activities of daily life. Activities include: brushing teeth, climbing stairs, combing hair, descending stairs, and so on. Each is performed by sixteen volunteers. The accelerometer samples the data at 32 Hz (i.e. this data samples and reports the acceleration 32 times per second). The accelerations are in the x, y and z-directions. The dataset was collected by Barbara Bruno, Fulvio Mastrogiovanni and Antonio Sgorbissa. Figure 12.11 shows the x-component of various examples of toothbrushing.

There is an important problem with using data like this. Different subjects take quite different amounts of time to perform these activities. For example, some subjects might be more thorough tooth-brushers than other subjects. As another example, people with longer legs walk at somewhat different frequencies than people with shorter legs. This means that the same activity performed by different subjects will produce data vectors *that are of different lengths*. It's not a good idea to deal with this by warping time and resampling the signal. For example, doing so will make a thorough toothbrusher look as though they are moving their hands very fast (or a careless toothbrusher look ludicrously slow: think speeding up or slowing down a movie). So we need a representation that can cope with signals that are a bit longer or shorter than other signals.

Another important property of these signals is that all examples of a particular activity should contain repeated patterns. For example, brushing teeth should show fast accelerations up and down; walking should show a strong signal at somewhere around 2 Hz; and so on. These two points should suggest vector quantization to you. Representing the signal in terms of stylized, repeated structures is probably a good idea because the signals probably contain these structures. And if we represent the signal in terms of the relative frequency with which these structures occur, the representation will have a fixed length, even if the signal doesn't. To do so, we need to consider (a) over what time scale we will see these repeated structures and (b) how to ensure we segment the signal into pieces so that we see these structures.

Generally, repetition in activity signals is so obvious that we don't need to be smart about segment boundaries. I broke these signals into 32 sample segments, one following the other. Each segment represents 1 s of activity. This is long enough for the body to do something interesting, but not so long that our representation will suffer if we put the segment boundaries

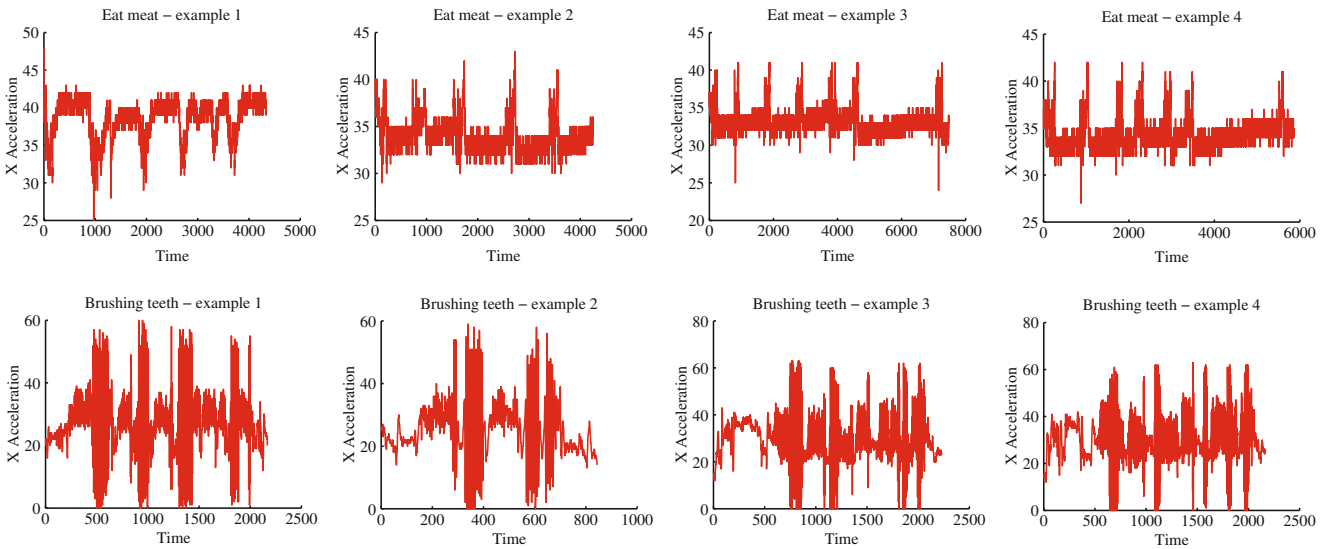

Fig. 12.11 Some examples from the accelerometer dataset at https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer. I have labelled each signal by the activity. These show acceleration in the X direction (Y and Z are in the dataset, too). There are four examples for *brushing teeth* and four for *eat meat*. You should notice that the examples don't have the same length in time (some are slower and some faster eaters, etc.), but that there seem to be characteristic features that are shared within a category (brushing teeth seems to involve faster movements than eating meet)

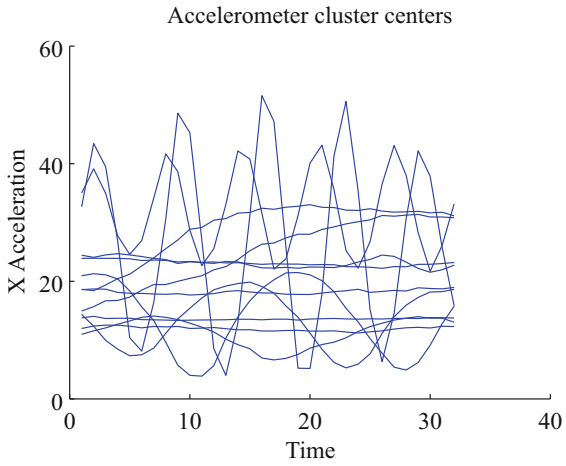

Fig. 12.12 Some cluster centers from the accelerometer dataset. Each cluster center represents a one-second burst of activity. There are a total of 480 in my model, which I built using hierarchical k-means. Notice there are a couple of centers that appear to represent movement at about 5 Hz; another few that represent movement at about 2 Hz; some that look like 0.5 Hz movement; and some that seem to represent much lower frequency movement. These cluster centers are samples (rather than chosen to have this property)

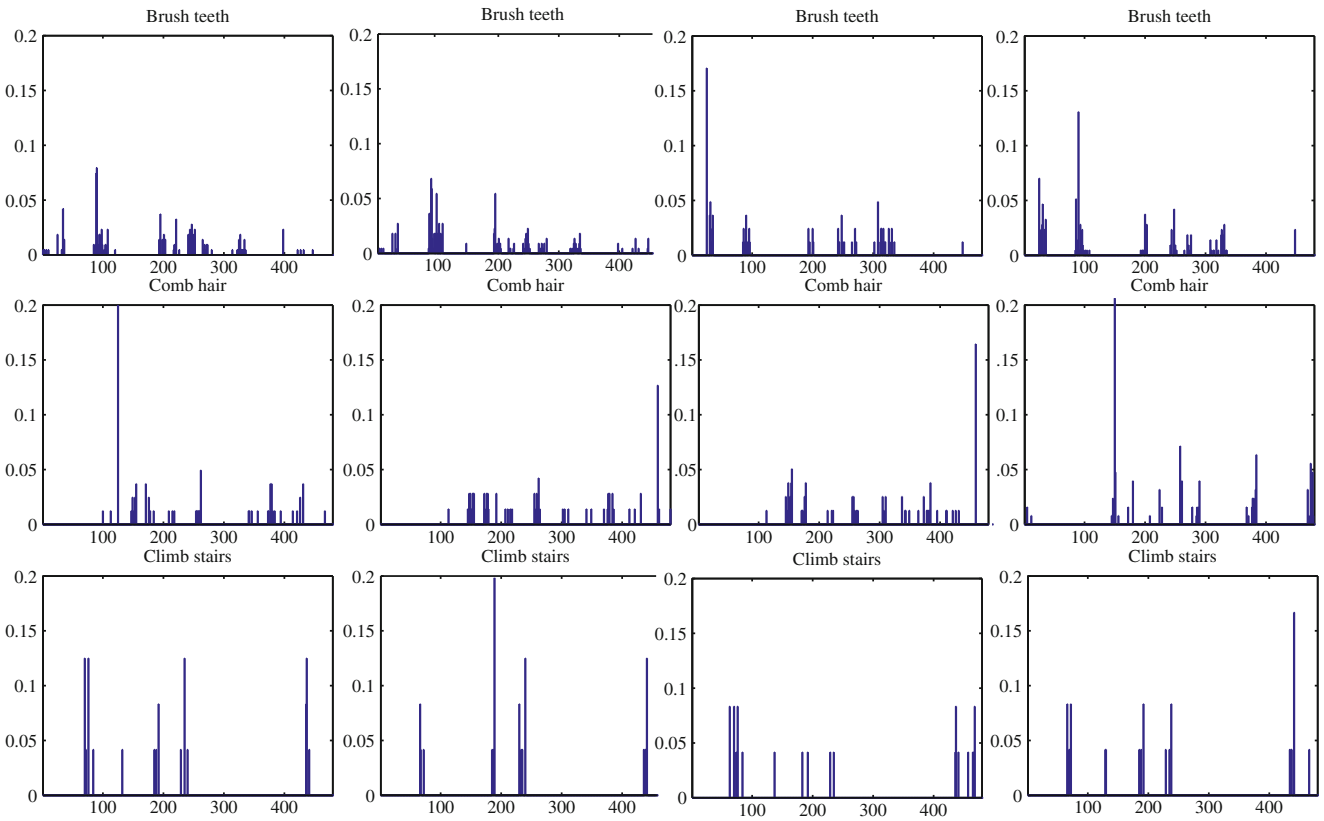

Fig. 12.13 Histograms of cluster centers for the accelerometer dataset, for different activities. You should notice that (a) these histograms look somewhat similar for different actors performing the same activity and (b) these histograms look somewhat different for different activities

in the wrong place. This resulted in about 40,000 segments. I then used hierarchical k-means to cluster these segments. I used two levels, with 40 cluster centers at the first level, and 12 at the second. Figure 12.12 shows some cluster centers at the second level.

I then computed histogram representations for different example signals (Fig. 12.13). You should notice that when the activity label is different, the histogram looks different, too.

Another useful way to check this representation is to compare the average within class chi-squared distance with the average between class chi-squared distance. I computed the histogram for each example. Then, for each pair of examples, I

0.9	2.0	1.9	2.0	2.0	2.0	1.9	2.0	1.9	1.9	2.0	2.0	2.0	2.0
	1.6	2.0	1.8	2.0	2.0	2.0	1.9	1.9	2.0	1.9	1.9	2.0	1.7
		1.5	2.0	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	1.9	2.0
			1.4	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	1.8
				1.5	1.8	1.7	1.9	1.9	1.8	1.9	1.9	1.8	2.0
					0.9	1.7	1.9	1.9	1.8	1.9	1.9	1.9	2.0
						0.3	1.9	1.9	1.5	1.9	1.9	1.9	2.0
							1.8	1.8	1.9	1.9	1.9	1.9	1.9
								1.7	1.9	1.9	1.9	1.9	1.9
									1.6	1.9	1.9	1.9	2.0
										1.8	1.9	1.9	1.9
											1.8	2.0	1.9
												1.5	2.0
													1.5

Table 12.1 Each column of the table represents an activity for the activity dataset https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer, as does each row

In each of the upper diagonal cells, I have placed the average chi-squared distance between histograms of examples from that pair of classes (I dropped the lower diagonal for clarity). Notice that in general the diagonal terms (average within class distance) are rather smaller than the off diagonal terms. This quite strongly suggests we can use these histograms to classify examples successfully

computed the chi-squared distance between the pair. Finally, for each pair of *activity labels*, I computed the average distance between pairs of examples where one example has one of the activity labels and the other example has the other activity label. In the ideal case, all the examples with the same label would be very close to one another, and all examples with different labels would be rather different. Table 12.1 shows what happens with the real data. You should notice that for some pairs of activity label, the mean distance between examples is smaller than one would hope for (perhaps some pairs of examples are quite close?). But generally, examples of activities with different labels tend to be further apart than examples of activities with the same label.

Yet another way to check the representation is to try classification with nearest neighbors, using the chi-squared distance to compute distances. I split the dataset into 80 test pairs and 360 training pairs; using 1-nearest neighbors, I was able to get a held-out error rate of 0.79. This suggests that the representation is fairly good at exposing what is important.

12.5 The Multivariate Normal Distribution

All the nasty facts about high dimensional data, above, suggest that we need to use quite simple probability models. By far the most important model is the multivariate normal distribution, which is quite often known as the multivariate gaussian distribution. We will not use a multivariate normal distribution explicitly in what follows, though if you look hard enough you might see it lurking beneath the surface in a couple of places. However, it's useful to have seen the basics, which you are quite likely to bump into elsewhere.

There are two sets of parameters in this model, the mean μ and the covariance Σ. For a d-dimensional model, the mean is a d-dimensional column vector and the covariance is a $d \times d$ dimensional matrix. The covariance is a symmetric matrix. For our definitions to be meaningful, the covariance matrix must be positive definite.

The form of the distribution $p(\mathbf{x}|\mu, \Sigma)$ is

$$p(\mathbf{x}|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma)}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1}(\mathbf{x}-\mu)\right).$$

The following facts explain the names of the parameters:

Useful Facts 12.1 (Parameters of a Multivariate Normal Distribution)
Assuming a multivariate normal distribution, we have

- $\mathbb{E}[\mathbf{x}] = \mu$, meaning that the mean of the distribution is μ.
- $\mathbb{E}\left[(\mathbf{x} - \mu)(\mathbf{x} - \mu)^T\right] = \Sigma$, meaning that the entries in Σ represent covariances.

Assume I know have a dataset of items $\mathbf{x}_i$, where i runs from 1 to N, and we wish to model this data with a multivariate normal distribution. The maximum likelihood estimate of the mean, $\hat{\mu}$, is

$$\hat{\mu} = \frac{\sum_i \mathbf{x}_i}{N}$$

(which is quite easy to show). The maximum likelihood estimate of the covariance, $\hat{\Sigma}$, is

$$\hat{\Sigma} = \frac{\sum_i (\mathbf{x}_i - \hat{\mu})(\mathbf{x}_i - \hat{\mu})^T}{N}$$

(which is rather a nuisance to show, because you need to know how to differentiate a determinant). These facts mean that we already know most of what is interesting about multivariate normal distributions (or gaussians).

12.5.1 Affine Transformations and Gaussians

Gaussians behave very well under affine transformations. In fact, we've already worked out all the math. There is one caveat, which is worth mentioning. You can't build a multivariate gaussian distribution unless the covariance is positive definite, because the distribution doesn't normalize. Assume I have a dataset $\mathbf{x}_i$. The mean of the maximum likelihood gaussian model is $\mathsf{mean}\left(\{\mathbf{x}_i\}\right)$, and the covariance is $\mathsf{Covmat}\left(\{\mathbf{x}_i\}\right)$, as long as $\mathsf{Covmat}\left(\{\mathbf{x}_i\}\right)$ is positive definite. I can now transform the data with an affine transformation, to get $\mathbf{y}_i = \mathcal{A}\mathbf{x}_i + \mathbf{b}$. As long as $\mathsf{Covmat}\left(\{\mathbf{y}_i\}\right)$ is positive definite, we can fit a multivariate gaussian with maximum likelihood. The mean of the maximum likelihood gaussian model for the transformed dataset is $\mathsf{mean}\left(\{\mathbf{y}_i\}\right)$, and we've dealt with this; similarly, the covariance is $\mathsf{Covmat}\left(\{\mathbf{y}_i\}\right)$, and we've dealt with this, too.

A very important point follows in an obvious way. I can apply an affine transformation to any multivariate gaussian to obtain one with (a) zero mean and (b) independent components. In turn, this means that, *in the right coordinate system*, any gaussian is a product of zero mean one-dimensional normal distributions. This fact is quite useful. For example, it means that simulating multivariate normal distributions is quite straightforward—you could simulate a standard normal distribution for each component, then apply an affine transformation.

12.5.2 Plotting a 2D Gaussian: Covariance Ellipses

There are some useful tricks for plotting a 2D Gaussian, which are worth knowing both because they're useful, and they help to understand Gaussians. Assume we are working in 2D; we have a Gaussian with mean μ (which is a 2D vector), and covariance Σ (which is a 2×2 matrix). We could plot the collection of points $\mathbf{x}$ that has some fixed value of $p(\mathbf{x}|\mu, \Sigma)$. This set of points is given by:

$$\frac{1}{2}\left((\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu)\right) = c^2$$

where c is some constant. I will choose $c^2 = \frac{1}{2}$, because the choice doesn't matter, and this choice simplifies some algebra. You might recall that a set of points $\mathbf{x}$ that satisfies a quadratic like this is a conic section. Because Σ (and so Σ^{-1}) is positive definite, the curve is an ellipse. There is a useful relationship between the geometry of this ellipse and the Gaussian.

This ellipse—like all ellipses—has a major axis and a minor axis. These are at right angles, and meet at the center of the ellipse. We can determine the properties of the ellipse in terms of the Gaussian quite easily. The geometry of the ellipse isn't

affected by rotation or translation, so we will translate the ellipse so that $\mu = \mathbf{0}$ (i.e. the mean is at the origin) and rotate it so that Σ^{-1} is diagonal. Writing $\mathbf{x} = [x, y]$ we get that the set of points on the ellipse satisfies

$$\frac{1}{2}(\frac{1}{k_1^2}x^2 + \frac{1}{k_2^2}y^2) = \frac{1}{2}$$

where $\frac{1}{k_1^2}$ and $\frac{1}{k_2^2}$ are the diagonal elements of Σ^{-1}. We will assume that the ellipse has been rotated so that $k_1 < k_2$. The points $(k_1, 0)$ and $(-k_1, 0)$ lie on the ellipse, as do the points $(0, k_2)$ and $(0, -k_2)$. The major axis of the ellipse, in this coordinate system, is the x-axis, and the minor axis is the y-axis. In this coordinate system, x and y are independent. If you do a little algebra, you will see that the standard deviation of x is $\mathsf{abs}\,(k_1)$ and the standard deviation of y is $\mathsf{abs}\,(k_2)$. So the ellipse is longer in the direction of largest standard deviation and shorter in the direction of smallest standard deviation.

Now rotating the ellipse is means we will pre- and post-multiply the covariance matrix with some rotation matrix. Translating it will move the origin to the mean. As a result, the ellipse has its center at the mean, its major axis is in the direction of the eigenvector of the covariance with largest eigenvalue, and its minor axis is in the direction of the eigenvector with smallest eigenvalue. A plot of this ellipse, which can be coaxed out of most programming environments with relatively little effort, gives us a great deal of information about the underlying Gaussian. These ellipses are known as **covariance ellipses**.

Remember this: *The multivariate normal distribution has the form*

$$p(\mathbf{x}|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d det(\Sigma)}} \exp\left(-\frac{1}{2}(\mathbf{x} - \mu)^T \Sigma^{-1}(\mathbf{x} - \mu)\right).$$

Assume you wish to model a dataset $\{\mathbf{x}\}$ *with a multivariate normal distribution. This will work as long as* $\mathsf{Covmat}\,(\{\mathbf{x}\})$ *is positive definite. The maximum likelihood estimate of the mean is* $\mathsf{mean}\,(\{\mathbf{x}\})$. *The maximum likelihood estimate of the covariance* Σ *is* $\mathsf{Covmat}\,(\{\mathbf{x}\})$.

12.6 You Should

12.6.1 Remember These Definitions

12.6.2 Remember These Terms

clustering . . . 281
clusters . . . 283
cluster center . . . 283
single-link clustering . . . 283
complete-link clustering . . . 283
group average clustering . . . 283
dendrogram . . . 284
whitening . . . 286
k-means . . . 287
affinity . . . 290
vector quantization . . . 296
covariance ellipses . . . 302

12.6.3 Remember These Facts

Parameters of a Multivariate Normal Distribution 301

12.6.4 Use These Procedures

To cluster agglomeratively 283
To cluster divisively 284
To cluster with k-means 287
To cluster with k-means, soft weights 291
To build a dictionary for vector quantization 296
To represent a signal with vector quantization 296

Programming Exercises

12.13 You can find a dataset dealing with European employment in 1979 at http://dasl.datadesk.com/data/view/47. This dataset gives the percentage of people employed in each of a set of areas in 1979 for each of a set of European countries.

(a) Use an agglomerative clusterer to cluster this data. Produce a dendrogram of this data for each of single link, complete link, and group average clustering. You should label the countries on the axis. What structure in the data does each method expose? it's fine to look for code, rather than writing your own. **Hint:** I made plots I liked a lot using R's `hclust` clustering function, and then turning the result into a phylogenetic tree and using a fan plot, a trick I found on the web; try `plot(as.phylo(hclustre- sult),type="fan")`. You should see dendrograms that "make sense" (at least if you remember some European history), and have interesting differences.
(b) Using k-means, cluster this dataset. What is a good choice of k for this data and why?

12.14 Obtain the activities of daily life dataset from the UC Irvine machine learning website (https://archive.ics.uci.edu/ml/datasets/Dataset+for+ADL+Recognition+with+Wrist-worn+Accelerometer; data provided by Barbara Bruno, Fulvio Mastrogiovanni and Antonio Sgorbissa).

(a) Build a classifier that classifies sequences into one of the 14 activities provided. To make features, you should vector quantize, then use a histogram of cluster centers (as described in the subsection; this gives a pretty explicit set of steps to follow). You will find it helpful to use hierarchical k-means to vector quantize. You may use whatever multi-class classifier you wish, though I'd start with R's decision forest, because it's easy to use and effective. You should report (a) the total error rate and (b) the class confusion matrix of your classifier.
(b) Now see if you can improve your classifier by (a) modifying the number of cluster centers in your hierarchical k-means and (b) modifying the size of the fixed length samples that you use.

CIFAR-10 and Vector Quantization Exercises

The following exercises are elaborate, but rewarding. The CIFAR-10 dataset is a set of labelled images in 10 classes, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The dataset consists of 60,000 32×32 colour images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 test images. You can find this dataset at https://www.cs.toronto.edu/~kriz/cifar.html; at that site, you will find pointers to information about how well various methods work, etc., too. The creators ask that anyone using this dataset acknowledge the technical report "Learning Multiple Layers of Features from Tiny Images," by Alex Krizhevsky written in 2009. It is very widely used to check simple image classification methods.

12.15 We will start by visualizing CIFAR-10.

(a) For each category, compute the mean image and the first 20 principal components. Plot the error resulting from representing the images of each category using the first 20 principal components against the category.
(b) Compute the distances between mean images for each pair of classes. Use principal coordinate analysis to make a 2D map of the means of each categories. For this exercise, compute distances by thinking of the images as vectors.
(c) Here is another measure of the similarity of two classes. For class A and class B, define $E(A \rightarrow B)$ to be the average error obtained by representing all the images of class A using the mean of class A and the first 20 principal components of class B. Now define the similarity between classes to be $(1/2)(E(A \rightarrow B) + E(B \rightarrow A))$. Use principal coordinate analysis to make a 2D map of the classes. Compare this map to the map in the previous exercise—are they different? why?

12.16 We will build a simple baseline. Here is a simple feature construction (called "local PCA" in the MNIST exercises). First, compute the first d principal components for each image class separately. Now for any image, compute a $10d$ dimensional feature vector by, for each class, subtracting that class mean from the image, then projecting the image onto the d principal components for that class. Finally, stack all 10 d dimensional features you get. This measures how much the difference between the image and the class mean looks like the difference between images of that class and the class mean. Use the local PCA construction to form features for the CIFAR 10 dataset. Use a package to train a decision forest to classify these images, using this feature vector. Compare the performance of this baseline to: (a) baselines on https://www.cs.toronto.edu/~kriz/cifar.html; and (b) chance.

12.17 We will use the simplest vector quantization and compare to the baseline of the previous exercise. Construct a dictionary using N 8×8 patches selected at random locations from randomly selected training images. Use k cluster centers, and (hierarchical) k-means to build your dictionary. Now carve your image into 8×8 patches, overlapping by 2, and vector quantize these patches. This will produce a k-dimensional histogram representing the image.

(a) Use a package to train a decision forest to classify these images using the vector quantized feature, making a choice of N and k that seems reasonable to you. Evaluate the accuracy of this classifier on the test set.
(b) Investigate the effects of changing N and k on the accuracy of your classifier.

12.18 Can you improve your MNIST classifiers using vector quantization strategies, as above?