Contents

CHAPTER 1

Clustering

As we have seen, high-dimensional data comes with problems. Except in special cases, the only really reliable probability model is the Gaussian (or Gaussian blob, or blob). 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**. We've seen this rule of thumb in operation already, in the case of classification. We separated the two classes with a hyperplane. We didn't discuss surfaces with more complicated geometries — and they tend not to be used — precisely because they are less simple. We could even measure this "simplicity" somewhat formally. The right way to think about this situation is that the more flexible the boundary between the two classes, the more likely there is a wrong answer that fits the training data.

A blob is another good, simple model. Modelling data as a blob involves computing its mean and its covariance. Sometimes, as we shall see, the covariance can be hard to compute. Even so, a blob model is really useful. It is natural to try and extend this model to cover datasets that don't obviously consist of a single blob.

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 (a) what the blob parameters are and (b) which datapoints belong to which blob. Generally, we will collect together data points that are close and form blobs out of them. This process is known as **clustering**.

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 bad is that the model is bad.

1.1 AGGLOMERATIVE AND DIVISIVE CLUSTERING

There are two natural algorithms for clustering. In **divisive clustering**, the entire data set is regarded as a cluster, and then clusters are recursively split to yield a good clustering (Algorithm 1.2). In **agglomerative clustering**, each data item is regarded as a cluster, and clusters are recursively merged to yield a good clustering (Algorithm 1.1).

There are two major issues in thinking about clustering:

- *What is a good inter-cluster distance?* Agglomerative clustering uses an inter-cluster distance to fuse nearby clusters; divisive clustering uses it to split insufficiently coherent clusters. Even if a natural distance between data points is available (which might not be the case for vision problems), there is no

```
Make each point a separate cluster
Until the clustering is satisfactory
     Merge the two clusters with the
       smallest inter-cluster distance
end
```

Algorithm 1.1: *Agglomerative Clustering or Clustering by Merging.*

```
Construct a single cluster containing all points
Until the clustering is satisfactory
     Split the cluster that yields the two
       components with the largest inter-cluster distance
end
```

Algorithm 1.2: *Divisive Clustering, or Clustering by Splitting.*

canonical inter-cluster distance. Generally, one chooses a distance that seems appropriate for the data set. For example, one might choose the distance between the closest elements as the inter-cluster distance, which tends to yield extended clusters (statisticians call this method **single-link clustering**). Another natural choice is the maximum distance between an element of the first cluster and one of the second, which tends to yield rounded clusters (statisticians call this method **complete-link clustering**). Finally, one could use an average of distances between elements in the cluster, which also tends to yield "rounded" clusters (statisticians call this method **group average clustering**).

- *How many clusters are there?* This is an intrinsically difficult task if there is no model for the process that generated the clusters. The algorithms we have described generate 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 Figure 1.1).

The main difficulty in using a divisive model is knowing where to split. This is sometimes made easier for particular kinds of data. For example, we could segment an image by clustering pixel values. In this case, you can sometimes find good splits by constructing a histogram of intensities, or of color values.

Another important thing to notice about clustering from the example of figure 1.1 is that there is no right answer. There are a variety of different clusterings of the same data. For example, depending on what scales in that figure mean, it might be right to zoom out and regard all of the data as a single cluster, or to zoom in and regard each data point as a cluster. Each of these representations may be useful.

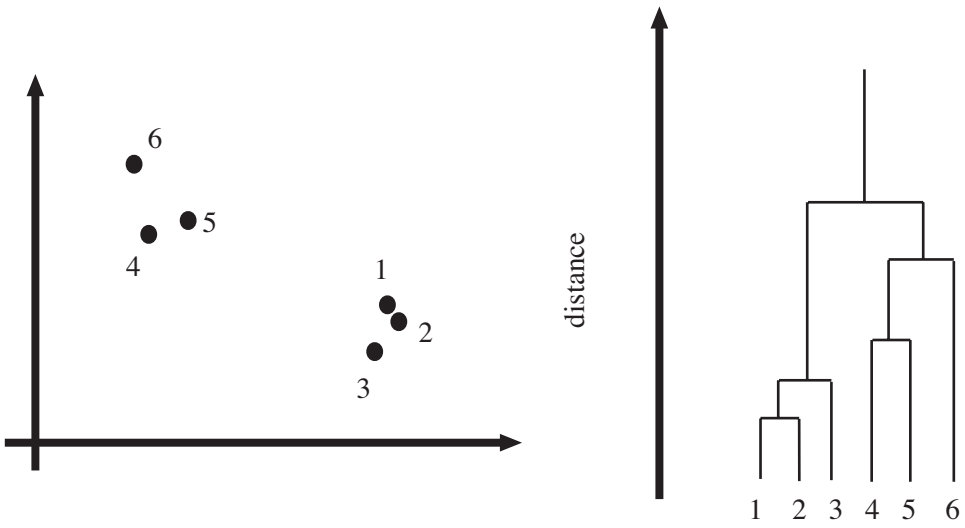

FIGURE 1.1: **Left**, *a data set;* **right**, *a dendrogram obtained by agglomerative clustering using single-link clustering. If one selects a particular value of distance, then a horizontal line at that distance splits the dendrogram into clusters. This representation makes it possible to guess how many clusters there are and to get some insight into how good the clusters are.*

1.1.1 Clustering and Distance

In the algorithms above, and in what follows, we assume that the features are scaled so that distances (measured in the usual way) between data points are 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. More sophisticated is to translate the data so that it has zero mean, then transform it so that each direction is independent and has unit variance. Doing so is sometimes referred to as **decorrelation** or **whitening**

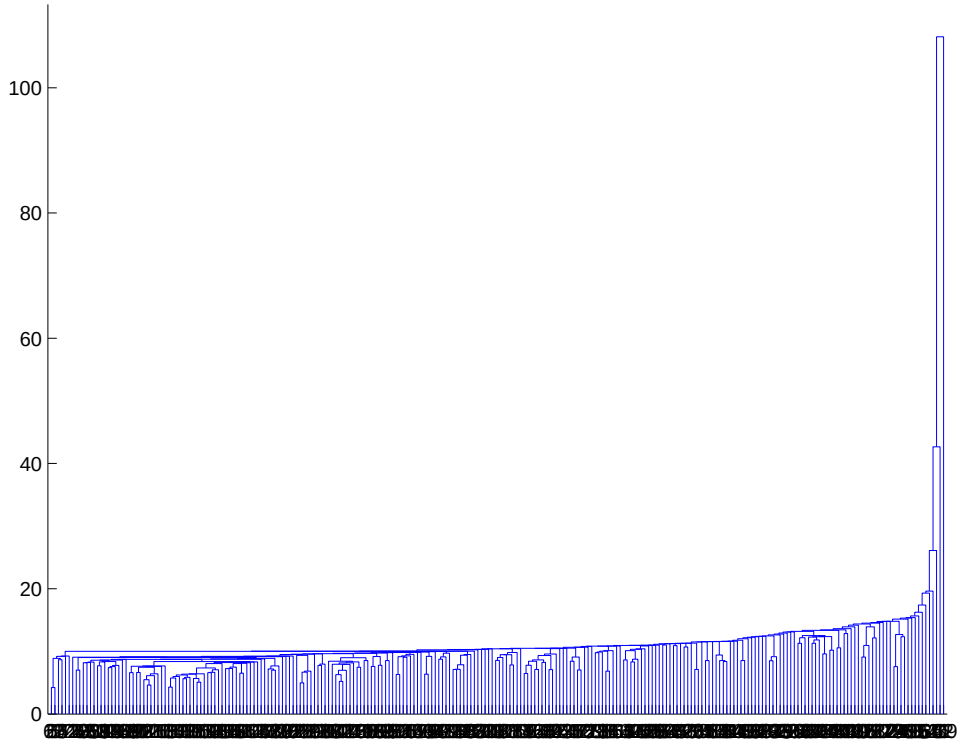

FIGURE 1.2: *A dendrogram obtained from the body fat dataset, using single link clustering. Recall that the data points are on the horizontal axis, and that the vertical axis is distance; there is a horizontal line linking two clusters that get merged, established at the height at which they're merged. I have plotted the entire dendrogram, despite the fact it's a bit crowded at the bottom, because it shows that most data points are relatively close (i.e. there are lots of horizontal branches at about the same height).*

(because you make the data more like white noise).

1.1.2 Example: Agglomerative Clustering

Matlab provides some tools that are useful for agglomerative clustering. These functions use a scheme where one first builds the whole tree of merges, then analyzes that tree to decide which clustering to report. `linkage` will determine which pairs of clusters should be merged at which step (there are arguments that allow you to choose what type of inter-cluster distance it should use); `dendrogram` will plot you a dendrogram; and `cluster` will extract the clusters from the linkage, using a variety of options for choosing the clusters. I used these functions to prepare the dendrogram of figure 1.2 for the height-weight dataset of section **??** (from `http://www2.stetson.edu/~jrasp/data.htm`; look for bodyfat.xls). I deliberately forced Matlab to plot the whole dendrogram, which accounts for the crowded look of the

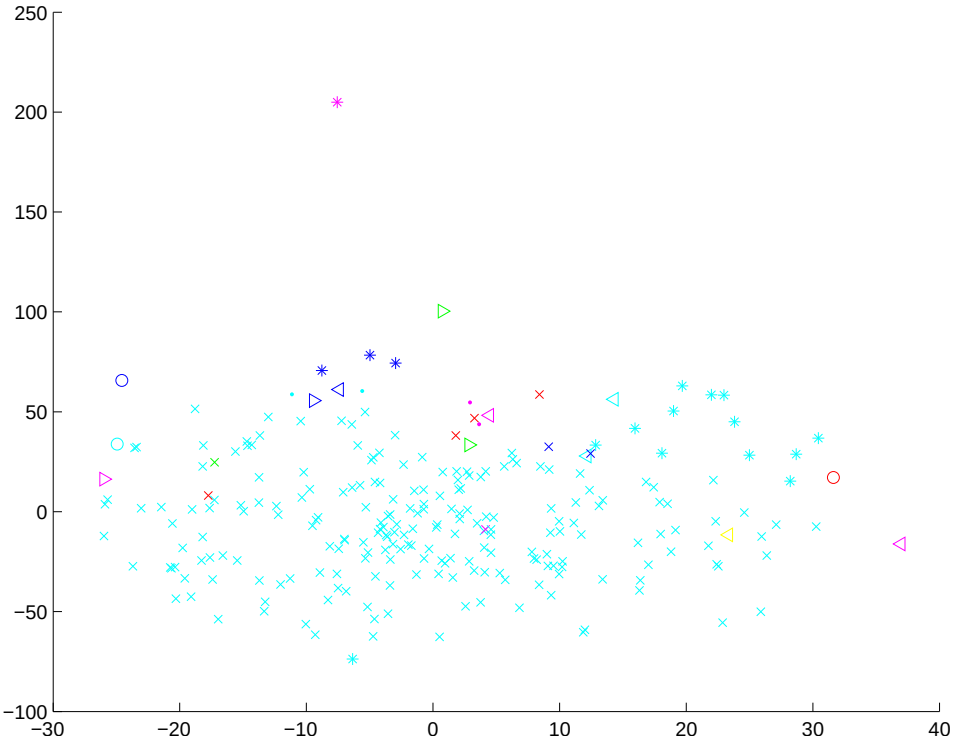

FIGURE 1.3: *A clustering of the body fat dataset, using agglomerative clustering, single link distance, and requiring a maximum of 30 clusters. I have plotted each cluster with a distinct marker (though some markers differ only by color; you might need to look at the PDF version to see this figure at its best). Notice that one cluster contains much of the data, and that there are a set of small isolated clusters. The original data is 16 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 16 dimensional space).*

figure (you can allow it to merge small leaves, etc.). I used a single-link strategy. In particular, notice that many data points are about the same distance from one another, which suggests a single big cluster with a smaller set of nearby clusters. The clustering of figure 1.3 supports this view. I plotted the data points on the first two principal components, using different colors and shapes of marker to indicate different clusters. There are a total of 30 clusters here, though most are small.

As another example, I obtained the seed dataset from the UC Irvine Machine Learning Dataset Repository (you can find it at `http://archive.ics.uci.edu/ml/datasets/seeds`). Each item consists of seven measurements of a wheat kernel; there are three types of wheat represented in this dataset. As you can see in figures 1.4 and 1.5, this data clusters rather well.

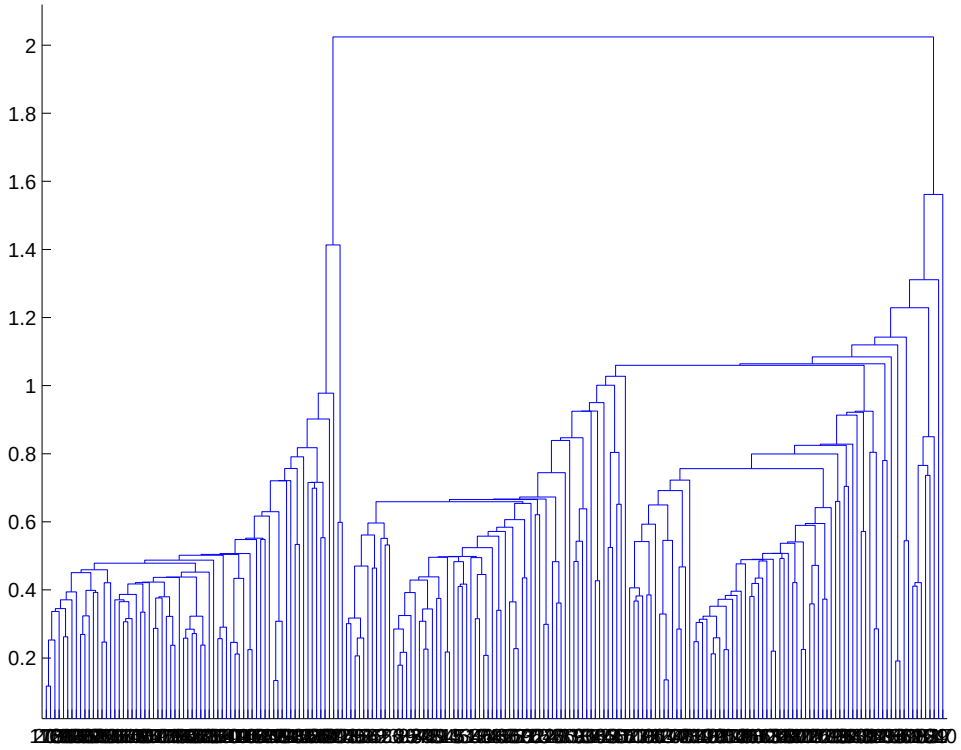

FIGURE 1.4: *A dendrogram obtained from the seed dataset, using single link clustering. Recall that the data points are on the horizontal axis, and that the vertical axis is distance; there is a horizontal line linking two clusters that get merged, established at the height at which they're merged. I 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".*

1.2 THE K-MEANS ALGORITHM

We could use any clustering method to vector quantize (Chapter **??** describes a number of different clustering methods in the context of segmentation). However, by far the most common method used is **k-means** clustering. Assume we have a set of data items that we wish to cluster. We now assume that we know how many clusters there are in the data, which we write k. This is equivalent to fixing the number of values we wish to quantize to. Each cluster is assumed to have a center; we write the center of the ith cluster as $\mathbf{c}_i$. The jth data item to be clustered is described by a feature vector $\mathbf{x}_j$. In our case, these items are vectors of filter responses observed at image locations.

Because pattern elements repeat, and so are common, we can assume that most data items are close to the center of their cluster. This suggests that we

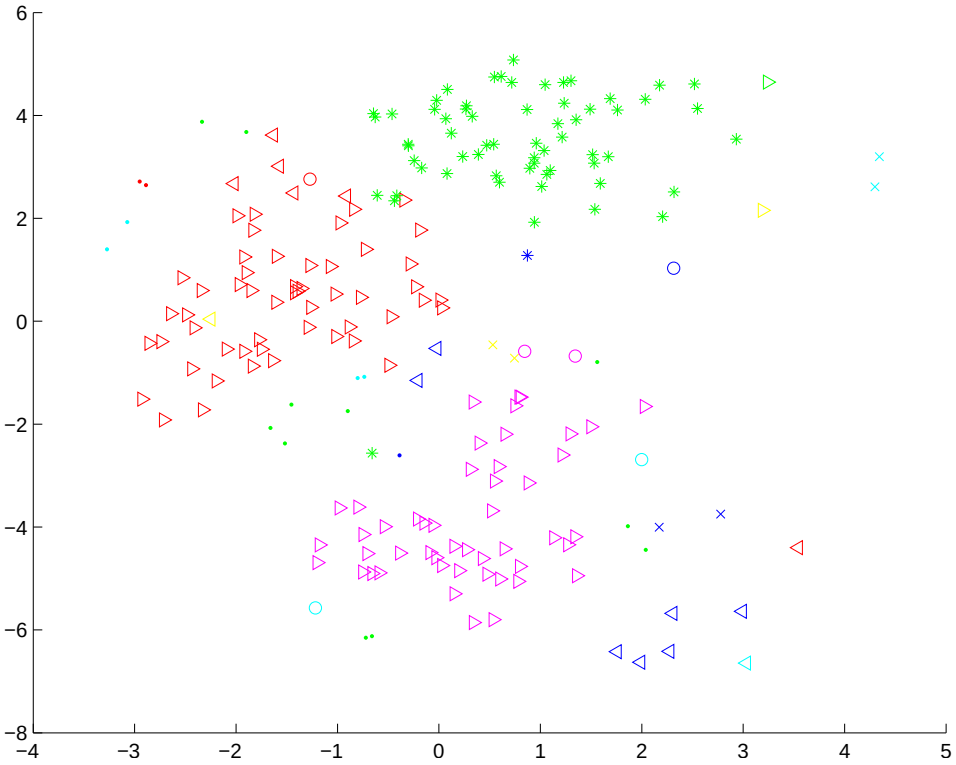

FIGURE 1.5: *A clustering of the seed dataset, using agglomerative clustering, single link distance, and requiring a maximum of 30 clusters. I have plotted each cluster with a distinct marker (though some markers differ only by color; you might need to look at the PDF version to see this figure at its best). Notice that there are a set of fairly natural isolated clusters. The original data is 8 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 8 dimensional space).*

cluster the data by minimizing the the objective function

$$\Phi(\text{clusters}, \text{data}) = \sum_{i \in \text{clusters}} \left\{ \sum_{j \in i\text{th cluster}} (\mathbf{x}_j - \mathbf{c}_i)^T (\mathbf{x}_j - \mathbf{c}_i) \right\}.$$

Notice that if we know the center for each cluster, it is easy to determine which cluster is the best choice for each point. Similarly, if the allocation of points to clusters is known, it is easy to compute the best center for each cluster. However, there are far too many possible allocations of points to clusters to search this space for a minimum. Instead, we define an algorithm that iterates through two activities:

- Assume the cluster centers are known and, allocate each point to the closest cluster center.

- Assume the allocation is known, and choose a new set of cluster centers. Each center is the mean of the points allocated to that cluster.

We then 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 1.3). It is possible to search for an appropriate number of clusters by applying k-means for different values of k and comparing the results; we defer a discussion of this issue until Section **??**.

```
Choose k data points to act as cluster centers
Until the cluster centers change very little
    Allocate each data point to 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.
end
```

Algorithm 1.3: *Clustering by K-Means.*

1.3 VECTOR QUANTIZATION AS A WAY TO BUILD FEATURES

We know how to classify data given feature vectors (or, at least, we should have read chapter **??**). But we don't have much in the way of techniques to construct good features. Clustering provides one of the most important techniques, because it gives us a way to detect and describe repeated patterns. These are important for pictures and for sound.

Texture is a phenomenon that is widespread, easy to recognise, and hard to define. Typically, whether an effect is referred to as texture or not depends on the scale at which it is viewed. A leaf that occupies most of an image is an object, but the foliage of a tree is a texture. Views of large numbers of small objects are often best thought of as textures. Examples include grass, foliage, brush, pebbles, and hair. Many surfaces are marked with orderly patterns that look like large numbers of small objects. 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. Textures tend to show **repetition**: (roughly!) the same local patch appears again and again, though it may be distorted by a viewing transformation.

Texture is important, because texture appears to be a very strong cue to object identity. Most modern object recognition programs are built around texture

representation machinery of one form or another. This may be because texture is also a strong cue to **material properties**: what the material that makes up an object is like. For example, texture cues can be used to tell tree bark (which is moderately hard and rough) from bare metal (which is hard, smooth, and shiny). People seem to be able to predict some mechanical properties of materials from their appearance. For example, often you can distinguish somewhat viscous materials, like hand cream, from highly viscous materials, like cream cheese, by eye (**?**). Material properties are correlated to the identity of objects, but they are not the same thing. For example, although hammers are commonly made of metal, a plastic hammer, a metal hammer, and a wooden hammer are all still hammers.

FIGURE 1.6: *Although texture is difficult to define, it has some important and valuable properties. In this image, there are many repeated elements (some leaves form repeated "spots"; others, and branches, form "bars" at various scales; and so on). Our perception of the material is quite intimately related to the texture (what would the surface feel like if you ran your fingers over it? what is soggy? what is prickly? what is smooth?). Notice how much information you are getting about the type of plants, their shape, the shape of free space, and so on, from the textures.* Geoff Brightling © Dorling Kindersley, used with permission.

But how should we represent textures? A texture is a set of textons — patches

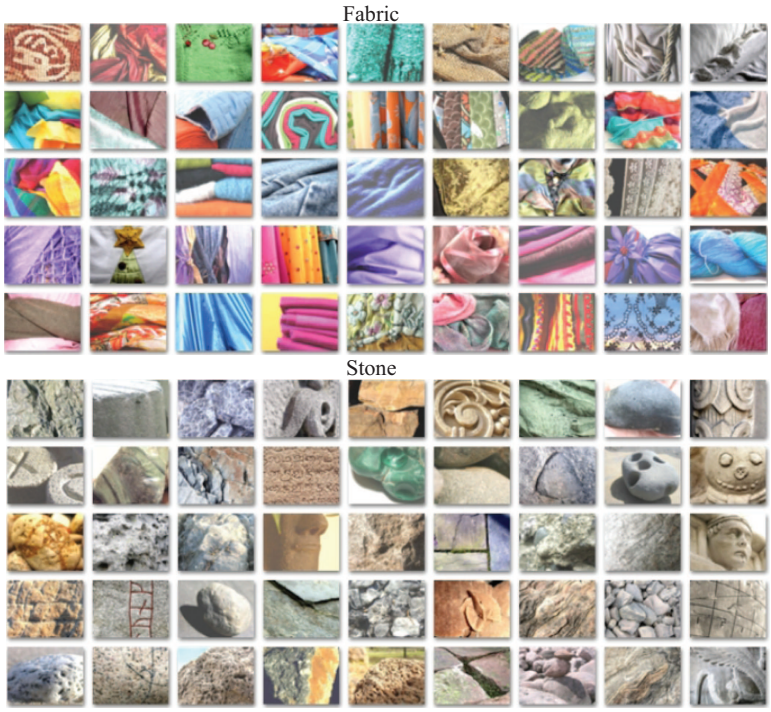

FIGURE 1.7: *Typically, different materials display different image textures. These are example images from a collection of 1,000 material images, described in by ?; there are 100 images in each of the ten categories, including the two categories shown here (fabric and stone). Notice how (a) the textures vary widely, even within a material category; and (b) different materials seem to display quite different textures.* This figure shows elements of a database collected by C. Liu, L. Sharan, E. Adelson, and R. Rosenholtz, and published at `http://people.csail.mit.edu/lavanya/research_sharan.html`. Figure by kind permission of the collectors.

of image — that repeat in some way. We could find these textons by looking for image patches that are common. There are two important difficulties in finding image patches that commonly occur together. First, these representations of the image are continuous. We cannot simply count how many times a particular pattern occurs, because each vector is slightly different. Second, the representation is high dimensional. A patch around a pixel might need hundreds of pixels to represent it well; similarly, hundreds of different filters might be needed to represent the image at a pixel. This means we cannot build a histogram directly, either, because it will have an unmanageable number of cells.

Similar issues occur with sound signals. Local (in time) patterns, like the phonemes of speech, are repeated. They are not repeated exactly, because they are continuous. Their repetition is characteristic of the type of sound, so, for example, one hears phonemes in speech but not in car crashes. But the signal has too high a dimension for us to simply build a histogram.

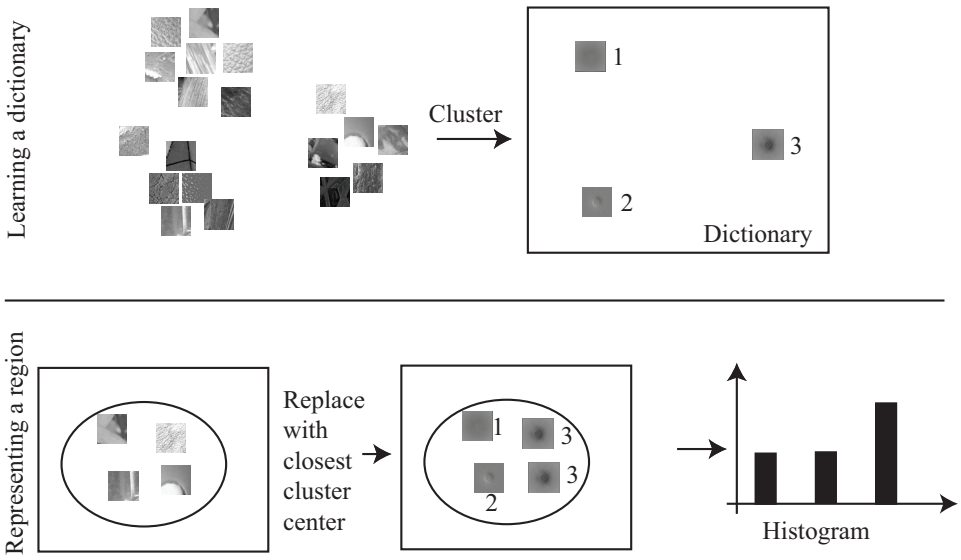

FIGURE 1.8: *There are two steps to building a pooled texture representation for a texture in an image domain. First, one builds a dictionary representing the range of possible pattern elements, using a large number of texture patches. This is usually done in advance, using a training data set of some form. Second, one takes the patches inside the domain, vector quantizes them by identifying the number of the closest cluster center, then computes a histogram of the different cluster center numbers that occur within a region. This histogram might appear to contain no spatial information, but this is a misperception. Some frequent elements in the histogram are likely to be textons, but others describe common ways in which textons lie close to one another; this is a rough spatial cue.* This figure shows elements of a database collected by C. Liu, L. Sharan, E. Adelson, and R. Rosenholtz, and published at `http://people.csail.mit.edu/lavanya/research_sharan.html`. Figure by kind permission of the collectors.

A good feature construction should: (a) identify things that repeat significantly often and (b) tell us which of those things appear in the particular signal we have.

Vector quantization is a strategy to deal with these difficulties. Vector quantization is a way of representing vectors in a continuous space with numbers from a set of fixed size. We first build a set of clusters out of a training set of vectors; this set of clusters is often thought of as a dictionary. We now replace any new vector with the cluster center closest to that vector. This strategy applies to vectors quite generally, though we will use it for texture representation. Many different clusterers can be used for vector quantization, but it is most common to use k-means or one of its variants.

We can now represent a collection of vectors as a histogram of cluster centers. This general recipe can be applied to texture representation by describing each pixel in the domain with some vector, then vector quantizing and describing the

domain with the histogram of cluster centers. A natural vector to use is obtained by reshaping the pixels from a fixed-size patch around the image pixel (Figure 1.9). In each case, we are building a representation in terms of commonly repeated pattern elements.

Build a dictionary:
- Collect many training example textures
- Construct the vectors $\mathbf{x}$ for relevant pixels; these could be
  - a reshaping of a patch around the pixel, a vector of filter outputs computed at the pixel, or the representation of Section **??**.
- Obtain k cluster centers $\mathbf{c}$ for these examples

Represent an image domain:
- For each relevant pixel i in the image
  - Compute the vector representation $\mathbf{x}_i$ of that pixel
  - Obtain j, the index of the cluster center $\mathbf{c}_j$ closest to that pixel
  - Insert j into a histogram for that domain

Algorithm 1.4: *Texture Representation Using Vector Quantization.*

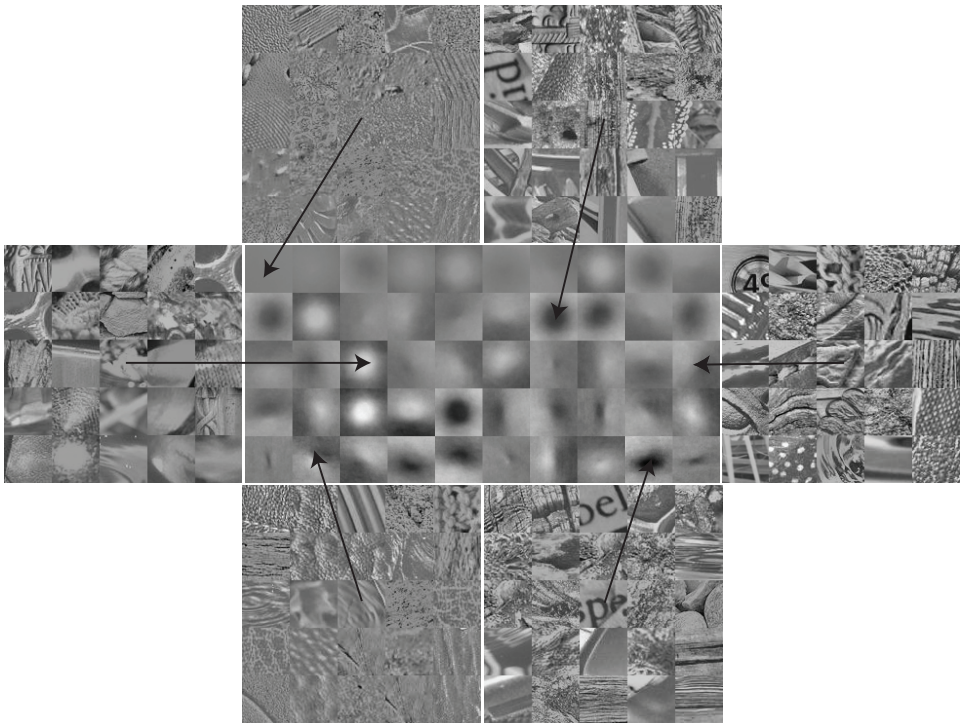

FIGURE 1.9: *Pattern elements can also be identified by vector quantizing vectors obtained by reshaping an image window centered on each pixel. Here we show the top 50 pattern elements (or textons), obtained using this strategy from all 1,000 images of the collection of material images described in Figure 1.7. Each subimage here illustrates a cluster center. For some cluster centers, we show the closest 25 image patches. To measure distance, we first subtracted the average image intensity, and we weighted by a Gaussian to ensure that pixels close to the center of the patch were weighted higher than those far from the center.* This figure shows elements of a database collected by C. Liu, L. Sharan, E. Adelson, and R. Rosenholtz, and published at `http://people.csail.mit.edu/lavanya/research_sharan.html`. Figure by kind permission of the collectors.