CHAPTER

8

Classification

8.1 Objects, Patterns, and Statistics

Until now, the discussion has surrounded images and the operations that can be performed on them to enhance or otherwise modify them. The purpose of the modification is not really relevant in general; the result must simply meet some user-defined criteria of ''goodness.'' All this falls into the realm of *image processing* and is very common these days, especially when images are to be used by humans. Examples can be found in many forensic television programs such as *NCIS* and in motion pictures. A famous scene in *Blade Runner* depicts an investigator magnifying an image so that a reflection in a human eye yields digits that can be read. This is impossible with imaging technology available now, but it is an example of image processing. Another example is the processing of Hubble Space Telescope data that gives such wonderful color images.

Computer vision is more than that. It is the analysis of digital images so as to extract information automatically. The extracted information may be trivially simple, such as an answer to the question ''What color is this?'' or it may be much more complex, such as ''Whose face is this?''

Computer vision depends on having a good image with at least some known properties. Image processing is often used to enhance an image for further processing by vision algorithms, and sometimes there are known parameters of the camera and capture system that are used to permit more vision processing. Knowing the nature of the sensor and the lens, for example, can help to

determine absolute distances within images. However, at its heart, computer vision is about making measurements on images and/or determining what objects appear within those images.

Many people have difficulty understanding why this is a hard problem. After all, people recognize complex objects with apparent ease, and quickly. Why is this hard for computers? The answer is that computers use pixels to represent objects rather than some more natural representation that has more structure. Raster images are fundamentally two-dimensional and discrete, and are a poor way to represent an object. Figure 8.1 is an attempt to illustrate this.

255	255	255	26	26	26	21	21	21	4	4	4	5	5	5	5	5	5	3	3	3	7	7	7	16	16	16	14
14	14	5	5	5	12	12	12	11	11	11	7	7	7	8	8	8	9	9	9	23	23	23	10	10	10	29	29
29	16	16	16	11	11	11	8	8	8	15	15	15	9	9	9	11	11	11	17	17	17	53	53	53	228	228	228
255	255	255	19	19	19	10	10	10	10	10	10	4	4	4	4	4	4	12	12	12	10	10	10	11	11	11	11
11	11	18	18	18	15	15	15	12	12	12	14	14	14	14	14	14	21	21	21	27	27	27	64	64	64	165	165
165	135	135	135	56	56	56	15	15	15	15	15	15	29	29	29	20	20	20	21	21	21	42	42	42	137	137	137
255	255	255	227	227	227	1	1	1	5	5	5	4	4	4	4	4	4	11	11	11	6	6	6	14	14	14	29
29	29	110	110	110	65	65	65	188	188	188	166	166	166	74	74	74	112	112	112	221	221	221	205	205	205	238	238
238	236	236	236	213	213	213	68	68	68	18	18	18	19	19	19	14	14	14	15	15	15	8	8	8	19	19	19
255	255	255	0	0	0	6	6	6	8	8	8	2	2	2	9	9	9	9	9	9	124	124	124	207	207	207	217
217	217	232	232	232	228	228	228	234	234	234	243	243	243	246	246	246	236	236	236	243	243	243	243	243	243	241	241
241	244	244	244	243	243	243	234	234	234	39	39	39	15	15	15	11	11	11	16	16	16	9	9	9	10	10	10
254	254	254	18	18	18	5	5	5	3	3	3	0	0	0	78	78	78	187	187	187	212	212	212	218	218	218	231
231	231	231	231	231	237	237	237	235	235	235	236	236	236	241	241	241	236	236	236	241	241	241	244	244	244	240	240
240	238	238	238	239	239	239	237	237	237	188	188	188	36	36	36	26	26	26	14	14	14	9	9	9	17	17	17
254	254	254	0	0	0	4	4	4	8	8	8	24	24	24	110	110	110	186	186	186	207	207	207	219	219	219	221
221	221	229	229	229	232	232	232	233	233	233	233	233	233	231	231	231	238	238	238	239	239	239	235	235	235	240	240
240	236	236	236	231	231	231	232	232	232	186	186	186	33	33	33	33	33	33	2	2	2	31	31	31	5	5	5
22	22	22	7	7	7	5	5	5	4	4	4	42	42	42	71	71	71	184	184	184	211	211	211	211	211	211	225
225	225	232	232	232	232	232	232	235	235	235	236	236	236	236	236	236	239	239	239	241	241	241	241	241	241	234	234
234	233	233	233	227	227	227	226	226	226	220	220	220	25	25	25	17	17	17	16	16	16	20	20	20	10	10	10
0	0	0	4	4	4	9	9	9	9	9	9	39	39	39	78	78	78	139	139	139	202	202	202	217	217	217	215
215	215	232	232	232	234	234	234	228	228	228	233	233	233	232	232	232	238	238	238	241	241	241	243	243	243	234	234
234	234	234	234	227	227	227	223	223	223	105	105	105	51	51	51	25	25	25	18	18	18	16	16	16	20	20	20
37	37	37	13	13	13	4	4	4	17	17	17	49	49	49	87	87	87	131	131	131	208	208	208	222	222	222	222
222	222	226	226	226	231	231	231	232	232	232	238	238	238	234	234	234	238	238	238	240	240	240	240	240	240	232	232
232	227	227	227	226	226	226	204	204	204	200	200	200	22	22	22	19	19	19	16	16	16	17	17	17	11	11	11
246	246	246	15	15	15	9	9	9	0	0	0	38	38	38	104	104	104	153	153	153	225	225	225	238	238	238	233
233	233	233	233	233	233	233	233	233	233	233	231	231	231	236	236	236	245	245	245	242	242	242	237	237	237	235	235
235	227	227	227	217	217	217	219	219	219	171	171	171	14	14	14	17	17	17	5	5	5	31	31	31	18	18	18
0	0	0	8	8	8	5	5	5	12	12	12	50	50	50	97	97	97	174	174	174	154	154	154	225	225	225	235
235	235	236	236	236	233	233	233	213	213	213	220	220	220	229	229	229	233	233	233	232	232	232	239	239	239	235	235

Figure 8.1: The pixels in an image. What does the image represent?

The figure shows an array of numbers that represent an image. These are grey-level pixel values, and are really how the data is presented to the

computer and the vision algorithms. What is this image? What objects appear within it? From looking at the numbers, it is very hard to say. Figure 8.2 shows the image in a form that we can more easily process (greys) and it is plain from this that we have a picture of a face — Albert Einstein, in fact.

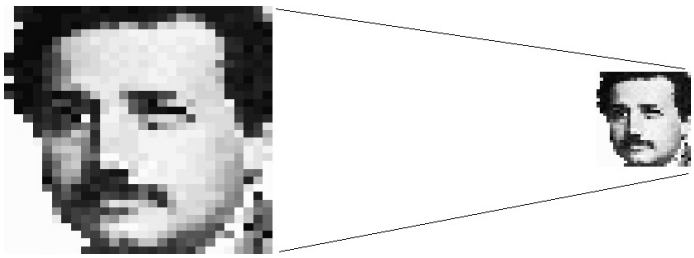

Figure 8.2: Rendering the data in Figure 8.1 as a grey-level image, we see that the 28x32 array of pixels is an image of Albert Einstein's face.

It is generally easier to decompose or parse a complex object into components than it is to synthesize low-level components into high-level complex objects or ideas. So, drawing objects into an image (i.e., rendering) is easier than the reverse, collecting pixels into related structures that represent objects. Computer vision is all about connecting the dots, literally: collecting pixels into logical structures that represent objects or portions of objects that can, in turn, be connected.

Some basic definitions are needed in order to clarify how computer vision will be carried out. Let's define an *object* as something, a two- or three-dimensional thing, that can appear in an image. *Object recognition* is the act of finding a collection of pixels in an image that represents an object being searched for, and the assigning of a label to the collection. The label is the name of the object. So, as a simple case, when a car is recognized in an image, an object recognition system will identify those pixels that are a part of the car and give it the label ''car.''

A *pattern* can be thought of as a combination of qualities (data) that form a characteristic arrangement. Patterns can occur in numbers, in pixels, sounds, or even behavior patterns. Patterns are important in computer vision because certain patterns of pixels represent specific objects in images. If those patterns can be detected, then it is an indication of the occurrence of that object in the picture. Sadly, characteristic patterns tend to be obscured by noise, scale, and orientation issues, lighting, and other practical matters. The usual method used within functioning vision systems is to collect simple patterns into low-level objects, which in turn are grouped into higher levels until the object can be built from the pieces. For instance, when trying to see faces in images, perhaps it is simpler to look for eyes or noses and try to build faces from these parts.

Because of the nature of the data (noisy and variable) and the difficulties in describing objects, vision algorithms are not perfect. Unlike, for example, a sorting algorithm, which must always yield a sorted collection of numbers upon demand, an object recognition algorithm usually works only sometimes. If a face recognizer can find 95% of the faces in an image, then 95% is the *success rate*, and it fails 5% of the time. Many things can cause failures, including shadows, motion blur, occlusion, noise, and scale problems. The developers of these systems generally know the success rates and often know the causes of failure, too. Thus, vision algorithms tend to be characterized by statistics, and frequently use statistics as an essential aspect of their function. Indeed, one of the most important methods of object recognition uses *statistical pattern recognition*, in which objects are characterized statistically using a set of measurements.

Now that some basic definitions have been presented, it is time for an essential aspect of computer vision to be made clear: *Vision systems are always looking for a specific set of objects*. This makes things a lot easier for the designer of the system. A vision system for counting cars on a freeway, for example, needs to be able to recognize vehicles, and perhaps people, roads, and static objects in the field of view. It does not have to recognize fish or camels, balloons, or chairs. If any of those objects appears in an image, the system will not identify them; indeed, these objects may interfere with the recognition of intended objects. The set of objects to be recognized is sometimes called the *domain* of the system or algorithm, and the behavior is to recognize an object or claim that none of the target objects appear within the image. Behavior outside of the domain, in other words, is not clearly defined.

8.1.1 Features and Regions

A crude but functional definition of a feature is *something that can be measured in an image*. A feature is therefore a number or a set of numbers derived from a digital image. The idea is that some objects belong to groups based on each of these measurements. Color is a simple feature, for example. It is a measurement (a determination of hue) and can, in fact, sometimes be used to recognize objects. For example, Figure 8.3 shows an image of a sample of carrots and tomatoes, carefully arranged to be isolated, but showing illumination effects and color differences. Vision problem: find the tomatoes.

Using color as a feature, we note that tomatoes are redder than carrots. Objects having a lot of red compared with other objects are more likely to be tomatoes than carrots (or peas). So, as a solution to the problem, it is possible to find isolated objects in the image, perhaps using an edge detector, and then count the red pixels within each object. In this example, it will work. The tomatoes can be isolated by thresholding hue at a value of 15/255. The result, as shown simply in Figure 8.4, is that all tomatoes can be found.

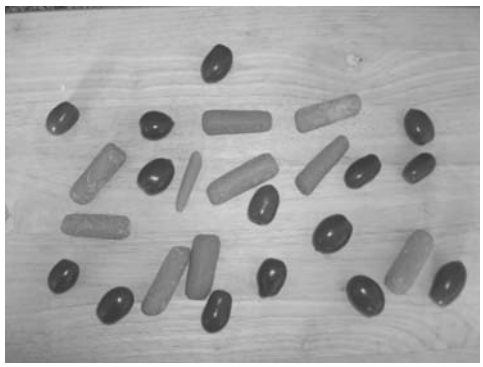

Figure 8.3: An image of tomatoes and carrots. Which are the tomatoes?

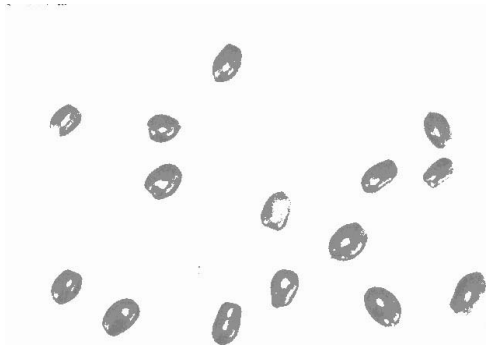

Figure 8.4: The tomatoes found by using hue as a feature.

Technically, this is still image processing and enhancement. A vision task would be to count the tomatoes, which can be done by counting the number of regions in the image of Figure 8.4. It does seem to show that hue can be used to solve the problem, though, and this is usually a part of the initial evaluation of a proposed solution.

Features are associated with image regions. An object within an image has a set of measurements (features) that can be used to characterize it. The initial task in object recognition vision problems is to isolate part of the image that might contain an object, measure its features, and determine which ones are useful in characterizing it.

The way that vision problems are approached usually has the following general form. A *target* is an object of the class being searched for; an example of what is to be recognized.

1. Find a way to isolate objects in the image that might be targets. This is often accomplished through edge detection, thresholding, region growing, or some other segmentation method.

2. Segmentation will create a simpler image in which targets are intermixed with other objects. Looking at the image, pick a feature that seems likely to characterize the targets.
3. Measure all potential targets using the proposed feature. Question: Is there a range of measured values that contain all the targets and no other objects? If so, this feature is the one to use.
4. If not all the targets can be found using this feature, record the percentage that can. Pick another feature and repeat from step 3.

It is possible to use more than one feature, so even if no one of them is perfect, we'll see that a set of merely acceptable ones can be just as good.

Let's use this scheme on the carrots/tomatoes problem. Isolation of the objects could be possible using edge detection or thresholding. Figure 5.1a shows a thresholded image, using Otsu's method (as in `thr_glh.c`). A surprising number of black regions are artifacts of the process, being neither carrots nor tomatoes. Most of these are tiny; some are too large. Area can be used to eliminate most of these, it would seem. Determining area requires a bit more processing.

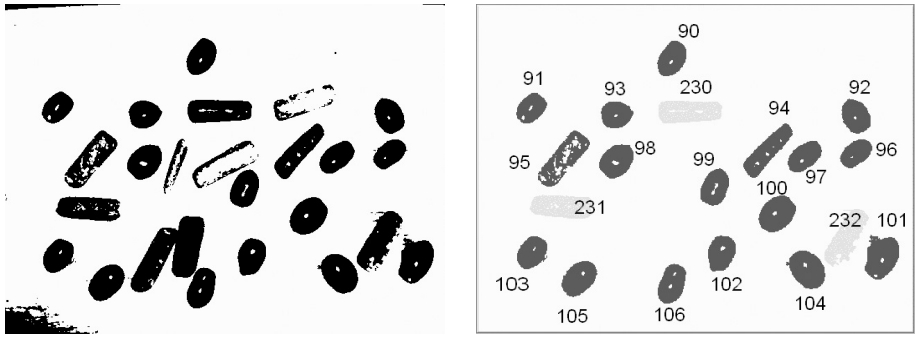

Figure 8.5: Marking potential targets in an image. (Left) A thresholded version of Figure 8.3. (Right) The regions that remain after selection using area. Each region is marked with a distinct grey level.

Each of the black areas in Figure 8.5a is separated from the others by regions of white pixels (background). A simple algorithm called a *flood fill* can identify isolated sets of connected black pixels; each of these could be a target (tomato). A basic recursive flood fill starts with a black pixel, and it recursively sets its black neighbors to another value (a *mark* value), and their black neighbors, and so on. In the present case, the target value (indicating an object, and to be replaced by a mark) is 0, and the mark value is anything other than 0 (black, object) and 255 (white, background). The result of a flood fill is that a region of

pixels that are connected to each other (a *connected region*) is given a particular grey level. The code for this is:

```
void flood (IMAGE img, int i, int j, int target, int replace)
{
       if (img->data[i][j] == target)
       {
             img->data[i][j] = 1;
             img->data[i][j] = replace;

             if (i+1 < img->info->nr) flood (img, i+1, j, target, replace);
             if (j-1 >= 0)            flood (img, i, j-1, target, replace);
             if (j+1 < img->info->nc) flood (img, i, j+1, target, replace);
             if (i-1 >= 0)            flood (img, i-1, j, target, replace);
       }
}
```

This particular implementation is slow and space consuming, but easy to code. Faster versions can no doubt be found on the Internet. The `flood` function requires that each pixel in the image be examined to see if it has the target value; if so, a region is grown around that *seed* pixel.

Winnowing the regions using area is simple. After a flood fill has marked a region with a new value M, the area of that region is simply the number of M-valued pixels in the entire image.

```
int area (IMAGE x, int c)
{
       int i,j,k=0;

       for (i=0; i<x->info->nr; i++)
             for (j=0; j<x->info->nc; j++)
                   if (x->data[i][j] == c) k++;
       return k;
}
```

If the area is too small (say, less than 900 pixels) or too big (perhaps 3100 pixels or bigger), then all the marked pixels are cleared (set to 255):

```
void clear (IMAGE x, int c)
{
       int i,j,k=0;

       for (i=0; i<x->info->nr; i++)
             for (j=0; j<x->info->nc; j++)
                   if (x->data[i][j] == c) x->data[i][j] = 255;
}
```

A final useful function would be `remark`, which will change the level of all pixels having one specific value to another one. The purpose is to uniquely mark a specified region. If there are a dozen regions on an image, it is useful

to know which ones are connected. Thus, we might mark the first connected region we find with the value 1, the second with 2, and so on.

```
void remark (IMAGE x, int oldg, int newg)
{
      int i,j,k=0;

      for (i=0; i<x->info->nr; i++)
            for (j=0; j<x->info->nc; j++)
                  if (x->data[i][j] == oldg)
                        x->data[i][j] = newg;
}
```

Now the first step — region identification — has the needed tools. The program `reg1.c` identifies the regions in the image that might be tomatoes and marks them with unique grey levels. Small regions are marked with values from 90 and higher, large regions are marked from 230 and higher. In the entire image, all pixels having a grey level of 90 belong to a single contiguous region, as do those having levels of 91, 92, and so on. Question: Does the region having level 90 correspond to a tomato?

So, we can return to the original issue: using color as a feature. Each pixel in the region-processed image corresponds to a color pixel in the original, and they both have the same coordinates. Pixels having a value of 90 in the region processed image can be examined in the original to see what their color is. Is there a similarity in hue within each region, and can that be used to classify the region as a tomato or a carrot? Yes.

8.1.2 Training and Testing

Let's look at the data. The image is scanned for pixels in the first region, which have the value 90. For each of these, fetch the corresponding pixel from the original color image and extract the RBG values. A mean for each color is computed, and associated with the region, and then the process is repeated for region 91, 92, and each other. Table 8.1 shows the results.

The values in the Area column are the number of pixels in each region. The Truth column is interesting and essential; it is the real classification of the object represented by the region. This was determined by a visual inspection of the image. A superficial examination of these data does not yield an obvious way to use them to determine perfectly which regions are carrots and which are tomatoes. If a chart is created showing the region class versus the region color, things become clearer.

Three scattergrams were created using the Microsoft Excel spreadsheet program. A *scattergram* shows data points or classes plotted against one or more features. Figure 8.6 shows the two classes, arbitrarily numbered 1 and 2, plotted against the value of each of the color components. This clarifies the situation rather well.

Table 8.1: Features for Classifying Vegetables

MARK	AREA	RED	GREEN	BLUE	TRUTH
90	1634	138	46	52	Tomato
91	1384	152	53	60	Tomato
92	1634	130	54	49	Tomato
230	2663	143	104	74	Carrot
93	1452	127	47	48	Tomato
94	2273	181	105	81	Carrot
95	2364	179	110	83	Carrot
96	1374	136	55	53	Tomato
97	1580	132	46	53	Tomato
98	1834	133	47	49	Tomato
99	1658	149	59	63	Tomato
231	2581	174	107	83	Carrot
100	2137	134	46	52	Tomato
232	2721	180	104	78	Carrot
101	2417	138	56	57	Tomato
102	1662	129	46	55	Tomato
103	1599	125	46	50	Tomato
104	2253	132	45	51	Tomato
105	1933	123	42	46	Tomato
106	1789	131	44	51	Tomato

The value of the green component is very useful for distinguishing the two classes, according to this figure. A vertical line can be drawn at green = 85 that has tomatoes on the left (smaller green value) and carrots on the right (larger green value). This means that for this image, tomato regions have a green component less than 85, and thus can be distinguished from carrots using a simple threshold. The same appears to be true for blue, although the gap between the two classes is smaller, and the threshold is about 70. The red component is most difficult to use as a feature, possibly because red and orange both contain significant red components. The tomatoes have red values between 110 and 170, and so require two thresholds to classify the objects. Figure 8.7 shows a classification of the regions according the blue component, as just described.

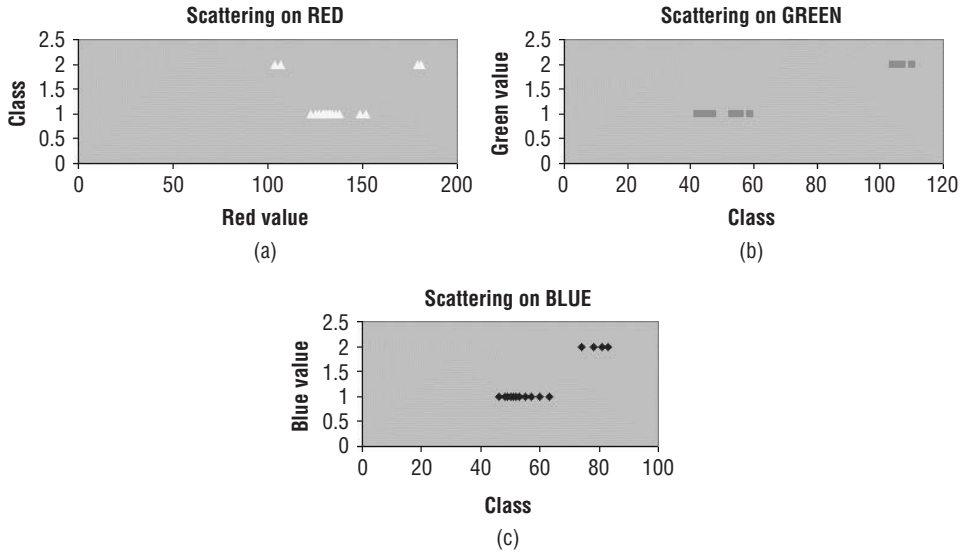

Figure 8.6: Scattergrams showing the connection between object class and color.

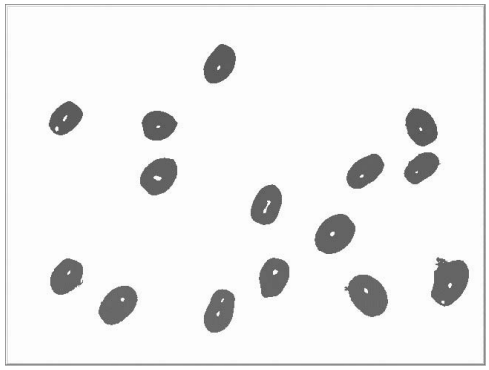

Figure 8.7: Tomatoes classified using the blue color component only. All tomatoes are found; no errors are encountered.

According to these scattergrams, only one color feature is necessary to distinguish between a tomato and a carrot. However, this is based on only one image. There would be little point in creating a vision system to analyze one image, so an assumption is that the classifier will be applied to a large set of images, one after the other, in some industrial or other real-world setting. So, it would seem that there are issues to be considered: how much data is needed to experiment with, how the values of features vary across images, and how to find features that can be used, individually or in combination, to classify an object in an image.

It is standard practice to measure and classify a set of data to establish a normal range for the features to be used in automatic classification. This is what is referred to as *training*, and it is an essential part of building a recognition

system for visual objects. The system *learns* — that is, establishes ranges for the features that were selected for use — by being given a known object and then identifying some pattern among the feature values. This works better if a large number of objects and images are used in the training process, and that means having a large set of classified objects on hand before the system is even completed. This is called *training data*. It may turn out that some of the selected features are not useful and will need to be discarded or replaced, and this will require modifications to the system.

So, for each object in each test image, all the proposed features are measured and stored. A classifier is built that uses these features to determine the class of the objects as well as can be done. Rarely will this be perfect, but it could be perfect for the training data. Finding out the actual rate of successful classification must be done using a different set of data, not the training data, because the system has been tuned specifically to recognize the training data set. We must know the actual classifications for the test data, too, since we need to determine how often the system returns the correct class. If we have 100 objects that are of known classes, then this set of data needs to be split into two sets: one for training, one for testing. For the time being, they should be split into two equal parts, but alternatives will be described starting in the next section and in the remainder of the chapter.

8.1.3 Variation: In-Class and Out-Class

Part of the problem with visual classification is that objects do not look the same in different images, in different orientations, and when seen through different cameras. Examining the data for the carrot problem, it is easy to see that tomatoes have a variety of different values for each of the features we have measured so far: color (red, green, blue) and area. Indeed, no two tomatoes have the same values for these four features. Consider the green component of tomato regions: the values in each row of the following table are an average of the green components of all pixels in the region, meaning that even within each region they are not all the same. The means over the various regions are not the same either.

	GREEN	AREA
MEAN OVER ALL TOMATOES	48.80	1756.00
STANDARD DEVIATION	4.95	299.79
MEAN OVER ALL CARROTS	106.0	2520.40
STANDARD DEVIATION	2.28	173.16

Samples of features such as these usually follow a statistical normal distribution. This is the famous bell-shaped curve, where the mean is in the center and the standard deviation specifies the width of the bell. The variation of the measurements is greater if the bell is wide, of course. A narrow range of

values, or a small standard deviation, is desired because it corresponds to an easier thresholding problem. It would also mean that the feature values would be less likely to overlap with those of other objects. A large distance between means of classes to be separated is important, too.

The situation of Figure 8.8a is a desirable one for a classification problem. Here, classes P and Q have very distinct means and a relatively small standard deviation, and so the feature values involved have a very small region where they can overlap. In this region it is not possible to accurately identify the class of the object from this feature. The situation of Figure 8.8b is much worse, because the means of the two distributions are closer together and the area of overlap is larger. There will be a greater proportion of measurements that fall into this ambiguous area. The best threshold to use is the feature value that corresponds to the point of intersection of the two normal curves, but in Figure 8.8b it seems certain this will not yield a correct classification in all cases.

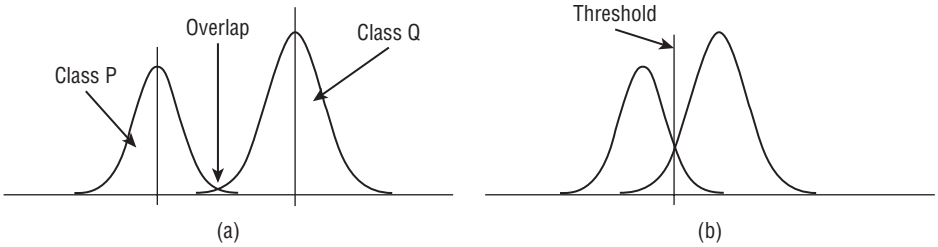

Figure 8.8: (a) The distribution of feature values between two classes, P and Q. The overlap between these distributions is small, meaning that this feature alone can distinguish between these classes. (b) A larger overlap area increases the number of feature measurements that are ambiguous. The vertical line here shows the location of the likely best threshold.

If one feature does not distinguish between the classes, then perhaps two will. As an example, let's use a classic set of data from many years ago, the *Iris* data set [Fisher, 1936; Anderson, 1935]. These data appear in Table 8.2 as numbers, and we'll not be concerned here with how the measurements were obtained. The interesting thing is how the data can be used to distinguish between three species of Iris: *setosa*, *versicolor*, and *virginica*. The measurements are width and length of petals and sepals, which are anatomical features of any flower, as illustrated in Figure 8.9a.

That no single feature can be used to classify all instances into a correct category can be established using scattergrams, or even by examining the data. Which combination is best is a harder question to answer, and how to tell is an interesting process to observe. Plotting pairs of features is useful in this case, and showing the class of the object as color in the scattergram gives effectively a third dimension to the plot, as shown in Figure 8.9b. Note that a straight line can be drawn that separates the red class (*setosa*) from the blue (*versicolor*), but no such line exists between the blue and the green (*virginica*).

Table 8.2: The Iris data set.

SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS
5.1	3.5	1.4	0.2	*setosa*	7.0	3.2	4.7	1.4	*versicolor*	6.3	3.3	6.0	2.5	*virginica*
4.9	3.0	1.4	0.2	*setosa*	6.4	3.2	4.5	1.5	*versicolor*	5.8	2.7	5.1	1.9	*virginica*
4.7	3.2	1.3	0.2	*setosa*	6.9	3.1	4.9	1.5	*versicolor*	7.1	3.0	5.9	2.1	*virginica*
4.6	3.1	1.5	0.2	*setosa*	5.5	2.3	4.0	1.3	*versicolor*	6.3	2.9	5.6	1.8	*virginica*
5.0	3.6	1.4	0.2	*setosa*	6.5	2.8	4.6	1.5	*versicolor*	6.5	3.0	5.8	2.2	*virginica*
5.4	3.9	1.7	0.4	*setosa*	5.7	2.8	4.5	1.3	*versicolor*	7.6	3.0	6.6	2.1	*virginica*
4.6	3.4	1.4	0.3	*setosa*	6.3	3.3	4.7	1.6	*versicolor*	4.9	2.5	4.5	1.7	*virginica*
5.0	3.4	1.5	0.2	*setosa*	4.9	2.4	3.3	1.0	*versicolor*	7.3	2.9	6.3	1.8	*virginica*
4.4	2.9	1.4	0.2	*setosa*	6.6	2.9	4.6	1.3	*versicolor*	6.7	2.5	5.8	1.8	*virginica*
4.9	3.1	1.5	0.1	*setosa*	5.2	2.7	3.9	1.4	*versicolor*	7.2	3.6	6.1	2.5	*virginica*
5.4	3.7	1.5	0.2	*setosa*	5.0	2.0	3.5	1.0	*versicolor*	6.5	3.2	5.1	2.0	*virginica*
4.8	3.4	1.6	0.2	*setosa*	5.9	3.0	4.2	1.5	*versicolor*	6.4	2.7	5.3	1.9	*virginica*
4.8	3.0	1.4	0.1	*setosa*	6.0	2.2	4.0	1.0	*versicolor*	6.8	3.0	5.5	2.1	*virginica*
4.3	3.0	1.1	0.1	*setosa*	6.1	2.9	4.7	1.4	*versicolor*	5.7	2.5	5.0	2.0	*virginica*
5.8	4.0	1.2	0.2	*setosa*	5.6	2.9	3.6	1.3	*versicolor*	5.8	2.8	5.1	2.4	*virginica*
5.7	4.4	1.5	0.4	*setosa*	6.7	3.1	4.4	1.4	*versicolor*	6.4	3.2	5.3	2.3	*virginica*
5.4	3.9	1.3	0.4	*setosa*	5.6	3.0	4.5	1.5	*versicolor*	6.5	3.0	5.5	1.8	*virginica*
5.1	3.5	1.4	0.3	*setosa*	5.8	2.7	4.1	1.0	*versicolor*	7.7	3.8	6.7	2.2	*virginica*
5.7	3.8	1.7	0.3	*setosa*	6.2	2.2	4.5	1.5	*versicolor*	7.7	2.6	6.9	2.3	*virginica*
5.1	3.8	1.5	0.3	*setosa*	5.6	2.5	3.9	1.1	*versicolor*	6.0	2.2	5.0	1.5	*virginica*
5.4	3.4	1.7	0.2	*setosa*	5.9	3.2	4.8	1.8	*versicolor*	6.9	3.2	5.7	2.3	*virginica*
5.1	3.7	1.5	0.4	*setosa*	6.1	2.8	4.0	1.3	*versicolor*	5.6	2.8	4.9	2.0	*virginica*
4.6	3.6	1.0	0.2	*setosa*	6.3	2.5	4.9	1.5	*versicolor*	7.7	2.8	6.7	2.0	*virginica*
5.1	3.3	1.7	0.5	*setosa*	6.1	2.8	4.7	1.2	*versicolor*	6.3	2.7	4.9	1.8	*virginica*
4.8	3.4	1.9	0.2	*setosa*	6.4	2.9	4.3	1.3	*versicolor*	6.7	3.3	5.7	2.1	*virginica*

Continued

Table 8.2: (*continued*)

SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS	SEPAL LENGTH	SEPAL WIDTH	PETAL LENGTH	PETAL WIDTH	CLASS
5.0	3.0	1.6	0.2	*setosa*	6.6	3.0	4.4	1.4	*versicolor*	7.2	3.2	6.0	1.8	*virginica*
5.0	3.4	1.6	0.4	*setosa*	6.8	2.8	4.8	1.4	*versicolor*	6.2	2.8	4.8	1.8	*virginica*
5.2	3.5	1.5	0.2	*setosa*	6.7	3.0	5.0	1.7	*versicolor*	6.1	3.0	4.9	1.8	*virginica*
5.2	3.4	1.4	0.2	*setosa*	6.0	2.9	4.5	1.5	*versicolor*	6.4	2.8	5.6	2.1	*virginica*
4.7	3.2	1.6	0.2	*setosa*	5.7	2.6	3.5	1.0	*versicolor*	7.2	3.0	5.8	1.6	*virginica*
4.8	3.1	1.6	0.2	*setosa*	5.5	2.4	3.8	1.1	*versicolor*	7.4	2.8	6.1	1.9	*virginica*
5.4	3.4	1.5	0.4	*setosa*	5.5	2.4	3.7	1.0	*versicolor*	7.9	3.8	6.4	2.0	*virginica*
5.2	4.1	1.5	0.1	*setosa*	5.8	2.7	3.9	1.2	*versicolor*	6.4	2.8	5.6	2.2	*virginica*
5.5	4.2	1.4	0.2	*setosa*	6.0	2.7	5.1	1.6	*versicolor*	6.3	2.8	5.1	1.5	*virginica*
4.9	3.1	1.5	0.2	*setosa*	5.4	3.0	4.5	1.5	*versicolor*	6.1	2.6	5.6	1.4	*virginica*
5.0	3.2	1.2	0.2	*setosa*	6.0	3.4	4.5	1.6	*versicolor*	7.7	3.0	6.1	2.3	*virginica*
5.5	3.5	1.3	0.2	*setosa*	6.7	3.1	4.7	1.5	*versicolor*	6.3	3.4	5.6	2.4	*virginica*
4.9	3.6	1.4	0.1	*setosa*	6.3	2.3	4.4	1.3	*versicolor*	6.4	3.1	5.5	1.8	*virginica*
4.4	3.0	1.3	0.2	*setosa*	5.6	3.0	4.1	1.3	*versicolor*	6.0	3.0	4.8	1.8	*virginica*
5.1	3.4	1.5	0.2	*setosa*	5.5	2.5	4.0	1.3	*versicolor*	6.9	3.1	5.4	2.1	*virginica*
5.0	3.5	1.3	0.3	*setosa*	5.5	2.6	4.4	1.2	*versicolor*	6.7	3.1	5.6	2.4	*virginica*
4.5	2.3	1.3	0.3	*setosa*	6.1	3.0	4.6	1.4	*versicolor*	6.9	3.1	5.1	2.3	*virginica*
4.4	3.2	1.3	0.2	*setosa*	5.8	2.6	4.0	1.2	*versicolor*	5.8	2.7	5.1	1.9	*virginica*
5.0	3.5	1.6	0.6	*setosa*	5.0	2.3	3.3	1.0	*versicolor*	6.8	3.2	5.9	2.3	*virginica*
5.1	3.8	1.9	0.4	*setosa*	5.6	2.7	4.2	1.3	*versicolor*	6.7	3.3	5.7	2.5	*virginica*
4.8	3.0	1.4	0.3	*setosa*	5.7	3.0	4.2	1.2	*versicolor*	6.7	3.0	5.2	2.3	*virginica*
5.1	3.8	1.6	0.2	*setosa*	5.7	2.9	4.2	1.3	*versicolor*	6.3	2.5	5.0	1.9	*virginica*
4.6	3.2	1.4	0.2	*setosa*	6.2	2.9	4.3	1.3	*versicolor*	6.5	3.0	5.2	2.0	*virginica*
5.3	3.7	1.5	0.2	*setosa*	5.1	2.5	3.0	1.1	*versicolor*	6.2	3.4	5.4	2.3	*virginica*
5.0	3.3	1.4	0.2	*setosa*	5.7	2.8	4.1	1.3	*versicolor*	5.9	3.0	5.1	1.8	*virginica*

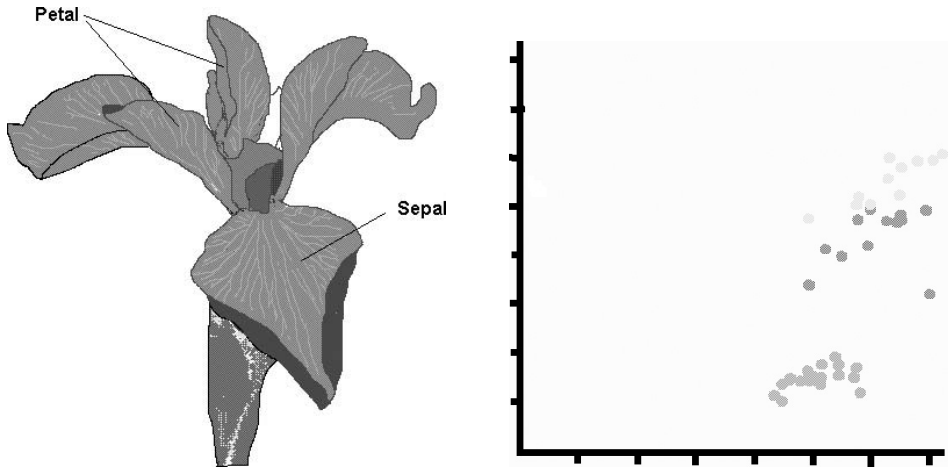

Figure 8.9: (a) The anatomy of a flower, showing the petals and sepals that are key to the Iris data set. (b) A scattergram of Sepal length vs. petal length for the three classes. Color codes the classes; note the spatial groupings.

A line breaks the green-blue region into two parts such that almost all green points are on one side and almost all blue points are on the other. This could be used to distinguish between the two classes with a small error. The line that does this is not horizontal, but that does not matter. This is called a *linear discriminant* and is commonly used in data classification and machine learning. There are many references to this technique in the literature. It is, of course, just one of many possible methods for classifying data.

8.2 Minimum Distance Classifiers

Looking again at the scattergram of Figure 8.9b, note that the data are grouped into two-dimensional regions such that it is possible to draw a curve that surrounds each class. Of course, such a curve can get very complex, and the curve would only surround the points we knew about. A new object and set of measurements may lie well outside of the curve. If an unknown object is measured and if the measurements form a point that falls inside that curve, then it probably should be classified with the others within the curve.

Because the curve is too complex to identify and hard to use as a classifier, we can introduce a simpler scheme: an unidentified region that is classified according to how far away it is (as a point) from any of the other points in the training set. Depending on how ''how far away'' is defined, this could work pretty well. This is what is commonly known as *distance*, and there are several reasonable ways to define and implement it.

8.2.1 Distance Metrics

The common, intuitive definition of distance is called *Euclidean distance*, because of Euclid's connection with many other common geometric concepts. It should be (and was in the past) called the *Pythagorean distance* because it uses the famous formula for the hypotenuse. The distance between a point $P = (p_1, p_2)$ and a point $Q = (q_1, q_2)$ is:

$$d = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2} \qquad \text{(EQ 8.1)}$$

For points in a space having more than two dimensions, say N dimensions, this formula generalizes as:

$$((p_1 - q_1)^2 + (p_2 + q_2)^2 + \cdots + p_N + q_N)^2)^{\frac{1}{2}} = \sqrt{\sum_{i=1}^{N}(p_i - q_i)^2} \qquad \text{(EQ 8.2)}$$

This is the distance ''as the crow flies,'' and while it makes sense in everyday life, there are problems with it in images. The main one is that pixel locations are integers, whereas the distance between pixels can be floating point. Another practical problem is that this calculation requires a square root operation, which is likely to take a hundred times longer to calculate than a simple integer operation. It is true that computers are faster than they used to be, but images have gotten bigger, too. Therefore, it is usual to omit the square root and work with d^2 whenever possible.

A commonly used distance measure when using pixels is the 8-distance. This is the maximum of the horizontal and vertical difference between the coordinates of the pixel; or, for the previously defined P and Q:

$$d_8 = max(|p_1 - q_1|, |p_2 - q_2|) \qquad \text{(EQ 8.3)}$$

One way to think of this is as the number of pixels between P and Q. It is called *8-distance* because the path traced between P and Q uses the eight discrete directions that are possible on a discrete grid.

If there is an 8-distance, then why not a 4-distance? There is, and it is also called *Manhattan distance* or *city block distance*. It is the distance in pixels between P and Q using only up/down and left/right directions (4 connected pixels). Mathematically:

$$d_4 = |p_1 - q_1| + |p_2 - q_2| \qquad \text{(EQ 8.4)}$$

Finally, at least for the purposes here, there is the most exotic, complex, and useful distance measure: *Mahanalobis distance*. It is difficult to explain in the general case, but for specific examples in classification it is more obvious. Consider the data in Table 8.1 again. If P and Q are the first two

entries, tomatoes both, and the area and the green component are used in a classification, then the data points are:

$$P = (1634, 46)$$
$$Q = (1384, 53)$$

The Euclidean distance between these two is:

$$\sqrt{(1634 - 1384)^2 + (46 - 53)^2} = \sqrt{62500 + 49} - 250.1$$

Now change the green component of P by 1 to (1634, 45). The distance between P and Q is now 250.13. Changing the area component by 1 so that $P = (1635, 46)$ changes the $P - Q$ distance to 251.1. This shows that a change in the first coordinate makes a bigger difference in the distance than does a change in the second. Or in other words, the scales of the two coordinate axes are different. This is very common in computer vision problems, and it really does make sense. Why would we expect that each of the measurements would have units of the same size?

Normalizing with respect to scale can be done using statistics. The standard deviation is a measure of variability, or what the range of values is. Dividing sample values by the standard deviation should narrow the range of values, and convert the units to universal ones. This is the basic idea behind Mahanalobis distance. For example, consider the same points P and Q as before and the normalized points P' and Q'. The overall standard deviations are:

$$s_{\text{area}} = 429.5 \qquad s_{\text{green}} = 25.2$$

The points are:

$$P = (1634, 46) \qquad Q = (1384, 53) \qquad \text{distance}\,(P, Q) = 250.1$$
$$P' = (3.8, 1.83) \qquad Q' = (3.2, 2.1) \qquad \text{distance}\,(P', Q') = 0.64$$

The standard deviations are used to normalize the raw sample values before computing distance. It's actually more complex than that; reality tends to make the math harder. The formula for computing the Mahanalobis distance between P and Q is:

$$d_M(P, Q) = \sqrt{(P - Q)^T S^{-1} (P - Q)} \qquad \text{(EQ 8.5)}$$

which is a matrix equation, in which P and Q are the points (vectors) for which the distance is being computed, $(P - Q)^T$ is the transpose of the difference of the vectors, and S is the *covariance matrix*.

The variance is the mean of the squared distances between a value and the mean of those values:

$$VAR = \frac{\sum_{i=1}^{n} (P_i - \mu_i)(P_i - \mu_i)}{n - 1} \qquad \text{(EQ 8.6)}$$

When two (or more) values are involved, this calculation can include combinations of the variables. In the case of P and Q:

$$COV = \frac{\sum_{i=1}^{n}(P_i - \mu_i)(Q_i - \mu_i)}{n-1} \quad \text{(EQ 8.7)}$$

So, covariance is a generalization of variance for multiple variables. The Mahanalobis distance is much more computationally expensive than the other distance measures, but it does have the important advantage of being scale independent, so is often used. However, for simplicity many people use Euclidean distance, too, and without loss of generality most of the rest of the examples will use Euclidean distance. Any distance measure may be substituted, of course.

8.2.2 Distances Between Features

Many pattern recognition tasks use a large number of features to distinguish between many classes. The Iris data set has four features, which is too many to visualize in a straightforward way, to characterize three classes. This data set will be used to illustrate distance-based classifiers, starting with the *nearest neighbor* classifier.

Given N classes $C_1, C_2, \ldots, C_N$ and M features $F_1 .. F_M$, consider the classification of an object, P. Measure all features for this object and create an M-dimensional vector, v, from them. Feature vectors for all objects in all N classes have also been created; the first such in class C_1 will be ${C_1}^1$, the eighth one in class 3 will be ${C_3}^8$, and so on. Classification of P by the nearest neighbor method involves calculating the distances between v and all feature vectors for all the classes. The class of the feature vector having the minimum distance from v will be assigned to v.

The name of the method is very descriptive. The class of an unknown target will be the same as that of its nearest neighbor in feature space. Let's see how this works using the Iris data set. First, the set needs to be broken into training data and test data: select the first half of the data for each class to be training data, and the last half as test data.

Next, feature vectors are created from the training data items. There are four features, so each vector has four components. This vector is compared against (i.e., the distance is computed to) all the training data vectors, and the class of the one with smallest distance is saved: this will be the class given to the target. This is done for each of the test data items, and success rates are computed; the raw success rate, the number of correct classifications divided by the number of test data items, is a good indicator of how good the features are and of how well the classifier will work overall.

There is another, better, way to evaluate the results. A *confusion matrix* is a table in which each column represents an actual class, and each row represents a class delivered by the classifier (an outcome). For the Iris data experiment, the confusion matrix is:

	SETOSA	VERSICOLOR	VIRGINICA
SETOSA	25	0	0
VERSICOLOR	0	24	3
VIRGINICA	0	1	22

The columns add up to the number of elements in each class, and the rows add up to the number of classifications that the classifier made to each class. The trace (sum of the elements along the diagonal) is the number of correct classifications, and the success rate is the trace divided by the total number of trials. In this instance, the success rate is nearly 95%, which is pretty good.

The nearest neighbor method is commonly implemented for a classifier because it is simple and gives pretty good results. However, if one neighbor gives good results, why not use many neighbors? This simple thought leads to the *k-nearest neighbor* method, in which the class is determined by a vote between the nearest k neighbors in feature space. This is a little more work, and can lead to ties in some cases. There are a two main ways to implement this: compute all distances and then sort them into descending order and read off the smallest k of them, or keep only the smallest k in a table and test/insert after every distance calculation. The example program provided (`nkn.c`) uses the first method. This allows the specification of k to change without much modification of the program so that the effect of changes to k can be explored.

The k-nearest neighbor algorithm should yield the same results as nearest neighbor for $k = 1$; this is a test of correctness. The results for the Iris data are as follows:

K	SUCCESS	K	SUCCESS
1	95%	12	93%
2	92%	13	95%
3	93%	14	93%
4	95%	15	93%
5	92%	16	95%
6	92%	17	96%

Continued

(*continued*)

K	SUCCESS	K	SUCCESS
7	93%	18	96%
8	95%	19	95%
9	95%	20	95%
10	93%	21	95%
11	93%	22	92%

The success of the *k*-nearest neighbor method depends on the way the data points are scattered near the overlap areas. In this case, it seems no better than the simple nearest neighbor method, but this is hard to predict in general, and it will be better sometimes.

The *nearest centroid method* uses many points as a basis for comparison, but it combines this with an ease of calculation that makes it attractive. The *centroid* is the point in a set of feature data that is in some sense the mean value. This point is a good representation of the entire set if any such place exists. The coordinates of the centroid are the mean values of the coordinates of all the points in the set; that is, the first coordinate of the centroid is the mean of all the first coordinates, and so on. For the Iris data set, this means that there are three centroids, one for each set. They are:

- Centroid 1 = (5.028000, 3.480000, 1.460000, 0.248000)
- Centroid 2 = (6.012000, 2.776000, 4.312000, 1.344000)
- Centroid 3 = (6.576000, 2.928000, 5.639999, 2.044000)

So, the nearest centroid classifier computes the distance between the sample point and the centroids, and the centroid at the smallest distance represents the classification. This has fewer computations at classification time, because the centroids are pre-computed and there is a need for only one distance calculation per class.

The results of the nearest centroid classifier for the Iris data set are precisely the same as for the nearest neighbor classifier. This will not be true for all data sets.

8.3 Cross Validation

Splitting the data sets into training and testing sets is necessary to avoid getting inflated success rates. One would expect high success on the data used for training. In the nearest neighbor classifier, for example, the success rate

on the training data should be 100%, because each of the points will be a distance of zero from at least one other — itself. Still, the selection of training versus test data is arbitrary, and the two data sets could be exchanged without distorting the results. If this is done for the Iris data using the nearest neighbor classifier, the results become as follows:

	SETOSA	VERSICOLOR	VIRGINICA
SETOSA	25	0	0
VERSICOLOR	0	23	2
VIRGINICA	0	2	23

The success rate is the same as before, but the details of the confusion matrix are different. Repeating the classification with the roles of the testing and training sets reversed gives us two different trials, though, and should give us more confidence, especially since there is relatively little data here. This process could be described as a 2-way (or 2-fold) cross validation.

The general description of cross validation is a process for partitioning data repeatedly into distinct training and testing sets. There are many ways to do this, some of them wrong. Any partition that uses the same samples in both sets would normally be in error, for example, and creating new data points based on statistical samples may in some instances be fine, but is not cross validated. Cross validation takes the data that exists and partitions it into training/testing sets multiple times so that the sets are different.

An *n-way cross validation* breaks the data into *n* more-or-less equal parts. Then each of these in turn is used as test data, while all the other parts together are used as training data. This gives *n* results, and the overall result is the average of those *n*. The Iris data set has 150 samples in all, so a 5-way cross validation would provide a convenient partitioning into 5 groups of 30 points each. There is no rule that says there have to be exactly the same number of samples in each set, although there should be as many examples of each class as possible.

The program `cross5.c` works the same way as the nearest neighbor program, except that it reads all the Iris data into one large array at the beginning and then partitions it before each experiment. The result is five distinct experiments with five confusion matrices and success rates:

	PARTITION 1	PARTITION 2	PARTITION 3	PARTITION 4	PARTITION 5
SUCCESS	96.7	96.7	93.3	93.3	100.0

This yields an average of 96%.

Cross validation can be done using random samples of the data, too. A test set would be built from random selections of the full data set, making sure

not to choose the same item more than once. All the items not selected will be the training set. In principle, this can be repeated arbitrarily many times, but nothing is gained by doing so. Between 5 and 10 trials would be sufficient for the Iris data set. Using random cross validation, keeping the classes balanced, and with 10 examples from each class in the test set, and overall success rate averaged over ten trials, a 93% success rate was obtained. This would be a little different each time due to the random nature of the experiment.

What might be called the ultimate in cross validation picks a single sample from the entire set as test data, and uses the rest as training data. This can be repeated for each of the samples in the set, and the average over all trials gives the success rate. For the Iris data, there would be 150 trials, each with a single classification. This is called *leave-one-out cross validation*, for obvious reasons.

For the Iris set again, leave-one-out cross validation leads to an overall success rate of 96% when used with a nearest neighbor classifier; it's probably the best that can be done. This is a good technique for use with smaller data sets, but is really too expensive for large ones.

8.4 Support Vector Machines

Section 8.1.3 discussed the concept of a linear discriminant. This is a straight line that divides the feature values into two groups, one for each class, and is an effective way to implement a classifier if such a line can be found. In higher dimensional spaces — that is, if more than two features are involved — this line becomes a plane or a hyperplane. It's still linear, just complicated by dimensionality. Samples that lie on one side of the plane belong to one class, while those on the other belong to a different class. A *support vector machine* (SVM) is a nitro-powered version of such a linear discriminant.

There are a couple of ways in which an SVM differs from simpler linear classifiers. One is in the fact that an SVM attempts to optimize the line or plane so that it is the *best* one that can be used. In the situation illustrated in Figure 8.10 there are two classes, white and black. Any of the lines shown in 8.10a will work to classify the data, at least the data that is seen there. New data could change the situation, of course. Because of that it would be good to select the line that does the best possible job of dividing the plane into the two areas occupied by the two classes. Such a line is shown in Figure 8.10b. The heavy dark line is the best line, and the thin lines on each side of it show the space between the two classes — the heavy line divides this space evenly into two parts, giving a maximum *margin* or distance between the groups. The point of an SVM is to find the maximum margin hyperplane. A line divides two-dimensional data into two parts; a plane divides three-dimensional data into two parts; and a hyperplane is a linear function that divides *N*-dimensional data into two parts. The maximum margin hyperplane is always as far from both data sets as possible.

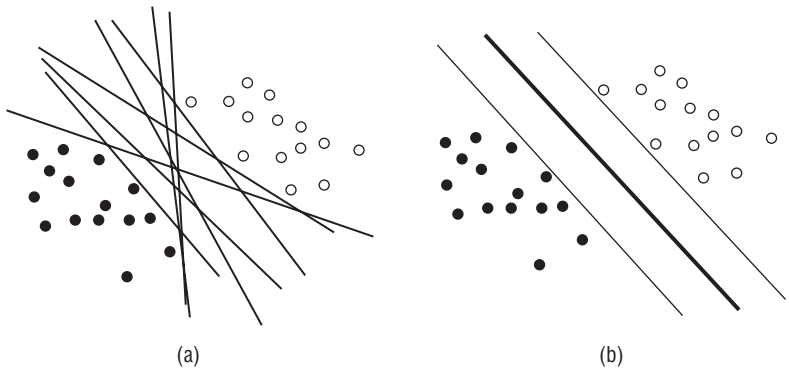

Figure 8.10: (a) A collection of straight lines that separate two classes. (b) The best line, or maximum margin line/plane/ hyperplane. The white area between the classes is the margin.

Finding a maximum or minimum margin is an optimization problem, and there are many methods for solving these [Bunch, 1980; Fletcher, 1987; Kaufman, 1998; Press, 1992], but they are beyond the scope of the present discussion. It suffices to say that it can be done. The basic idea, though, is to use feature vectors on the convex hull of the data sets as candidates to be used to guide the optimization. The candidates are called *support vectors* and are illustrated, along with the convex hulls for the data sets, in Figure 8.11. The support vectors completely define the maximal margin line, which is the line that passes as far as possible from all three of those vectors. There can be more than three support vectors, but not fewer.

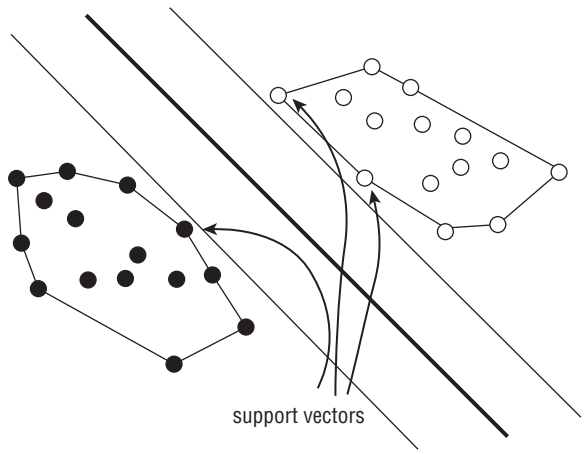

Figure 8.11: The convex hull of the feature vectors for the two classes, and the three support vectors for the final maximal margin line.

Support vector machines can also find non-linear boundaries between classes, which is their other major advantage over other methods. This is not

done by finding curved or piecewise linear paths between the feature vectors of each class, but in fact is accomplished by transforming those feature vectors so that a linear boundary can be found. A simple and clear example of this situation can be seen in Figure 8.12a, where the vectors of one class completely surround those of the other. It is obvious that there is no line or plane that can divide these vectors into the two classes.

A transformation of these vectors can yield a separable set. The vectors shown are in two dimensions; they lie in a plane. If we add a dimension and transform the points appropriately into a third dimension, a plane can be found that divides the classes (Figure 8.12b). The data has been projected into a different, higher dimensional feature space. In SVM parlance, this transformation uses a *kernel*, which is the function that projects the data. There are many possible kernels; Figure 8.12 shows the result of using a Gaussian (a *radial basis* function), but polynomials and other functions can be used, depending on the data.

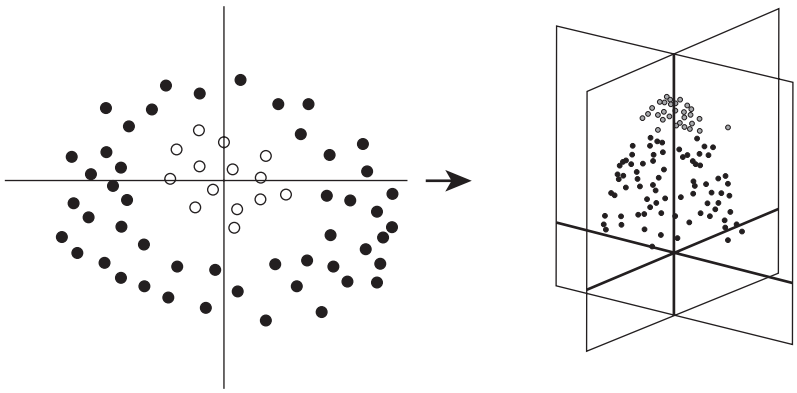

Figure 8.12: (a) Feature vectors for two classes that cannot be separated linearly. (b) The same vectors after being projected into a third dimension using a radial basis function. The can now be separated using a plane.

So, the points near the origin are given a larger value in the third dimension than those farther away, pushing the feature vectors near (0, 0) to a greater height. The maximal margin plane will cut the points in two in this third dimension, giving a perfect linear classifier.

Incidentally, SVMs can distinguish between only two classes. If there are more classes, an SVM classifier must approach them pair-wise. This is true of any classifier that uses linear discriminants.

This has been a fairly high-level description of support vector machines. It is a complex subject about which volumes have been written; see Burges,

Vapnick, and Witten for more details. Actual working software can be found in many places on the Internet, including the WEKA system (`www.cs.waikato.ac.nz/ml/weka/`), SVM^{light} (`http://svmlight.joachims.org/`), and LIBSVM (`www.csie.ntu.edu.tw/~cjlin/libsvm/`), for starters. There are many links at `www.support-vector.net/software.html`.

8.5 Multiple Classifiers—Ensembles

In complex situations, where there are many classes and many features, it is often true that some classifiers work better for some of the classes than others. One classifier may be able to identify cars in an image, for example, while another is better at trucks, or perhaps even hatchbacks. It may also be that some classifiers work better in some kinds of lighting, or in the presence of specific sorts of noise. In those situations it may be desirable to use more than one kind of classifier, and to merge the results after classification. These are referred to as *ensemble classifiers*.

The key with an ensemble is to find a way to merge the diverse results from the individual classifiers. They may be of quite different types and have very different methods, but all have the same basic goal, even if the problem has been distributed. In the following description, the hand-printed digital recognition problem of Chapter 9 will be developed. In this problem, an image is presented to the classifier that contains a single hand-printed digit, 0 through 9, which has been scanned or otherwise converted into image form. The question: what digit is this?

8.5.1 Merging Multiple Methods

A classifier can produce one of three kinds of output. The simplest and probably the most common is a basic, unqualified expression of the class determined for the data object. For a digit-classification scheme, this would mean that the classifier might simply state, ''This is a SIX,'' for example; this will be called a *type 1* response [Xu, 1992]. A classifier may also produce a ranking of the possible classes for a data object. In this case, the classifier may say, ''This is most likely a FIVE, but could be a THREE, and is even less likely to be a TWO.'' Probabilities are not associated with the ranking. This will be called a *type 2* response. Finally, a classifier may give a probability or other such confidence rating to each of the possible classes. This is the most specific case of all, since either a ranking or a classification can be produced from it. In this case, each possible digit would be given a confidence number that can be normalized to any specific range. This will be called a *type 3* response.

Any reasonable scheme for merging the results from multiple classifiers must deal with three important issues:

1. The response of the multiple classifier must be the best one given the results of the individual classifiers. It should in some logical way represent the most likely true classification, even when presented with contradictory individual classifications.
2. The classifiers in the system may produce different types of response. These must be merged into a coherent single response.
3. The multiple classifier must yield the correct result more often than any of the individual classifiers, or there is no point.

The first problem has various potential solutions for each possible type of response, and these will be dealt with first.

8.5.2 Merging Type 1 Responses

Given that the output of each classifier is a single, simple classification value, the obvious way to combine them is by using a voting strategy. A majority voting scheme can be expressed as follows: let $C_i(x)$ be the result produced by classifier i for the digit image x, where there are k different classifiers in the system; then let *H(d)* be the number of classifiers giving a classification of d for the digit image x, where d is one of {*0,1,2,3,4,5,6,7,8,9*}. H can be thought of as a histogram, and could be calculated in the following manner:

```
for (i=0; i<k; i++)
       H[ Ci(x) ] += 1;
```

Then, the overall classification E, expressing the opinions of the k classifiers, could be:

$$E(x) = \begin{cases} j & \text{if } \max(H(i)) = H(j) \text{ and } H(j) > \dfrac{k}{2} \\ 10 & \text{otherwise} \end{cases} \qquad \text{(EQ 8.8)}$$

This is called a *simple majority vote* (SMV). For comparison, a *parliamentary majority vote* would simply select j so that *H(j)* was a maximum. An easy generalization of this scheme replaces the constant $k/2$ in the above expression with $k^*\alpha$ for $0 <= \alpha <= 1$ [Xu, 1992]. This permits a degree of flexibility in deciding what degree of majority will be sufficient, and will be called a *weighted majority vote* (WMV). This scheme can be expressed as:

$$E(x) = \begin{cases} j & \text{if } \max(H(i)) = H(j) \text{ and } H(j) > \alpha k \\ 10 & \text{Otherwise} \end{cases} \qquad \text{(EQ 8.9)}$$

For example, many important votes in government and administrative committees require a 2/3 majority in order to pass. This would be equivalent to a value of $\alpha = 2/3$ in Equation 8.9.

Neither of the preceding two equations takes into account the possibility that all the dissenting classifiers agree with each other. Consider the following cases. In case A there are ten classifiers, with six of them supporting a classification of ''6,'' one supporting ''5,'' one supporting ''2,'' and two classifiers rejecting the input digit. In case B, using the same ten classifiers, six of them support the classification ''6,'' and the other four all agree that it is a ''5.'' Do cases A and B both support a classification of ''6,'' and do they do so equally strongly?

One way to incorporate dissent into the decision is to let max1 be the number of classifiers that support the majority classification j (max1 = $H(j)$), and to let max2 be the number supporting the second most popular classification h (max2 = $H(h)$). Then the classification becomes:

$$E(x) = \begin{cases} j & \text{if } \max(H(i)) = H(j) \text{ and } \max1 - \max2 >= (\alpha k) \\ 10 & \text{Otherwise} \end{cases} \qquad \text{(EQ 8.10)}$$

where α is between 0.0 and 1.0. This is called a *dissenting-weighted majority vote* (DWMV).

8.5.3 Evaluation

A multiple classifier system involves passing the input image to each classifier, gathering the results from each, and then using those as a vote on the final result. This must be done repeatedly for images having a known content, and the results of the vote compared against the known true value. Each of these is one trial, and the percentage of the trials that yield a correct answer gives a key metric of the value of the classifier combination. Some methods are more difficult to evaluate. For example, WMV requires an assessment of the effect of the value of α on the results. A way to deal with this is to write a small program to vary α from 0.05 to 0.95, classifying all sample digits on each iteration.

This evaluation process can be then repeated multiple times, omitting one of the classifiers each time to test the relative effect of each classifier on the overall success. If omitting a classifier actually improves the result, then that classifier should be removed from the collection for that kind of data. This much data requires a numerical value that can be used to assess the quality of the results. The recognition rate could be used alone, but this does not take into account that a rejection is much better than a misclassification; both would count against the recognition rate. A measure of *reliability* can be computed as:

$$\text{Reliability} = \frac{\text{Recognition}}{100\% - \text{Rejection}} \qquad \text{(EQ 8.11)}$$

The reliability value will be low when few misclassifications occur. Unfortunately, it will be high if recognition is only 50%, with the other 50% being rejections. This would not normally be thought of as acceptable performance. A good classifier will combine high reliability with a high recognition rate; in

that case, why not simply use the product *reliability*recognition* as a measure of performance? In the 50/50 example above, this measure would have the value 0.5: reliability is 100% (1.0) and recognition is 50% (0.5). In a case where the recognition rate was 50%, with 25% rejections and 25% misclassifications, this measure will have the value 0.333, indicating that the performance is not as good. The value *reliability*recognition* will be called *acceptability*. The first thing that should be done is to determine which value of α gives the best results, which is more accurately done when the data is presented in tabular form or as a graph of alpha versus acceptability. For example, consider the data in Table 8.3.

Table 8.3: Acceptability of the Multiple Classifier Using a Weighted Majority Vote

ALPHA	ACCEPTABILITY
0.05	0.992
0.25	0.993
0.50	0.978
0.75	0.823

Given that this is a table of results from the multiple classifier using WMV, it can be concluded that α should be between 0.25 and 0.5, for in this range the acceptability peaks without causing a drop in recognition rate.

8.5.4 Converting Between Response Types

Before proceeding to analyze methods for merging type 2 responses (ranks), it would be appropriate to discuss means of converting one response type to another. In particular, not all the classifiers yield a rank ordering, and this will be needed before merging the type 2 responses with those of types 1 and 3.

- **Type 3 to Type 1** — Select the class having the maximum confidence rating as the response.
- **Type 3 to Type 2** — Sort the confidence ratings in descending order. The corresponding classes are in rank order.
- **Type 2 to Type 1** — Select the class having the highest rank as the type 1 response.

Converting a type 1 response to a type 3 cannot be done in a completely general and reliable fashion. However, an approximation can be based on the measured past performance of the particular algorithm. Each row in the confusion matrix represents the classifications actually encountered for a

particular digit with that classifier expressed as a probability, and the columns represent the other classifications possible for a specified classification; this latter could be used as the confidence rating. The conversions from type 1 can be expressed as:

- **Type 1 to Type 3** — Compute the confusion matrix K for the classifier. If the classification in this case is j, then first compute:

$$S = \sum_{i=0}^{9} K(i,j) \qquad \text{(EQ 8.12)}$$

  Now compute the type 3 response as a vector V, where

$$V(i) = \frac{K(i,j)}{S} \qquad \text{(EQ 8.13)}$$

- **Type 1 to Type 2** — Convert from type 1 to type 3 as above, and then convert to type 2 from type 3.

8.5.5 Merging Type 2 Responses

The problem encountered when attempting to merge type 2 responses is as follows: given M rankings, each having N choices, which choice has the largest degree of support? For example, consider the following 3-voter/4-choice problem [Straffin, 1980]:

Voter 1: a b c d **Voter 2:** c a b d **Voter 3:** b d c a

This case has no majority winner; a, b and c each get one first place vote. Intuitively, it seems reasonable to use the second place votes in this case to see if the situation resolves itself. In this case, b receives two second place votes to a's one, which would tend to support b as the overall choice. In the general case, there are a number of techniques for merging rank-ordered votes, four of which will be discussed here.

The *Borda count* [Borda, 1781; Black, 1958] is a well-known scheme for resolving this kind of situation. Each alternative is given a number of points, depending on where in the ranking it has been placed. A selection is given no points for placing last, one point for placing next to last, and so on, up to $N-1$ points for placing first. In other words, the number of points given to a selection is the number of classes below it in the ranking. For the 3-voter/4-choice problem, the situation is:

Voter 1: a (3) b (2) c (1) d (0)

Voter 2: c (3) a (2) b (1) d (0)

Voter 3: b (3) d (2) c (1) a (0)

where the points received by each selection appears in parentheses behind the choice. The overall winner is the choice receiving the largest total number of points:

$$a = 3 + 2 + 0 = 5$$
$$b = 2 + 1 + 3 = 6$$
$$c = 1 + 3 + 1 = 5$$
$$d = 0 + 0 + 2 = 2$$

This gives choice b as the ''Borda winner.'' However, the Borda count does have a problem that might be considered serious. Consider the following 5-voter/3-choice problem:

Voter 1: a b c **Voter 2:** a b c **Voter 3:** a b c

Voter 4: b c a **Voter 5:** b c a

The Borda counts are $a = 6$, $b = 7$, $c = 2$, which selects b as the winner. However, a simple majority of the first place votes would have selected a! This violates the so-called *majority criterion* [Straffin, 1980]:

If a majority of voters have an alternative X as their first choice, a voting rule should choose X.

This is a weaker version of the *Condorcet winner criterion* [Condorcet, 1785]:

If there is an alternative X which could obtain a majority of votes in pair-wise contests against every other alternative, a voting rule should choose X as the winner.

This problem may have to be taken into account when assessing performance of the methods.

A procedure suggested by Thomas Hare [Straffin, 1980] falls into the category of an *elimination* process. The idea is to repeatedly eliminate undesirable choices until a clear majority supports one of the remaining choices. Hare's method is as follows: If a majority of the voters rank choice *X* in first place, then *X* is the winner; otherwise, the choice with the *smallest number of first place votes* is removed from consideration, and the first place votes are re-counted. This elimination process continues until a clear majority supports one of the choices.

The Hare procedure satisfies the majority criterion but fails the Condorcet winner criterion, as well as the *monotonicity criterion*:

If X is a winner under a voting rule, and one or more voters change their preferences in a way favorable to X without changing the order in which they prefer any other alternative, then X should still be the winner.

No rule that violates the monotonicity criterion will be considered as an option for the multiple classifier. This decision will eliminate the Hare procedure, but not the Borda count. With the monotonicity criterion in mind, two relatively simple rank merging strategies become interesting. The first is by Black [Black, 1958], and chooses the winner by the Condorcet criterion if

such a winner exists; if not, the Borda winner is chosen. This is appealing in its simplicity, and can be shown to be monotonic. Another strategy is the so-called *Copeland rule* [Straffin, 1980]: for each option compute the number of pair-wise wins of that option with all other options, and subtract from that the number of pair-wise losses. The overall winner is the class for which this difference is the greatest. In theory this rule is superior to the others discussed so far, but it has a drawback in that it tends to produce a relatively large number of tie votes in general.

8.5.6 Merging Type 3 Responses

The classifier systems discussed so far have no single classifier that gives a proper type 3 response. Because of this, the problem of merging type 3 responses was not pursued with as much vigor as were the type 1 and 2 problems. Indeed, the solution may be quite simple. Suen [Xu, 1992] decides that any set of type 3 classifiers can be combined using an averaging technique. That is,

$$P_E(x \in C_i|x) = \frac{1}{k}\sum_{j=1}^{k} P_j(x \in C_i|x), i = 1, \ldots, M \qquad \text{(EQ 8.14)}$$

where P_E is the probability associated with a given classification for the multiple classifier, and P_k is the probability associated with a given classification for each individual classifier k. The overall classification is the value j, for which

$$P_E(x \in C_j|x) \qquad \text{(EQ 8.15)}$$

is a maximum.

8.6 Bagging and Boosting

The methods referred to in the literature as bagging and boosting are re-sampling and weighting schemes designed to improve the overall success rate of a classifier.

8.6.1 Bagging

Bagging (or *bootstrap aggregation*) involves creating multiple training sets from the overall set of training data. Each set is drawn at random from the base set, with replacement. This means that the same training item could appear more than once in a particular training set. Each set has the same size, N, and there are T sets. A classifier is trained using each of the T data sets (sometimes called *bootstrap samples*), meaning that the classifier is trained using bootstrap sample

1 and the result is called *classifier 1*; then the classifier is trained again using bootstrap sample 2, and this is called *classifier 2*; and so on for all T samples, yielding T classifiers. These T classifiers are then combined using a majority vote to give a single classification that is based on many similar but differently trained classifiers.

A popular claim, due to Breiman [1996], is that bagging works best on ''unstable'' learning algorithms. In these cases, a small change in the training set can create a large change in classifications. Such unstable algorithms include neural networks and decision trees.

8.6.2 Boosting

The idea behind bagging is fundamentally the technique called *boosting*. In this technique, classifiers are trained on a sequence of training data sets. Unlike in bagging, however, the training sets are selected with a purpose. Samples that have failed to be classified by previous iterations of the process become increasingly likely to be used in training sets. Thus, the current iteration of boosting is an attempt to explicitly create a classifier, for example, where the previous failed to succeed. Another, early, name for boosting was *arcing*, from the descriptive phrase **a**daptively **r**esample and **c**ombine.

The method begins at iteration 1. The training set will contain N items from a set of M in total, and these are selected at random. That means that the probability of any item being used in the training set is $1/N$. The classifier is then trained on these data and tested. At this point, a set of probabilities is calculated for each data item based on whether it was classified successfully. Training set items remain at $1/N$, whereas items that failed to be classified have their probability increase. Then a second set of training items is chosen, one in which the failed items from the previous trial are more likely to be included. Selection is done with replacement, so the same hard to classify items could be used in the same set many times. The process creates a set of classifiers, each one more likely to succeed on the difficult items. All classifiers are used in an ensemble system, and the overall success rate should be higher than any one classifier. A boosting scheme usually has a weighted voting scheme, where each classifier's vote has a different value. Bagging considers the votes of each classifier as being worth the same amount.

After classifying all items in iteration t, an error is computed as:

$$e_t = \sum_{\text{All misclassified items } i} \frac{w^t i}{\sum_i w^t i} \qquad \text{(EQ 8.16)}$$

These error values will be used to update the weights for the next iteration. First, a scale factor is determined. One example is:

$$a_t = \frac{1}{2}\log\left(\frac{(1-e_t)}{e_t}\right) \quad \text{(EQ 8.17)}$$

Then the weights are updated according to the following scheme:

$$\begin{aligned} w_i^{t+1} &= w_i^{t-a_t} e \quad \text{if } i \text{ represented a correct classification} \\ w_i^{t+1} &= w_i^{t a_t} e \quad \text{if } i \text{ represented a incorrect classification} \end{aligned} \quad \text{(EQ 8.18)}$$

The weights are then normalized so that they sum to 1.0 (divide each one by the sum of them all).

There are many kinds of boosting algorithms listed in the literature. Most differ in the way that the probabilities are computed for selecting data for the next iteration, and in the way that the classifier votes are weighted in the ensemble. The result is a linear combination of the classifier sequence:

$$f(x) = \sum_{t=1}^{T} w_t C_t(x) \quad \text{(EQ 8.19)}$$

where the w_t are weights and the C_t are the classifiers.

Adaboost (Adaptive boosting) [Freund, 1995] was an early development and remains a popular choice today. (It uses the scale factor of Equation 8.17, by the way.) C++ code can be found on the Web (e.g., `www.di.unipi.it/%7Egulli/coding/adaboost.tgz`). *LPBoost* is a scheme that uses linear programming to maximize the margin between training sets [Demiriz, 2002].

8.7 Website Files

`nn.c`	C program to compute the nearest neighbor classification of the Iris data
`nkn.c`	k-nearest neighbor classifier for Iris data
`nc.c`	Nearest centroid classifier
`reg1.c`	Region marking program for tomato/carrot image
`cross5.c`	5-way cross validation for Iris data
`loo.c`	Leave-one-out cross validation
`iris-train1.txt`	Training data, Iris *setosa*
`iris-train2.txt`	Training data, Iris *versicolor*

`iris-train3.txt`	Training data, Iris *virginica*
`iris-test1.txt`	Test data, Iris *setosa*
`iris-test2.txt`	Test data, Iris *versicolor*
`iris-test3.txt`	Test data, Iris *setosa*
`iris-data.txt`	Complete Iris data set

8.8 References

Anderson, E. ''The Irises of the Gaspé Peninsula.'' Bulletin of the American Iris Society 59 (1935): 2–5.

Black, D. *The Theory of Committees and Elections*, Cambridge: Cambridge University Press, 1958.

Borda, Jean-Charles de. ''Mémoire sur les Elections au Scrutin,'' *Histoire de l'Académie Royale des Sciences*. Paris, 1781.

Breiman, L., ''Bagging Predictors.'' *Machine Learning* 24, no. 2 (1996): 123–140.

Brams, S. J., and P. C. Fishburn. *Approval Voting*, Boston: Birkhauser, 1983.

Bunch, J. R., and L. Kaufman. ''A Computational Method for the Indefinite Quadratic Programming Problem.'' *Linear Algebra and its Applications* 34 (1980): 341–370.

Burges, C. J. C. ''A Tutorial on Support Vector Machines for Pattern Recognition,'' *Data Mining and Knowledge Discovery* 2 (1998): 121–167.

Condorcet, Marquis de. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. Paris, 1785.

Demiriz, A., K. P. Bennett, and J. Shawe-Taylor. ''Linear Programming Boosting via Column Generation.'' *Kluwer Machine Learning* 46 (2002): 225–254.

Devijver, P. A., and J. Kittler, *Pattern Recognition: A Statistical Approach*. London: Prentice-Hall, 1982.

Duda, R. O., P. E. Hart, and D. H. Stork. *Pattern Classification* (2nd ed.). Wiley Interscience, 2000.

Enelow, J. M., and M. J. Hinich. *The Spatial Theory of Voting: An Introduction*. Cambridge: Cambridge University Press, 1984.

Farquharson, R. *Theory of Voting*, New Haven: Yale University Press, 1969.

Fisher, R. A. ''The Use of Multiple Measurements in Taxonomic Problems.'' *Annals of Eugenics* 7 (1936): 179– 188. `http://digital.library.adelaide.edu.au/coll/special//fisher/138.pdf`.

Fletcher, R. *Practical Methods of Optimization*. 2nd. ed. John Wiley and Sons, Inc., 1987.

Freund, Y., and R. E. Schapire. ''A Short Introduction to Boosting.'' *Journal of Japanese Society for Artificial Intelligence* 14, no. 5 (September, 1999):771–780.

Freund, Y. ''Boosting a weak learning Algorithm by Majority.'' *Information and Computation* 121, no. 2 (1995): 256–285.

Freund, Y. ''An Adaptive Version of the Boost by Majority Algorithm.'' *Proceedings of the Twelfth Annual Conference on Computational Learning Theory*, 1999.

Ho, T. K., J. J. Hull, and S. N. Srihari. ''Decision Combination in Multiple Classifier Systems.'' *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16, no. 1 (January 1994).

Kaufman, L. ''Solving the Quadratic Programming Problem Arising in Support Vector Classification.'' In *Advances in Kernel Methods: Support Vector Learning*, edited by Bernhard Schölkopf, Chrisopher J. C. Burges, and Alexander J. Smola. Cambridge, MA: MIT Press, 1998.

McLachlan, G. J. *Discriminant Analysis and Statistical Pattern Recognition*. Wiley Interscience, 2004.

Parker, J. R. *Practical Computer Vision Using C*. New York: John Wiley & Sons, Inc., 1994.

Picard, Richard, Dennis Cook. ''Cross-Validation of Regression Models.'' *Journal of the American Statistical Association* 79, no. 387 (1984): 575–583.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vettering. *Numerical Recipes in C: The Art of Scientific Computing*. 2nd ed. Cambridge: Cambridge University Press, 1992.

Shawe-Taylor, J., and N. Cristianini. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge: Cambridge University Press, 2000.

Straffin, P. D., Jr. *Topics in the Theory of Voting*. Boston: Birkhauser, 1980.

Vapnik, V. *The Nature of Statistical Learning Theory*. Springer, 1995.

Witten, I. H., and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. San Francisco: Morgan Kaufmann, 2000.

Xu, L., A. Krzyzak, and C. Y. Suen. ''Methods of Combining Multiple Classifiers and Their Application to Handwriting Recognition.'' *IEEE Transactions on Systems, Man, and Cybernetics* 22, no. 3.