CHAPTER 9

# Image Segmentation with Clustering

*Segmentation* methods break images into groups of pixels, to obtain a more compact representation of what is interesting in the image. There is no correct segmentation of an image. Instead, the segmentation you want depends on what you are going to do with it. A *region* is a collection of pixels that belong together.

Image segmentation is usually an intermediate step in some other process. For example, you could cut out an object in an image, to replace it with something else or to put it into some other image. As another example, you could decompose an image into coherent chunks (*superpixels* if they are moderately sized; *regions* or *segments* if they are big) as part of compressing the image.

## 9.1 BACKGROUND SUBTRACTION AND SHOT BOUNDARY DETECTION

Two applications are often not recognized as examples of segmentation, because the methods are straightforward. Nonetheless, they are useful. In *background subtraction* , you aim to separate an object (*foreground*) from a largely irrelevant *background*. There are two regions here – foreground and background. The simplest case occurs when you see multiple frames, with a possibly moving object on a stable background. Typically, the foreground is passed on for further analysis and the background is ignored. In *shot boundary detection*, you break a video into *shots* — much shorter subsequences of frames that show largely the same objects (so the shot is the region here). It is helpful to represent a video as a collection of shots, where each shot is represented with a *key frame*. A key frame is a typical or representative frame. If the shot is coherent, this could even be chosen at random. A representation like this can be used to search for videos or to summarize videos to support browsing.

### 9.1.1 Background Subtraction

Background subtraction works as follows. Obtain an estimate of what the background looks like; optionally, obtain an estimate of what the foreground looks like. Now use this estimate to classify a pixel by determining whether it is more like the background or the foreground. Finally, keep the foreground pixels. There are a large number of variants of this recipe. Important sources of variation include: how to obtain the estimates of appearance; how to use these to classify pixels; and whether to use spatial models when classifying. Spatial models capture, for example, the tendency of a background pixel to have background neighbors. Such models can improve the foreground at the cost of increased complexity of classification.

One way to model the background is simply to take a picture. This approach works rather poorly because the background typically changes slowly over time. For example, the road may get more shiny as it rains and less when the weather dries up; people may move books and furniture around in the room, and so on. An alternative that usually works quite well is to estimate the value of background pixels using a *moving average*. In this approach, you estimate the value of a particular background pixel as a weighted average of the previous values. Typically, pixels in the distant past should be weighted at zero, and the weights increase smoothly. Ideally, the moving average should track the changes in the background, meaning that if the weather changes quickly (or the book mover is frenetic) relatively few pixels should have nonzero weights, and if changes are slow the number of past pixels with nonzero weights should increase. The approach can be quite successful, but needs to be used on quite coarse scale images as Figures 9.2 and **??** illustrate.

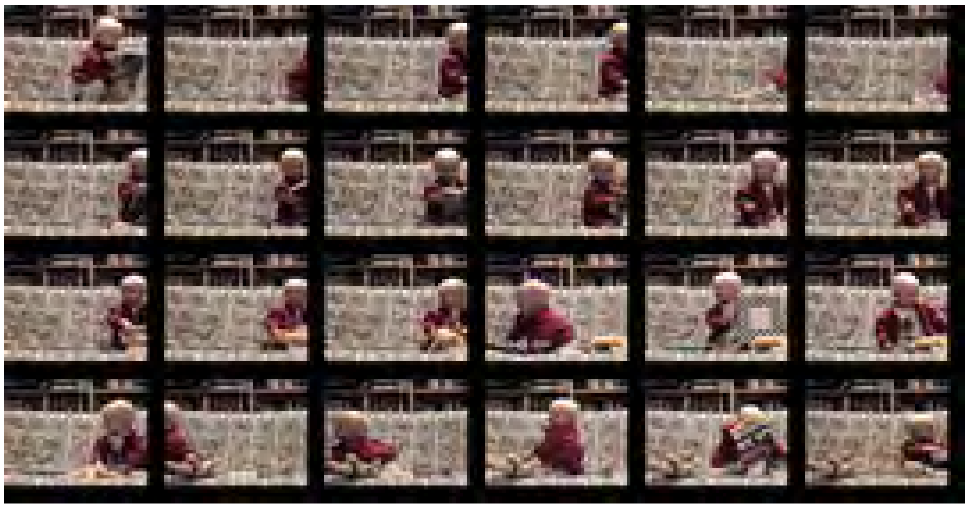

FIGURE 9.1: *The figure shows every fifth frame from a sequence of 120 frames of a child playing on a patterned sofa. The frames are used at an 80 x 60 resolution, for reasons discussed in Figure* **??**. *Notice that the child moves from one side of the frame to the other during the sequence.*

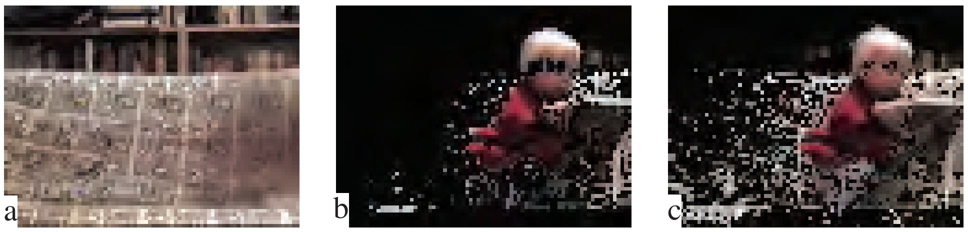

FIGURE 9.2: *Background subtraction results for the sequence of Figure 9.1 using 80 x 60 frames. I compare two methods of computing the background: (***a***) The average of all 120 frames — notice that the child spent more time on one side of the sofa than the other, leading to the faint blur in the average there. (***b***) Pixels whose difference from the average exceeds a small threshold. (***c***) Those whose difference from the average exceeds a somewhat larger threshold. Notice that, in each case, there are some excess pixels and some missing pixels.*

### 9.1.2 Shot Boundary Detection

A shot boundary detection algorithm must find frames in the video that are significantly different from the previous frame. Our test of significance must take account of the fact that, within a given shot, both objects and the background can move around in the field of view. Typically, this test takes the form of a distance; if the distance is larger than a threshold, a shot boundary is declared.

There are a variety of standard techniques for computing a distance. **Frame differencing** algorithms take pixel-by-pixel differences between each two frames in a sequence and sum the squares of the differences. **Histogram-based** algorithms compute color histograms for each frame and compute a distance between the histograms. **Block comparison** algorithms compare frames by cutting them into a grid of boxes and comparing the boxes. **Edge differencing** algorithms compute edge maps for each frame, and then compare these edge maps.

## 9.2 IMAGE SEGMENTATION AS CLUSTERING

There is a master recipe for image segmentation. This relies on *clustering*. a procedure that takes individual data items – for example, pixels, image patches – and produces blobs or *clusters* consisting of many similar data items.

**Procedure: 9.1** *Image Segmentation: Master recipe*

Compute a feature vector at each pixel of the image, then cluster the feature vectors. Each segment consists of the pixels whose feature vectors are in the same cluster.

Clustering algorithms rely on estimates of how similar two pixels are. You obtain these from distances between feature vectors describing the pixels. The key topics for this chapter are how to cluster and how to compute distances. I will use quite simple feature vectors in examples, but much of modern image segmentation rests on sophisticated feature vector constructions.

### 9.2.1 Clustering Generalities

Generally, to cluster data items, you must determine (a) how many clusters there are; and (b) which data items belong to which cluster. There are two natural clustering algorithms. In **agglomerative clustering**, you start with each data item being a cluster, and then merge clusters recursively to yield a good clustering. In **divisive clustering**, you start with the entire data set being a cluster, and then split clusters recursively to yield a good clustering. Mostly, divisive clustering isn't much used in vision applications.

Either algorithm needs to know when to stop. This can be difficult if there is no model for the process that generated the clusters. One strategy, popular in interactive data visualization and sometimes useful in other circumstances, is to recognize the recipes generate a hierarchy of clusters. You could simply generate the whole hierachy, then navigate it in some application appropriate way. For

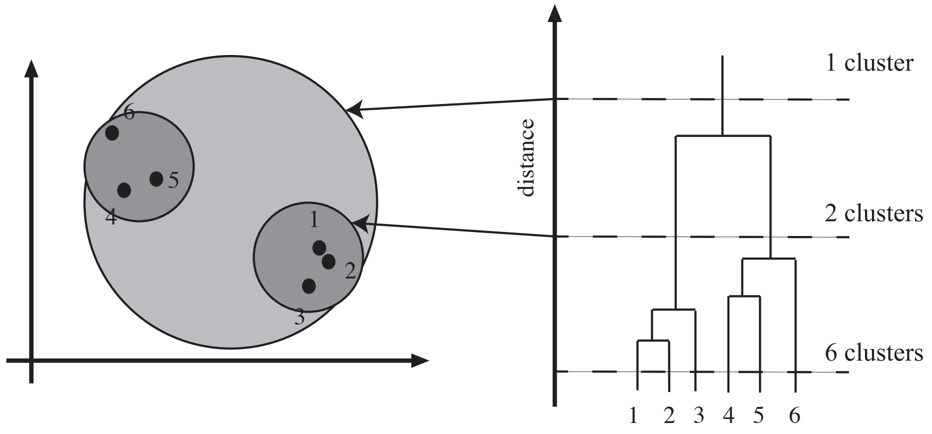

FIGURE 9.3: **Left**, *a data set;* **right**, *a dendrogram obtained by agglomerative clustering using single-link clustering. If one selects a particular value of distance, then a horizontal line at that distance splits the dendrogram into clusters. This representation makes it possible to guess how many clusters there are and to get some insight into how good the clusters are.*

visualization, this hierarchy can be displayed 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. Dendrograms are difficult to intepret when there are many data items, but can be a helpful guide in simple cases (Figure 9.3).

Another important thing to notice about clustering from the example of figure **??** 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. This is a general feature of clustering as a problem. It is hardly ever helpful to talk about whether a clustering is right, but it is often important to focus on how useful it is.

### 9.2.2 Elementary Distances

For the moment, work with elementary feature vectors. These are: color, color and position, and color, position and texture. For **color** features, it's a good idea to use a color representation where distances reflect perceived change of color well, so Lab values or Luv values are a sensible choice. Notice that regions of pixels with similar colors may not be connected. For **color and position** features, stack the $x$, $y$ position of the pixel together with the color. For **color, position and texture** features, stack a texture descriptor together with position and color. This texture descriptor could come from many sources. A traditional option is to choose a set of small pattern detectors like filters (Chapter 21.3). The texture descriptor is then an average of each filter's response in some local patch.

Agglomerative clustering needs a good inter-cluster distance to fuse nearby clusters. There are three recipes for inter cluster distances. The distance between the closest elements tends to yield extended clusters and is known to statisticians as *single-link clustering*). The maximum distance between an element of the first cluster and one of the second tends to yield rounded clusters and is known to statisticians as *complete-link clustering.* The average of distances between elements in the cluster also tends to yield rounded clusters and is known to statisticians as *group average clustering.*

### 9.2.3 The Mahalanobis Distance

The clustering methods I have described all depend on some measure of distance. Even the affinity measure of Section 21.3 depends on a distance. But distances between feature vectors can be misleading, because different components might be scaled differently. As a simple example, imagine clustering using a pixel feature vector consisting of R, G and B (on a scale from 0-1) and two position coordinates on a scale from 1-512. The distances you work with will be entirely dominated by the position coordinates; your results would change very significantly if the position coordinates were scaled from 0-1.

If you are working with a small feature vector whose components are easily understood, you may be able to control this problem with elementary means. But it is relatively straightforward to construct very large vectors of features whose meaning is quite obscure (some details in Chapters 21.3 and 21.3). Computing sensible distances between these features requires care.

Start with a dataset of $N$ $d$-dimensional vectors; write $\{\mathbf{x}\}$ for the dataset and $\mathbf{x}_i$ for the $i$'th item. Write the covariance matrix for this dataset $\mathsf{Covmat}\left(\{\mathbf{x}\}\right)$.

**Definition: 9.1** *The Mahalanobis Distance*

The *Mahalanobis distance* between two vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ is

$$(\mathbf{x}_i - \mathbf{x}_j)^T \mathsf{Covmat}\left(\{\mathbf{x}\}\right)^{-1} (\mathbf{x}_i - \mathbf{x}_j)$$

To understand this distance, diagonalize the covariance matrix to get

$$\mathcal{U}^T \mathsf{Covmat}\left(\{\mathbf{x}\}\right) \mathcal{U} = \Lambda.$$

Because $\mathcal{U}$ is a rotation, it has no effect on distances. If you transform the coordinates to obtain $\mathbf{r}_i = \mathcal{U}\mathbf{x}_i$, the covariance for the $\mathbf{r}_i$ is $\Lambda$ **exercises** . This is diagonal, so the directions are independent. In the Mahalanobis distance, each direction is scaled by its variance **exercises** . This makes sense – if the "blob" of data is spread out more in one direction, large differences in that direction should not count much when you compute the distance. But in directions where the data does not spread out, even small distances are important.

### 9.2.4 Reducing Dimension

Just using the Mahalanobis distance is often not a good idea. Typically, you would estimate the covariance from a sample of vectors (for example, descriptors associated with a random subset of pixels from a random subset of images), then use it for all vectors. If the vectors are high dimensional, there is a strong chance that many directions have very small variance. However, there may be minor fluctuations in these directions in the data you cluster (as opposed to the data you use to estimate the covariance matrix) and these minor fluctuations will produce huge distances. **exercises** It turns out that, for high dimensional data, this is the usual behavior – you should expect problems from this effect.

The effect is a manifestation of an important rule of thumb. High dimensional datasets are almost always "lying" about their dimension. Experience shows that the items in most high dimensional data sets can be represented successfully (for some purposes!) as the mean plus a weighted sum of a small set of basis vectors. You can think of the dataset as lying on a low dimensional space inside the original space. It's an experimental fact that this model of a dataset is usually accurate for real high-dimensional data, and it is often an extremely convenient model. Furthermore, representing a dataset like this very often suppresses noise – if the original measurements in your vectors are noisy, the low dimensional representation may be closer to the true data than the measurements are.

You can compute a lower dimensional representation, and obtain a Mahalanobis distance using that. Start with a dataset of $N$ $d$-dimensional vectors $\{\mathbf{x}\}$. Diagonalize $\mathsf{Covmat}(\{\mathbf{x}\}) = \mathsf{Covmat}(\{\mathbf{x}\})$ to get

$$\mathcal{U}^T \mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} = \Lambda.$$

Do this by finding the eigenvalues and eigenvectors of $\mathsf{Covmat}(\{\mathbf{x}\})$. **exercises** Ensure when you do this that the terms on the diagonal of the diagonal matrix $\Lambda$ are sorted largest to smallest. This is just a matter of checking whether your API does it naturally, or you have to sort the diagonal of $\Lambda$ and the columns of $\mathcal{U}$.

Now consider the dataset $\{\mathbf{r}\}$, constructed using the rule

$$\mathbf{r}_i = \mathcal{U}^T \mathbf{x}_i.$$

The covariance of this dataset is diagonal. The values on the diagonal are interesting. It is quite usual for high dimensional datasets to have a small number of large values on the diagonal, and a lot of small values. This means that the blob of data is really a low dimensional blob in a high dimensional space. For example, think about a line segment (a 1D blob) in 3D. Now assume that $\mathsf{Covmat}(\{\mathbf{r}\})$ has many small and few large diagonal entries. In this case, the blob of data represented by $\{\mathbf{r}\}$ admits an accurate low dimensional representation. Choose $s$ for the dimension of the low dimensional representation (the exercises sketch out a procedure to do this **exercises** ). Form the $s \times d$ matrix $\mathcal{P}_s$ consisting of the first $s$ rows of $\mathcal{U}^T$. Then the dataset $\{\mathbf{l}\}$, constructed using the rule

$$\mathbf{l}_i = \mathcal{P}_s \mathbf{x}_i$$

consists of $s$ dimensional vectors, and their covariance (a) is diagonal and (b) has large values on the diagonal (by choie of $s$). Now use the Mahalanobis distance for this representation.

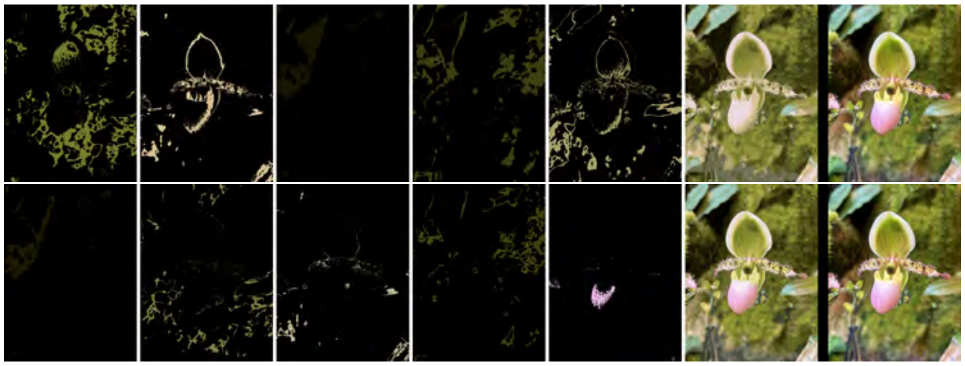

FIGURE 9.4: *K-means on image color yields very scattered segments. Segmentations of the image on the* **right** *obtained with k-means applied to RGB color values using* **top** $k = 10$ *and* **bottom** $k = 30$. *Each row shows five segments selected at random and colored with the mean color in the segment, then the image obtained by "flattening" the segments next to the true image.*

**Procedure: 9.2** *Computing Mahalanobis Distance with Dimension Reduction*

**At training time:** Obtain a large, representative sample of the data items you will work with; write $\{\mathbf{x}\}$ for the dataset of $N$ $d$-dimensional vectors and $\mathbf{x}_i$ for the $i$'th item. Write the covariance matrix for this dataset $\mathsf{Covmat}(\{\mathbf{x}\})$. Diagonalize this covariance to obtain

$$\mathcal{U}^T \mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} = \Lambda.$$

Ensure that the values along the diagonal are sorted. Choose $s < d$ for the new dimension of the data. Form the $s \times d$ matrix $\mathcal{P}_s$ consisting of the first $s$ rows of $\mathcal{U}^T$. Form $\Lambda_s$ consisting of the $s \times s$ upper left block of $\Lambda$. Finally, form $\mathcal{D} = \mathcal{P}_s^T \Lambda_s^{-1} \mathcal{P}_s$.
**At run time:** The dimension reduced Mahalanobis distance between two vectors $\mathbf{u}$ and $\mathbf{v}$ is

$$(\mathbf{u}-\mathbf{v})^T \mathcal{D}(\mathbf{u}-\mathbf{v})$$

## 9.3 THE K-MEANS ALGORITHM

Write $\mathbf{x}_i$ for a set of $N$ data items, which have been coerced to be vectors. Assume you know that there are $k$ clusters. Write $\mathbf{c}_j$ for the center of the $j$th cluster. Write $\delta_{i,j}$ for a discrete variable that records which cluster a data item belongs to, so

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

Every data item belongs to exactly one cluster, so $\sum_j \delta_{i,j} = 1$. Every cluster must contain at least one point, so $\sum_i \delta_{i,j} > 0$ for every $j$. The sum of squared distances from data points to cluster centers is then

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

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.

You could cluster the data by choosing the $\delta$ 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 $\Phi$ 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.

Notice that if the $\mathbf{c}$'s are known, getting the $\delta$'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 zero. 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. These observations yield a remarkably effective approximate algorithm. 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; if there are no points in the cluster, restart the cluster by choosing some point uniformly and at random from the dataset and making that the cluster center.

Choose a start point by randomly choosing cluster centers (there are better options, below), and then iterate these stages alternately. This process eventually converges (**exercises** ). It is not guaranteed to converge to the global minimum of the objective function, however. This algorithm is usually referred to as *k-means*

**Procedure: 9.3** *K-means clustering*

**Initialize** by choosing $k$ and $k$ initial cluster centers $\mathbf{c}_i$.
**Iterate** until convergence:

- Allocate each data item $\mathbf{x}_j$ to the closest cluster center.
- Replace each center with the mean of the points allocated to that cluster; if there are no points in the cluster, choose some point uniformly and at random from the dataset and make that the cluster center.

**Test convergence** by a combination of tests. Always stop when the number of iterations exceeds a threshold. You could stop if the cluster centers have moved by less than a threshold between iterations. Finally, if the allocation of data items to cluster centers has not changed between iterations, the method must have converged.

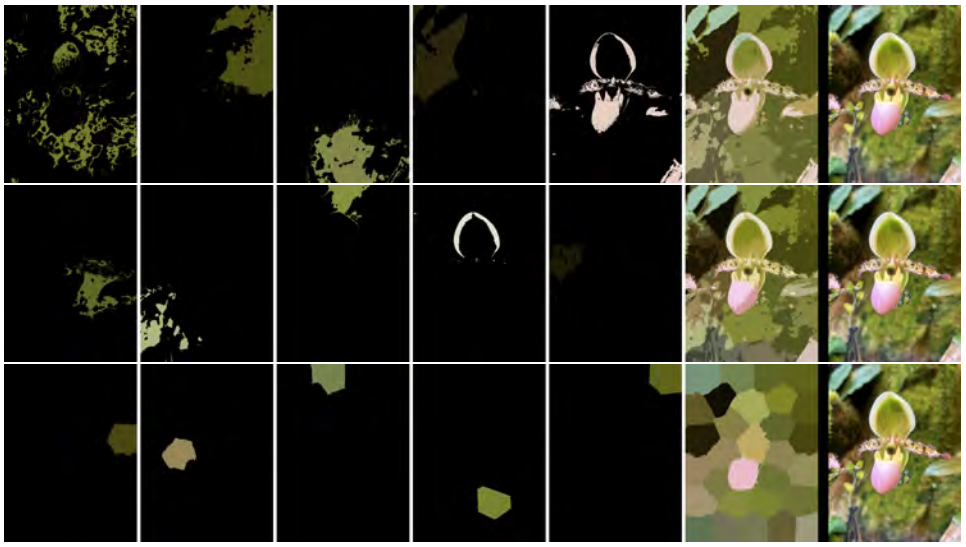

FIGURE 9.5: *K-means on image color and position yields more compact segments, depending on the scaling of the xy distance relative to the color distance. Segmentations of the image on the* **right** *obtained with k-means applied to RGB color values using* **top** $k = 10$; **center** *and* **bottom**$k = 30$. *Each row shows five segments selected at random and colored with the mean color in the segment, then the image obtained by "flattening" the segments next to the true image. The top and center rows have xy distance scale so that the height of the image is 1; the bottom row scales xy distance so the height is 10, forcing clusters to be much more compact.*

## 9.3.1 Initializing K-means

One natural strategy for initializing k-means is to choose $k$ data items at random, then use each as an initial cluster center. This approach is widely used, but has some difficulties. The quality of the clustering can depend quite a lot on initialization, and an unlucky choice of initial points might result in a poor clustering. One (again quite widely adopted) strategy for managing this is to initialize several times, and choose the clustering that performs best in your application. Another strategy, which has quite good theoretical properties and a good reputation, is known as *k-means++*.

**Procedure: 9.4** *K-means++ for initializing k-means*

Choose a point $\mathbf{x}$ uniformly and at random from the dataset to be the first cluster center. Then compute the squared distance between that point and each other point; write $d_i^2(\mathbf{x})$ for the distance from the $i$'th point to the first center. Now choose the other $k-1$ cluster centers as IID draws from the probability distribution

$$\frac{d_i^2(\mathbf{x})}{\sum_u d_u^2(\mathbf{x})}.$$

### 9.3.2 How to choose K

Usually, you don't know how many clusters there should be, and need to choose this by experiment. The obvious strategy – cluster for a variety of different values of $k$, then look at the value of the cost function for each – does not work. If there are more centers, each data point can find a center that is closer to it, so the value should go down as $k$ goes up. The best $k$ is then the number of data points, which is not helpful.

In some special cases, you might know a label associated with each data point. 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, you won't have a label to check purity.

The alternative strategy, which might seem crude to you, 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 Section **??**). 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 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.

### 9.3.3 Scattered Points and Junk Clusters

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.

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

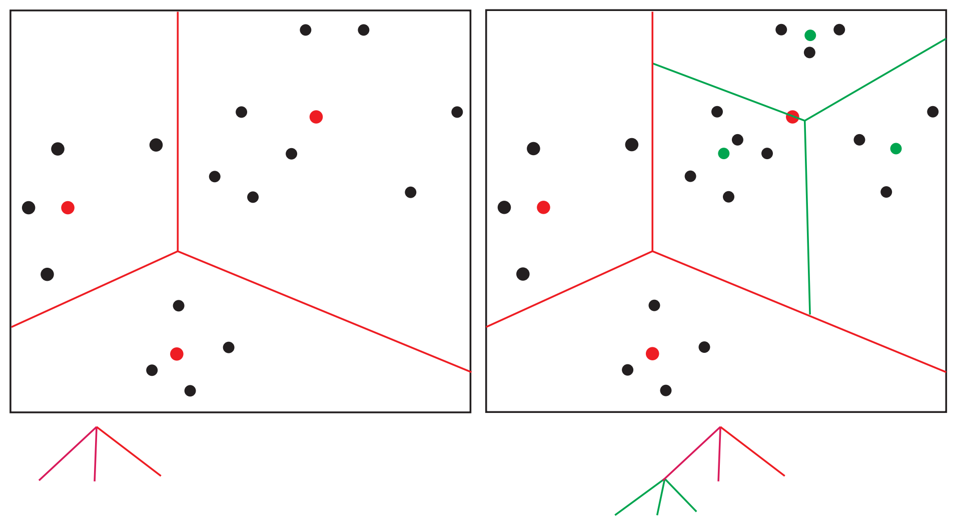

FIGURE 9.6: *Hierachical k-means builds a tree of cells in space. On the* **left**, *a set of data points (***black***) in 2D, with three cluster centers in* **red**. *This clustering divides the plane into three cells corresponding to the cluster centers. The cells contain the points that are closest to their cluster center. On the* **right**, *the points in the top right cell are clustered again into three clusters (centers in* **green**), *so that cell is subdivided into three cells. Below each 2D set of points is the relevant abstract tree.*

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 finds a set of cluster centers and an allocation of data items to cluster centers that minimizes the sum of distances between data items and their cluster center. The algorithm chooses centers and then allocates items to the cluster with the closest center, and then updates the centers; this is repeated until the (guaranteed) convergence to a local minimum.*

### 9.3.4 Efficient Clustering and Hierarchical K-Means

One important difficulty occurs in applications. You might need to have an enormous dataset (billions 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

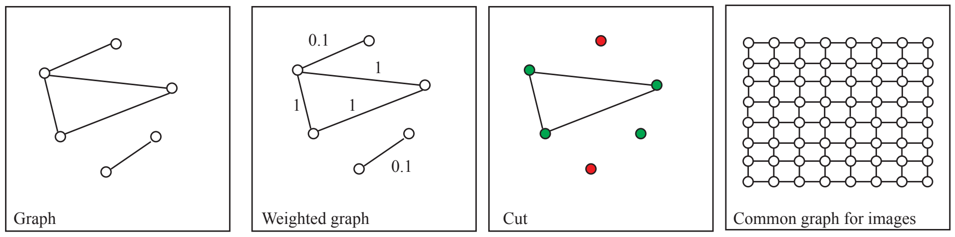

FIGURE 9.7: *Graphs are quite visual concepts. On the* **left***, a graph (vertices are circles, and edges line segments; note this graph isn't connected).* **Center left***, a weighted graph;* **center right***, a cut (***red** *vertices are in $\mathcal{A}$ and* **green** *vertices are in $\mathcal{B}$; and* **right***, the most usual graph used for images, shown for a small image.*

is closest to a particular data point scales linearly with $k$ (and you have to do this for every data point at every iteration).

Manage this problem by building a hierarchy of k-means clusters. Randomly subsample the data (typically quite aggressively), then cluster the sample 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. You now have something that looks like a two-level tree of clusters. The leaves of this tree contain subsets of the data. All intermediate nodes contain a set of cluster centers, and there is a child associated with each cluster center. Of course, applying this process recursively will produce a more interesting tree of clusters (**exercises** ). You can stop the recursion by not clustering when there are few data points in a leaf.

You should think of this tree as dividing the space of data items into cells. It is straightforward to draw the tree. The root consists of a set of cluster centers, each of which is associated with a child node. Datapoints belong to the child node associated with the closest cluster center, meaning that the root divides the space into cells (formally, Voronoi cells). This applies recursively to the children, which each divide their cell into subcells, and so on (Figure 9.6)

**Remember this:** *Build a tree using k-means by clustering a sample of the data, then allocating new data to the cluster with the closest center, and recurring. Stop when there is too little data in a cluster.*

## 9.4 CLUSTERING WITH GRAPH THEORETIC METHODS

A *graph* is given by a set of vertices $\mathcal{V}$ and a set of edges (pairs of vertices, $\mathcal{E}$). The graph is a *weighted graph* if each edge has an associated weight. A *cut* partitions the graph into two sets of vertices (say $\mathcal{V}_A \subset \mathcal{V}$ and $\mathcal{V}_B \subset \mathcal{V}$) and two sets of edges ($\mathcal{E}_A \subset \mathcal{E}$ and $\mathcal{E}_B \subset \mathcal{E}$) such that $\mathcal{V}_A \cap \mathcal{V}_B = \emptyset$, $\mathcal{V}_A \cup \mathcal{V}_B = \mathcal{V}$, $\mathcal{E}_A$ contains only pairs of $\mathcal{V}_A$ vertices and $\mathcal{E}_B$ contains only pairs of $\mathcal{V}_B$ vertices. These ideas are quite visual (Figure 9.7).

Clustering is about relations – which data items are close, and which are far – and you can use a graph to specify which relations should matter to a clustering algorithm by making each data item a vertex and each relation that matters an edge. You then apply weights to those relations, and cut the graph (often, but not always, repeatedly) to obtain subgraphs which are clusters.

### 9.4.1 GrabCut: Separating Foreground from Background with a Graph

An important application for graphs is interactive segmentation, where you are given an image; a user does something to hint at what the foreground might be; and you must cut the foreground object from the background. The segmentation boundary should run exactly along the boundary of the object. The hints might take the form of a box around the foreground object or some scribbles on the image (Figure 9.8).

For the moment, assume that the hints have been good enough that you have a model of what foreground and background pixels look like. Write $\mathbf{f}$ for the feature vector describing a pixel. Express your appearance model for foreground and background as a conditional probability model giving the probability of a particular feature vector conditioned on the pixel coming from foreground or background; write $P(\mathbf{f}|\text{foreground})$, etc. The next section sketches some ways to get these models.

Each pixel is now a vertex in a graph, and I will construct an optimization problem that makes it possible to cut this graph into two components, one for foreground and one for background. Reindex the pixels so that they have one index, and construct an appropriate set of edges. By far the most common choice is to connect each pixels to its four neighbors only (ie up, down, left and right). Associate $z_i$ with the $i$'th node, where $z_i$ is one when the node is foreground and zero when it is background. Now construct an optimization problem for $z_i$ out of the graph by associating a cost with each node and a cost with each edge. The solution of that optimization problem is a segmentation.

Assume the graph has no edges. Then the optimization problem would be to choose the $z$ that minimize:

$$\sum_i \left(z_i\left[-\log P(\mathbf{f}_i|\text{foreground})\right] + (1-z_i)\left[-\log P(\mathbf{f}_i|\text{background})\right]\right).$$

Notice how $z_i$ switches between costs for foreground and background. This problem isn't particularly interesting as an optimization problem (choose each $z_i$ independently that minimizes the cost). The solution provides a bad segmentation, for an interesting reason: each pixel is labelled entirely independently of its neighbor, so you should expect a number of isolated foreground pixels in areas of mostly background, and vice versa (Figure 9.2 shows this effect). This optimization problem associates costs with vertices only, and could be written

$$\sum_{i\in\mathcal{V}} \left(z_i V_{i,1} + (1-z_i)V_{i,0}\right).$$

where the costs $V_{i,\cdot}$ are known (and are derived from the probability model as above).

A more interesting case occurs when the graph has edges. Assume there is an edge from $i$ to $j$. There are four possible sets of label: $z_i = z_j = 1$, $z_i = z_j = 0$, $z_i = 1 \mathrm{and} z_j = 0$ and $z_j = 1 \mathrm{and} z_i = 0$. Each could have a cost associated with it. The cost associated with this edge alone can be written:

$$z_i z_j E_{ij,11} + (1 - z_i) z_j E_{ij,01} + z_i (1 - z_j) E_{ij,10} + (1 - z_i)(1 - z_j) E_{ij,00}$$

where $E_{ij,11}$ is the cost for having $z_i = z_j = 1$ across the edge. You must now minimize

$$\sum_{i \in \mathcal{V}} (z_i V_{i,1} + (1 - z_i) V_{i,0}) + \sum_{(i,j) \in \mathcal{E}} \left( \begin{array}{c} z_i z_j E_{ij,11} + \\ (1 - z_i) z_j E_{ij,01} + \\ z_i (1 - z_j) E_{ij,10} + \\ (1 - z_i)(1 - z_j) E_{ij,00} \end{array} \right)$$

by choice of $z_i$. Intuition may (should, if you've taken an algorithms class) tell you that this is a nasty optimization problem. This is true in general case, but not in the special case associated with images. For images, you expect that pixels look like their neighbors. Further, objects tend to be "blobby", so if you choose the graph to connect each pixel to its four neighbors only (ie up, down, left and right), most $z$ will look like their neighbors as well. In turn, you should choose $E_{ij,11} = E_{ij,00} = 0$ (it's cheap to agree with your neighbors) and $E_{ij,01} > 0$ , $E_{ij,10} > 0$ (it's expensive to disagree with your neighbors). For this case, it is known how to solve the optimization problem efficiently ([]) and good fast codes are available (for example, OpenCV has one).

Write $\mathbf{f}_i$ for the feature representing the $i$'th pixel. GrabCut chooses

$$E_{ij,01} = E_{ij,10} = \gamma e^{-\beta \left[ (\mathbf{f}_i - \mathbf{f}_j)^T (\mathbf{f}_i - \mathbf{f}_j) \right]}.$$

Here $\beta = (1/2)(1/m)$, $m$ is an average of $(\mathbf{f}_i - \mathbf{f}_j)^T (\mathbf{f}_i - \mathbf{f}_j)$ over a sample of distinct image pixels, and $\gamma$ is a parameter. Notice that once you have a solution, you can re-estimate the foreground and background models. Just use the foreground or background pixels identified by the algorithm. These models may change from the models used to obtain the solution, so it is worthwhile iterating the two steps a few times.

A large value of $\gamma$ makes it very expensive for two pixels with an edge between them to have different $z$ values. If you make the common choice of graph, then pixels are forced to agree with their neighbors in the absence of strong evidence they should disagree. This should remove isolated foreground or background labels. A large value of $\gamma$ favors a shorter boundary between foreground and background, so too large a value of $\gamma$ can result in an oversimplified boundary. As you reduce the value of $\gamma$, the boundary will become more complicated, but isolated pixels may appear.

### 9.4.2 Building Models of Foreground and Background for GrabCut

Assume you have a collection of feature vectors from mostly foreground pixels (call the $i$'th such $\mathbf{f}_i$) and another of background pixels (the $i$'th is $\mathbf{b}_i$). You must build models of the conditional probabilities $P(\mathbf{f}|\text{foreground})$ and $P(\mathbf{f}|\text{background})$ from

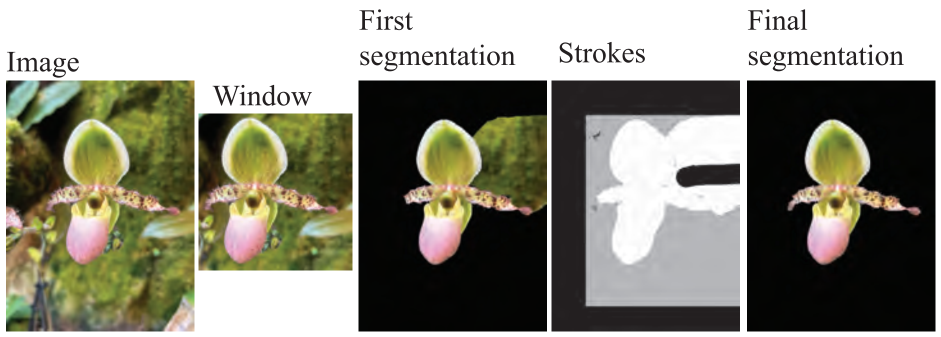

FIGURE 9.8: *An example of an interactive segmentation of a foreground object with GrabCut. The user selects a window in the image which is mostly foreground; the method then builds a model of foreground using the pixels in this window and a model of background pixels using the rest of the image. These models lead to the initial segmentation, which is missing a small blob of object pixels (on the "wing" of the petal at the bottom left) and has a large streak of extra pixels. The user then applies strokes to the image to identify known foreground and known background pixels, resulting in new foreground and background models and an improved segmentation.*

these examples. The simplest option is a Gaussian (details in exercises, **exercises** ). An alternative is to use histograms, but you must be careful about zero counts (details in exercises, **exercises** ). The most popular option is a mixture of Gaussians, which requires more careful estimation (some details in exercises).

Using some background pixels to build the foreground model (and vice versa) should not lead to major problems. Doing this will cause the probability model to be somewhat smoothed. For example, if the foreground is red and the background is green, there will be slightly more probability that the foreground generates a green pixel than there should be. But there will be much more probability that the background generates this green pixel, so the segmentation should be fine. This is quite an important point, because it means the user can provide quite rough hints to the model and still get a good segmentation. Usually, the first models are built from a box on the image containing"likely foreground" pixels (those outside the box are "likely background"). If this isn't sufficient to get a good segmentation, the user can apply *strokes* to the image – make marks on the image identifying pixels that are "certainly foreground" or "certainly background" (Figure 21.3). GrabCut is available in a number of API's (**exercises** ).

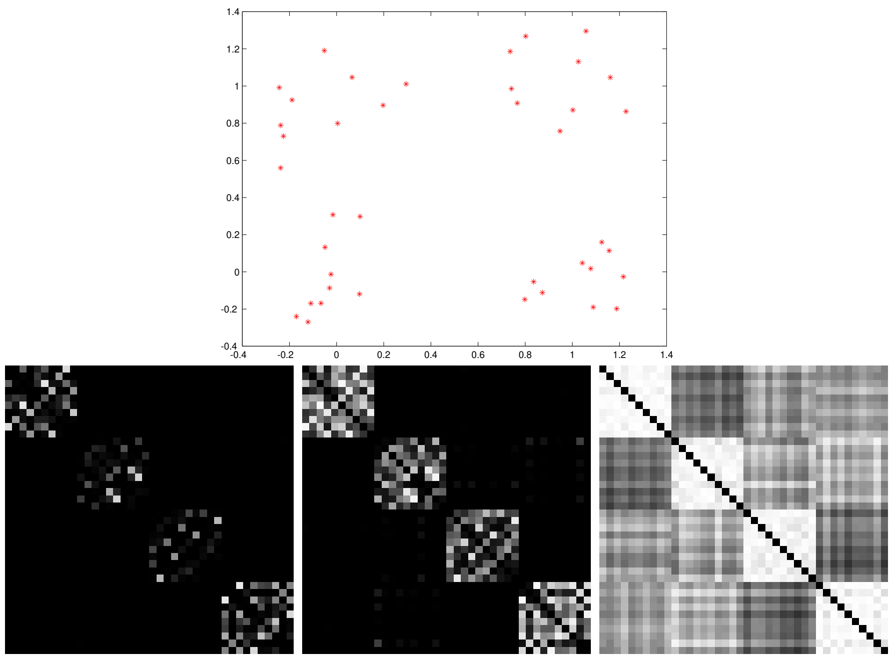

FIGURE 9.9: *The choice of scale for the affinity affects the affinity matrix. The top row shows a dataset, which consists of four groups of 10 points drawn from a rotationally symmetric normal distribution with four different means. The standard deviation in each direction for these points is 0.2. In the second row, affinity matrices computed for this dataset using different values of $\sigma_d$. On the* **left**, $\sigma_d = 0.1$, *in the* **center** $\sigma_d = 0.2$, *and on the* **right**, $\sigma_d = 1$. *For the finest scale, the affinity between all points is rather small; for the next scale, there are four clear blocks in the affinity matrix; and for the coarsest scale, the number of blocks is less obvious.*

**Remember this:** *GrabCut implements interactive segmentation of a foreground object. The user marks rough estimates of foreground and background pixels, resulting in a probability model of foreground and background appearance; this is used to segment the image by solving an optimization problem that allocates pixels to foreground or background. The optimization problem balances similarity to foreground or background with the tendency of pixels to have the same class as their neighbors.*

### 9.4.3 Affinities

An alternative application of graphs to clustering is this: Take each data item of the collection to be clustered and associate it with a vertex on a graph. Now construct

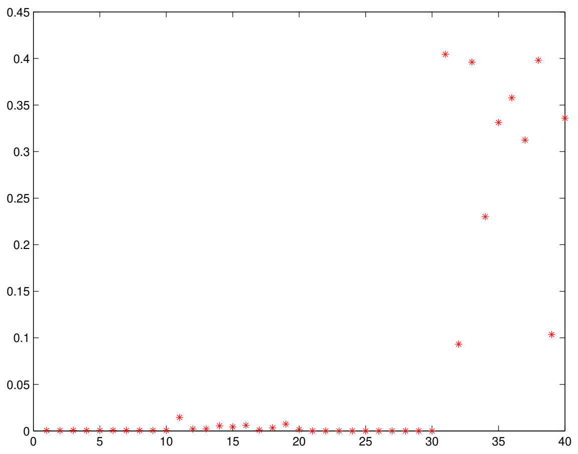

FIGURE 9.10: *The eigenvector corresponding to the largest eigenvalue of the affinity matrix for the dataset of Figure 9.9 using $\sigma_d = 0.2$. Notice that most values are small, but some — corresponding to the elements of the main cluster — are large. The sign of the association is not significant, because a scaled eigenvector is still an eigenvector.*

an edge from every element to every other, and associate with this edge a weight representing the extent to which the elements are similar. Each weight is an *affinity* – if the two vertices are similar, it is large, and if they are different, it is small. Now cut edges in the graph to form a good set of connected components — ideally, the within-component edges are large compared with the across-component edges. Each component is a cluster.

It is straightforward to construct affinities out of distances. If $d(\mathbf{x}_i, \mathbf{x}_j)$ is the distance between two feature vectors, then

$$\exp\left(-\frac{d(\mathbf{x}_i, \mathbf{x}_j)^2}{2\sigma}^2\right)$$

is an affinity (big when similar, small when different). Here $\sigma$ is a scale used to adjust the affinity. Now represent the affinities between each pair of points in a matrix $\mathcal{A}$, and recover clusters by analysis of that matrix.

### 9.4.4 Extracting Clusters with Eigenvectors

A good cluster is one where elements that are strongly associated with the cluster also have large values connecting one another in the affinity matrix. Write $\mathbf{w}$ for the vector of weights linking elements to the cluster. Now the function

$$\mathbf{w}^T \mathcal{A} \mathbf{w}$$

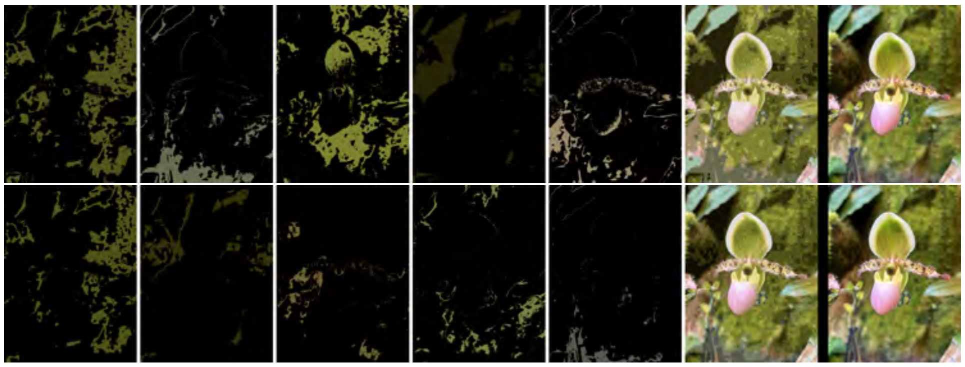

FIGURE 9.11: *Normalized cuts on image color alone yields very scattered segments. I used the OpenCV implementation, which requires to be told how many segments it must find, and has deep relations to k-means. Segmentations of the image on the* **right** *obtained with normalized cuts applied to RGB color values using* **top** $k = 10$ *and* **bottom** $k = 30$. *Each row shows five segments selected at random and colored with the mean color in the segment, then the image obtained by "flattening" the segments next to the true image.*

is a sum of terms of the form

$$\begin{aligned}&\{\text{association of element } i \text{ with the cluster}\}\\ &\quad\times\{\text{affinity between } i \text{ and } j\}\\ &\quad\times\{\text{association of element } j \text{ with the cluster}\}.\end{aligned}$$

You can obtain a cluster by choosing a set of association weights that maximize this objective function. The objective function is useless on its own because scaling $\mathbf{w}$ by $\lambda$ scales the total association by $\lambda^2$. However, you can normalise the weights by requiring that $\mathbf{w}^T\mathbf{w} = 1$, so it is natural to maximize $\mathbf{w}^T\mathcal{A}\mathbf{w}$ subject to $\mathbf{w}^T\mathbf{w} = 1$. This is an eigenvalue problem (**exercises** ) and you must solve

$$\mathcal{A}\mathbf{w} = \lambda\mathbf{w}.$$

For problems where reasonable clusters are apparent, these cluster weights should be large for some elements, which belong to the cluster, and nearly zero for others, which do not (Figure 9.10). In fact, you can get the weights for other clusters from other eigenvectors of $\mathcal{A}$ as well.

In typical vision problems, there are strong association weights between relatively few pairs of elements. You can reasonably expect to be dealing with clusters that are quite tight and distinct. A natural procedure is to take the first eigenvector, use that to decide what belongs in the first cluster (large values in the eigenvector), then take the second eigenvector to decide what belongs in the second cluster (large values in the eigenvector for things that aren't in the first) and so on.

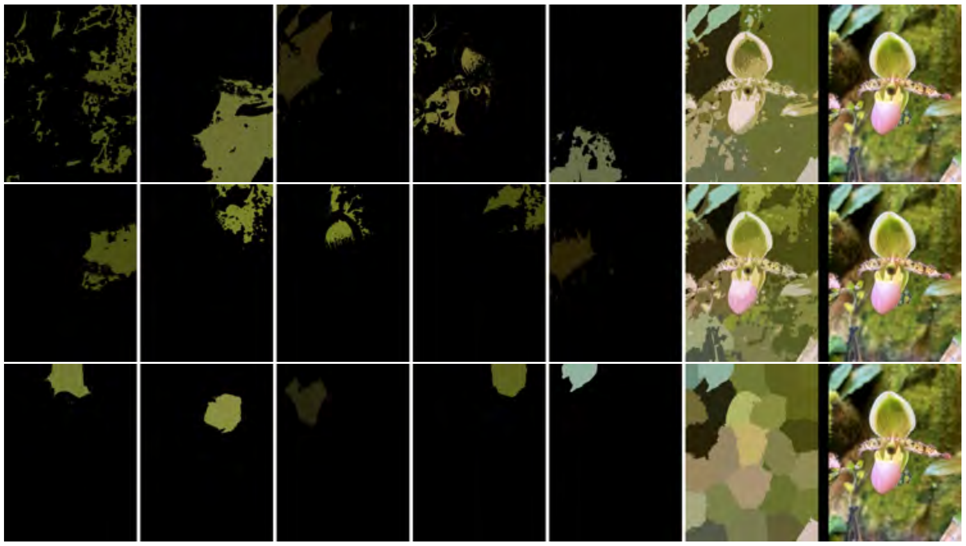

FIGURE 9.12: *Normalized cuts on image color and position yields more compact segments, depending on the scaling of the xy distance relative to the color distance. I used the OpenCV implementation, which requires to be told how many segments it must find, and has deep relations to k-means. Segmentations of the image on the* **right** *obtained with normalized cuts applied to RGB color values using* **top** $k = 10$*;* **center** *and* **bottom**$k = 30$*. Each row shows five segments selected at random and colored with the mean color in the segment, then the image obtained by "flattening" the segments next to the true image. The top and center rows have xy distance scale so that the height of the image is 1; the bottom row scales xy distance so the height is 10, forcing clusters to be much more compact.*

### 9.4.5 Normalized Cuts

An alternative approach is to cut the graph into two connected components such that the cost of the cut is a small fraction of the total affinity within each group. Formalize this as decomposing a weighted graph $V$ into two components $A$ and $B$ and scoring the decomposition with

$$\frac{cut(A,B)}{assoc(A,V)} + \frac{cut(A,B)}{assoc(B,V)}$$

(where $cut(A,B)$ is the sum of weights of all edges in $V$ that have one end in $A$ and the other in $B$, and $assoc(A,V)$ is the sum of weights of all edges that have one end in $A$). This score is small if the cut separates two components that have few edges of low weight between them and many internal edges of high weight. One seeks the cut with the minimum value of this criterion, called a *normalized cut*. Actually finding this cut is algorithmically tricky. Reasonable approximation procedures are known, but are out of scope. The criterion is successful in practice (Figure 9.11 and 9.12). At least one efficient implementation is available for download (`https://ncut-pytorch.readthedocs.io/en/latest/`).

There are very deep relationships between normalized cuts and k-means, which should be suggested by comparing Figure **??** with Figure 9.11 or Figure **??** with Figure 9.12. Current practice uses an approximation to normalized cuts to establish a representation to which k-means is then applied. The best behavior occurs with feature constructions I describe in later chapters.

## 9.5 EVALUATING SEGMENTATION

Evaluating clustering in general is hard because there is usually no known right answer. But image segmentation is special, because you can compare with human segmentations. Humans are fairly – but not completely – consistent in the way they segment images. The Berkeley Segmentation Dataset (500 image version at `https://github.com/BIDS/BSDS500`) contains a set of images that have been manually segmented. Each image has been segmented by multiple annotators, who are not required to agree, so that the dataset contains some information about the difference in opinion between human annotators.

Segment boundaries can be taken as edges. These are evaluated by testing whether human edge points are the same as predicted edge points. The *F-measure* measures this property. Write TP (*true positive*) for the number of points where the method predicts an edge and so do humans; FP (*false positive*) for the number of points where the method predicts an edge and humans do not; TN (*true negative*) for the number of points where neither method nor human predict an edge; and FN (*false negative*) for the points where the method does not predict an edge but humans do.

**Definition: 9.2** *Recall*

Recall is the fraction of predicted edge points that are also marked by humans as edge points. The recall of a system is given by

$$R = \frac{TP}{TP + FP}.$$

**Definition: 9.3** *Precision*

Precision is the fraction of actual edge points (those marked by humans) that the method finds. The precision of a system is given by

$$P = \frac{TP}{TP + FN}.$$

You cannot properly evaluate a predictor using only one of these numbers. For example, you can get excellent recall by building an extremely cautious predictor that has no false positives, and very few true positives (though the precision will be low). Similarly, an enthusiastic predictor might label almost everything as positive, and so get a very good precision (but low recall). One can summarize these two numbers with an F1 measure, which is the harmonic mean of recall and precision.

**Definition: 9.4** *The F-measure*

A system with precision $P$ and recall $R$ has F-measure

$$F = \frac{2}{\frac{1}{R} + \frac{1}{P}}.$$

With some effort, you can show that

$$F = \frac{2}{\frac{1}{R} + \frac{1}{P}} = \frac{2TP}{2TP + FP + FN}.$$

Notice that to get a large value of this number, both R and P must be large.

In practice, evaluation requires care, and it is important to have a consistent protocol, because humans are not consistent with one another. This means that a very good predicted edge point may be very close to, but not on top of, any human prediction. Such a point should be counted as a true positive. There is a standard protocol in place, with code available at `https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html`.

## 9.6 YOU SHOULD

### 9.6.1 remember these facts:

| | |
|---|---|
| The K-means algorithm | 155 |
| The Hierachical K-means algorithm | 156 |
| The GrabCut Algorithm | 160 |

### 9.6.2 remember these procedures:

| | |
|---|---|
| Image Segmentation: Master recipe | 147 |
| Computing Mahalanobis Distance with Dimension Reduction | 151 |
| K-means clustering | 153 |
| K-means++ for initializing k-means | 154 |

### 9.6.3 be able to:

- Cluster data items using k-means.
- Rescale feature representations as required using a Mahalanobis distance and dimension reduction.
- Cluster pixels using k-means using color, position and filter outputs as features to obtain simple segmentations.
- Use grabcut as implemented by your chosen API.

## EXERCISES

### QUICK CHECKS

**9.1.** Single link clustering tends to yield extended clusters; why?

**9.2.** Complete link clustering tends to yield rounded clusters; why?

**9.3.** Assume the covariance of some features is diagonal. When you compute a distance function, why does it make sense to scale each direction by the standard deviation?

**9.4.** Recall $\mathsf{Covmat}(\{\mathbf{x}\})$ must be a symmetric matrix. Show you can diagonalize this matrix by finding its eigenvalues and eigenvectors.

**9.5.** Confirm that the cost function for k-means does not go up at each iteration, as long as no cluster required restarting.

**9.6.** Section 9.3.2 has "If there are more centers, each data point can find a center that is closer to it, so the value should go down as $k$ goes up." Does this *always* happen? could the value go up?

**9.7.** Section 9.3.2 has "The best $k$ is then the number of data points, which is not helpful." Explain.

**9.8.** You cluster 1e6 data points with hierarchical k-means, using random subsampling and $k = 100$; roughly how many leaves to you expect? How deep do you expect the tree to be?

**9.9.** How would you model foreground pixels with a Gaussian for GrabCut? How good do you expect this model to be?

**9.10.** Assume you model foreground pixels with a histogram for GrabCut. What problems would zero counts create?

### LONGER PROBLEMS

**9.11.** You wish to maximise $\mathbf{w}^T\mathcal{A}\mathbf{w}$ as a function of $\mathbf{w}$.

**(a)** Why is this problem meaningless without a constraint on $\mathbf{w}$?

**(b)** Choose the constraint $\mathbf{w}^T\mathbf{w} = 1$. Write $\lambda$ for the associated Lagrange multiplier. Show that solving this problem is equivalent to solving $\mathcal{A}\mathbf{w} = \lambda\mathbf{w}$. What do you do about the constraint?

**(c)** Choose the constraint $\mathbf{w}^T\mathbf{w} = 1$. How many local maxima that satisfy the constraint are there? Do they present problems in practice?

**(d)** Choose the constraint $\mathbf{w}^T\mathbf{w} = 1$. How many local maxima that satisfy the constraint are there? Do they present problems in practice?

**(e)** Choose the constraint $\mathbf{1}^T\mathbf{w} = 1$. Show that if $\mathcal{A}$ is positive definite, you can obtain a solution by solving $\mathcal{A}\mathbf{x} = \mathbf{1}$. What could go wrong with your procedure if $\mathcal{A}$ was not positive definite? Can you link this problem to image segmentation?

**9.12.** Given a dataset of $N$ $d$-dimensional vectors; write $\{\mathbf{x}\}$ for the dataset and $\mathbf{x}_i$ for the $i$'th item. Compute the matrix of eigenvectors that solves

$$\mathsf{Covmat}(\{\mathbf{x}\})\mathcal{U} = \mathcal{U}\Lambda$$

for $\Lambda$ diagonal.

**(a)** Show that $\mathcal{U}^T\mathcal{U} = \mathcal{I}$ (the identity matrix).

**(b)** Transform the coordinates to obtain $\mathbf{r}_i = \mathcal{U}\mathbf{x}_i$. Show the covariance for the $\mathbf{r}_i$ is $\Lambda$.

**(c)** Show that the variance of the $k$'th component of $\mathbf{r}$ is $\Lambda_{ii}$.

**(d)** When you compute a distance function, why does it make sense to scale each direction by the standard deviation?

## PROGRAMMING EXERCISES

**9.13.** Given a dataset of $N$ $d$-dimensional vectors; write $\{\mathbf{x}\}$ for the dataset and $\mathbf{x}_i$ for the $i$'th item. Compute the matrix of eigenvectors that solves

$$\mathsf{Covmat}\left(\{\mathbf{x}\}\right)\mathcal{U} = \mathcal{U}\Lambda$$

for $\Lambda$ diagonal.

**(a)** Form the $s \times d$ matrix $\mathcal{P}_s$ consisting of the first $s$ rows of $\mathcal{U}^T$. Show the dataset $\{\mathbf{l}\}$, constructed using the rule

$$\mathbf{l}_i = \mathcal{P}_s \mathbf{x}_i$$

consists of $s$ dimensional vectors. Show their covariance (a) is diagonal and (b) has large values on the diagonal.

**(b)** Form the $s \times d$ matrix $\mathcal{P}_s$ consisting of the first $s$ rows of $\mathcal{U}^T$. Show the dataset $\{\mathbf{l}\}$, constructed using the rule

$$\mathbf{h}_i = \mathcal{P}_s^T \mathcal{P}_s \mathbf{x}_i$$

consists of $d$ dimensional vectors. Show their covariance (a) is diagonal; (b) has $s$ large values on the diagonal and (c) has $d - s$ zero values on the diagonal.

**(c)** Show that

$$\frac{1}{N}\sum_i \left[\mathbf{h}_i - \mathbf{x}_i\right]^T \left[\mathbf{h}_i - \mathbf{x}_i\right] = \sum_{u=s+1}^{u=d} \Lambda_{uu}.$$

Describe a method to choose $s$ based on this observation.

**9.14.** Use a GrabCut implementation to investigate the effect of $\gamma$ on the segmentation. I found this hard to do with OpenCV's implementation, but you might use the implementation at `https://github.com/luiscarlosgph/grabcut`.

**9.15.** Obtain a k-means implementation (I used the one in scikit-learn) and at least 20 images. Represent these images as RGB and as LAB.

**(a)** Now segment these images using only the color representation of the pixels. Does the choice of color representation affect segmentation significantly?

**(b)** Now attach $x$ and $y$ coordinate to the representation of each pixel. Investigate the effect of scaling these coordinates with respect to the color coordinates using a uniform scale. Can you find a scaling and choice of $k$ that gives reasonable segments?

**(c)** Can you obtain improved segmentations by using a two level hierarchical k-means? For the first clustering, cluster on color alone; then cluster the contents of each child on color and position.

**(d)** Obtain the Berkeley Segmentation Dataset, and use it to choose a segmentation procedure from the examples in this exercise. Use the evaluation protocol with code available at `https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html`. How does your choice compare to current best performers?

**9.16.** Obtain a normalized cuts implementation (I used the one at `https://ncut-pytorch.readthedocs.io/en/latest/`; I had to deal with minor drama installing it, but found it excellent) and at least 20 images. Represent these images as RGB and as LAB.

**(a)** Does the choice of color representation affect segmentation significantly?

**(b)** Now attach $x$ and $y$ coordinate to the representation of each pixel. Investigate the effect of scaling these coordinates with respect to the color coordinates using a uniform scale. Can you find a scaling and choice of $k$ that gives reasonable segments?

**(c)** Obtain the Berkeley Segmentation Dataset, and use it to choose a segmentation procedure from the examples in this exercise. Use the evaluation protocol with code available at `https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/resources.html`. How does your choice compare to current best performers?