22 A QUICK LOOK AT MACHINE LEARNING

The amount of digital data in the world has been growing at a rate that defies human comprehension. The world's data storage capacity has doubled about every three years since the 1980s. During the time it will take you to read this chapter, approximately 10^{18} bits of data will be added to the world's store. It's not easy to relate to a number that large. One way to think about it is that 10^{18} Canadian pennies would have a surface area roughly twice that of the earth.

Of course, more data does not always lead to more useful information. Evolution is a slow process, and the ability of the human mind to assimilate data does not, alas, double every three years. One approach that the world is using to attempt to wring more useful information from "big data" is **statistical machine learning**.

Machine learning is hard to define. In some sense, every useful program learns something. For example, an implementation of Newton's method learns the roots of a polynomial. One of the earliest definitions was proposed by the American electrical engineer and computer scientist Arthur Samuel,[155] who defined it as a "field of study that gives computers the ability to learn without being explicitly programmed."

Humans learn things in two ways—memorization and generalization. We use memorization to accumulate individual facts. In England, for example, primary school students might learn a list of English monarchs. Humans use **generalization** to deduce new facts from old facts. A student of political science, for example, might observe the behavior of a large number of politicians, and generalize from those observations to conclude that all politicians lie on the campaign trail.

When computer scientists speak about machine learning, they most often mean the discipline of writing programs that automatically learn to make useful inferences from implicit patterns in data. For example, linear regression (see Chapter 18) learns a curve that is a model of a collection of examples. That model can then be used to make predictions about previously unseen examples. The basic paradigm is

1. Observe a set of examples, frequently called the **training data**, that represent incomplete information about some statistical phenomenon
2. Use inference techniques to create a model of a process that could have generated the observed examples, and
3. Use that model to make predictions about previously unseen examples.

Suppose, for example, you were given the two sets of names in Figure 22.1 and the **feature vectors** in Figure 22.2.

```
A: {Abraham Lincoln, George Washington, Charles de Gaulle}
B: {Benjamin Harrison, James Madison, Louis Napoleon}
```

Figure 22.1: Two sets of names

```
Abraham Lincoln: [American, President, 193 cm tall]
George Washington: [American, President, 189 cm tall]
Charles de Gaulle: [French, President, 196 cm tall]
Benjamin Harrison: [American, President, 168 cm tall]
James Madison: [American, President, 163 cm tall]
Louis Napoleon: [French, President, 169 cm tall]
```

Figure 22.2: Associating a feature vector with each name

Each element of a vector corresponds to some aspect (i.e., feature) of the person. Based on this limited information about these historical figures, you might infer that the process that assigned either the label `A` or the label `B` to each of these examples was intended to separate tall presidents from shorter ones.

There are a large number of different approaches to machine learning, but all try to learn a model that is a generalization of the provided examples. All have three components:

- A representation of the model,
- An objective function for assessing the goodness of the model, and
- An optimization method for learning a model that minimizes or maximizes the value of the objective function.

Broadly speaking, machine learning algorithms can be thought of as either supervised or unsupervised.

In **supervised learning**, we start with a set of feature vector/value pairs. The goal is to derive from these pairs a rule that predicts the value associated with a previously unseen feature vector. **Regression models** associate a real number with each feature vector. **Classification models** associate one of a finite number of labels with each feature vector.[156]

In Chapter 18, we looked at one kind of regression model, linear regression. Each feature vector was an x-coordinate, and the value associate with it was the corresponding y-coordinate. From the set of feature vector/value pairs we learned a model that could be used to predict the y-coordinate associated with any x-coordinate.

Now, let's look at a simple classification model. Given the sets of presidents we labeled `A` and `B` in Figure 22.1 and the feature vectors in Figure 22.2, we can generate the feature vector/label pairs in Figure 22.3.

```
[American, President, 193 cm tall], A
[American, President, 189 cm tall], A
[French, President, 196 cm tall], A
[American, President, 168 cm tall], B
[American, President, 163 cm tall], B
[French, President, 169 cm tall], B
```

Figure 22.3 Feature vector/label pairs for presidents

From these labeled examples, a learning algorithm might infer that all tall presidents should be labeled `A` and all short presidents labeled `B`. When asked to assign a label to

`[American, President, 189 cm.]`[157]

it would use the rule it had learned to choose label `A`.

Supervised machine learning is broadly used in practice for such tasks as detecting fraudulent use of credit cards and recommending movies to people.

In **unsupervised learning**, we are given a set of feature vectors but no labels. The goal of unsupervised learning is to uncover latent structure in the set of feature vectors. For example, given the set of presidential feature vectors, an unsupervised learning algorithm might separate the presidents into tall and short, or perhaps into American and French. Broadly speaking, approaches to unsupervised machine learning can be categorized as either methods for clustering or methods for learning latent variable models.

A **latent variable** is a variable whose value is not directly observed, but can be inferred from the values of variables that are observed. Admissions officers at universities, for example, try to infer the probability of an applicant being a successful student (the latent variable), based on a set of observable values such as secondary school grades and performance on standardized tests. There is a rich set of methods for learning latent variable models, but we do not cover them in this book.

Clustering partitions a set of examples into groups (called clusters) such that examples in the same group are more similar to each other than they are to examples in other groups. Geneticists, for example, use clustering to find groups of related genes. Many popular clustering methods are surprisingly simple.

We present a widely used clustering algorithm in Chapter 23, and several approaches to supervised learning in Chapter 24. In the remainder of this chapter, we discuss the process of building feature vectors and different ways of calculating the similarity between two feature vectors.

22.1 Feature Vectors

The concept of **signal-to-noise ratio** (**SNR**) is used in many branches of engineering and science. The precise definition varies across applications, but the basic idea is simple. Think of it as the ratio of useful input to irrelevant input. In a restaurant, the signal might be the voice of your dinner date, and the noise the voices of the other diners.[158] If we were trying to predict which students would do well

in a programming course, previous programming experience and mathematical aptitude would be part of the signal, but hair color merely noise. Separating the signal from the noise is not always easy. And when it is done poorly, the noise can be a distraction that obscures the truth in the signal.

The purpose of **feature engineering** is to separate those features in the available data that contribute to the signal from those that are merely noise. Failure to do an adequate job of this can lead to a bad model. The danger is particularly high when the **dimensionality** of the data (i.e., the number of different features) is large relative to the number of samples.

Successful feature engineering is an abstraction process that reduces the vast amount of information that might be available to information from which it will be productive to generalize. Imagine, for example, that your goal is to learn a model that will predict whether a person is likely to suffer a heart attack. Some features, such as their age, are likely to highly relevant. Other features, such as whether they are left-handed, are less likely to be relevant.

There are **feature elimination** techniques that can be used to automatically identify which features in a given set of features are most likely to helpful. For example, in the context of supervised learning, one can select those features that are most strongly correlated with the labels of the examples.[159] However, these feature elimination techniques are of little help if relevant features are not there to start with. Suppose that our original feature set for the heart attack example includes height and weight. It might be the case that while neither height or weight is highly predictive of a heart attack, body mass index (BMI) is. While BMI can be computed from height and weight, the relationship (weight in kilograms divided by the square of height in meters) is too complicated to be automatically found by current machine learning techniques. Successful machine learning often involves the design of features by those with domain expertise.

In unsupervised learning, the problem is even harder. Typically, we choose features based upon our intuition about which features might be relevant to the kinds of structure we would like to find. However, relying on intuition about the potential relevance of features is problematical. How good is your intuition about whether one's dental history is a good predictor of a future heart attack?

Consider Figure 22.4, which contains a table of feature vectors and the label (reptile or not) with which each vector is associated.

Name	Egg-laying	Scales	Poisonous	Cold-blooded	# Legs	Reptile
Cobra	True	True	True	True	0	Yes
Rattlesnake	True	True	True	True	0	Yes
Boa constrictor	False	True	False	True	0	Yes
Alligator	True	True	False	True	4	Yes
Dart frog	True	False	True	False	4	No
Salmon	True	True	False	True	0	No
Python	True	True	False	True	0	Yes

Figure 22.4 Name, features and labels for assorted animals

A supervised machine learning algorithm (or a human) given only the information about cobras—

i.e., only the first row of the table—cannot do much more than to remember the fact that a cobra is a reptile. Now, let's add the information about rattlesnakes. We can begin to generalize, and might infer the rule that an animal is a reptile if it lays eggs, has scales, is poisonous, is cold-blooded, and has no legs.

Now, suppose we are asked to decide if a boa constrictor is a reptile. We might answer "no," because a boa constrictor is neither poisonous nor egg-laying. But this would be the wrong answer. Of course, it is hardly surprising that attempting to generalize from two examples might lead us astray. Once we include the boa constrictor in our training data, we might formulate the new rule that an animal is a reptile if it has scales, is cold-blooded, and is legless. In doing so, we are discarding the features `egg-laying` and `poisonous` as irrelevant to the classification problem.

If we use the new rule to classify the alligator, we conclude incorrectly that since it has legs it is not a reptile. Once we include the alligator in the training data we reformulate the rule to allow reptiles to have either none or four legs. When we look at the dart frog, we correctly conclude that it is not a reptile, since it is not cold-blooded. However, when we use our current rule to classify the salmon, we incorrectly conclude that a salmon is a reptile. We can add yet more complexity to our rule, to separate salmon from alligators, but it's a losing battle. There is no way to modify our rule so that it will correctly classify both salmon and pythons, since the feature vectors of these two species are identical.

This kind of problem is more common than not in machine learning. It is quite rare to have feature vectors that contain enough information to classify things perfectly. In this case, the problem is that we don't have enough features.

If we had included the fact that reptile eggs have amnios,[160] we could devise a rule that separates reptiles from fish. Unfortunately, in most practical applications of machine learning it is not possible to construct feature vectors that allow for perfect discrimination.

Does this mean that we should give up because all of the available features are mere noise? No. In this case, the features `scales` and `cold-blooded` are necessary conditions for being a reptile, but not sufficient conditions. The rule that an animal is a reptile if it has scales and is cold-blooded will not yield any false negatives, i.e., any animal classified as a non-reptile will indeed not be a reptile. However, it will yield some false positives, i.e., some of the animals classified as reptiles will not be reptiles.

22.2 Distance Metrics

In Figure 22.4 we described animals using four binary features and one integer feature. Suppose we want to use these features to evaluate the similarity of two animals, for example, to ask whether a rattlesnake is more similar to a boa constrictor or to a dart frog.[161]

The first step in doing this kind of comparison is converting the features for each animal into a sequence of numbers. If we say `True = 1` and `False = 0`, we get the following feature vectors:

```
Rattlesnake: [1,1,1,1,0]
Boa constrictor: [0,1,0,1,0]
Dart frog: [1,0,1,0,4]
```

There are many different ways to compare the similarity of vectors of numbers. The most commonly used metrics for comparing equal-length vectors are based on the **Minkowski distance**:[162]

$$distance(V, W, p) = (\sum_{i=1}^{len} abs(V_i - W_i)^p)^{1/p}$$

where *len* is the length of the vectors.

The parameter p, which must be at least 1, defines the kinds of paths that can be followed in traversing the distance between the vectors *V* and *W*.[163] This can be mostly easily visualized if the vectors are of length two, and can therefore be represented using Cartesian coordinates. Consider the picture in Figure 22.5.

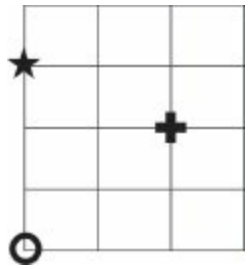

Figure 22.5 Visualizing distance metrics

Is the circle in the bottom left corner closer to the cross or closer to the star? It depends. If we can travel in a straight line, the cross is closer. The Pythagorean Theorem tells us that the cross is the square root of 8 units from the circle, about 2.8 units, whereas we can easily see that the star is 3 units from the circle. These distances are called Euclidean distances, and correspond to using the Minkowski distance with p = 2. But imagine that the lines in the picture correspond to streets, and that we have to stay on the streets to get from one place to another. The star remains 3 units from the circle, but the cross is now 4 units away. These distances are called **Manhattan distances**,[164] and they correspond to using the Minkowski distance with p = 1. Figure 22.6 contains a function implementing the Minkowski distance.

Figure 22.7 contains class `Animal`. It defines the distance between two animals as the Euclidean distance between the feature vectors associated with the animals.

```
def minkowskiDist(v1, v2, p):
    """Assumes v1 and v2 are equal-length arrays of numbers
       Returns Minkowski distance of order p between v1 and v2"""
    dist = 0.0
    for i in range(len(v1)):
        dist += abs(v1[i] - v2[i])**p
    return dist**(1/p)
```

Figure 22.6 Minkowski distance

```
class Animal(object):
    def __init__(self, name, features):
        """Assumes name a string; features a list of numbers"""
        self.name = name
        self.features = pylab.array(features)

    def getName(self):
        return self.name

    def getFeatures(self):
        return self.features

    def distance(self, other):
        """Assumes other an Animal
           Returns the Euclidean distance between feature vectors
              of self and other"""
        return minkowskiDist(self.getFeatures(),
                             other.getFeatures(), 2)
```

Figure 22.7 Class `Animal`

Figure 22.8 contains a function that compares a list of animals to each other and produces a table showing the pairwise distances. The code uses a PyLab plotting facility that we have not previously used: `table`.

The `table` function produces a plot that (surprise!) looks like a table. The keyword arguments `rowLabels` and `colLabels` are used to supply the labels (in this example the names of the animals) for the rows and columns. The keyword argument `cellText` is used to supply the values appearing in the cells of the table. In the example, `cellText` is bound to `tableVals`, which is a list of lists of strings. Each element in `tableVals` is a list of the values for the cells in one row of the table. The keyword argument `cellLoc` is used to specify where in each cell the text should appear, and the keyword argument `loc` is used to specify where in the figure the table itself should appear. The last keyword parameter used in the example is `colWidths`. It is bound to a list of floats giving the width (in inches) of each column in the table. The code `table.scale(1, 2.5)` instructs PyLab to leave the horizontal width of the cells unchanged, but to increase the height of the cells by a factor of 2.5 (so the tables look prettier).

```
def compareAnimals(animals, precision):
    """Assumes animals is a list of animals, precision an int >= 0
       Builds a table of Euclidean distance between each animal"""
    #Get labels for columns and rows
    columnLabels = []
    for a in animals:
        columnLabels.append(a.getName())
    rowLabels = columnLabels[:]
    tableVals = []
    #Get distances between pairs of animals
    #For each row
    for a1 in animals:
        row = []
        #For each column
        for a2 in animals:
            if a1 == a2:
                row.append('--')
            else:
                distance = a1.distance(a2)
                row.append(str(round(distance, precision)))
        tableVals.append(row)
    #Produce table
    table = pylab.table(rowLabels = rowLabels,
                        colLabels = columnLabels,
                        cellText = tableVals,
                        cellLoc = 'center',
                        loc = 'center',
                        colWidths = [0.2]*len(animals))
    table.scale(1, 2.5)
    pylab.savefig('distances')
```

Figure 22.8 Build table of distances between pairs of animals

If we run the code

```
rattlesnake = Animal('rattlesnake', [1,1,1,1,0])
boa = Animal('boa\nconstrictor', [0,1,0,1,0])
dartFrog = Animal('dart frog', [1,0,1,0,4])
animals = [rattlesnake, boa, dartFrog]
compareAnimals(animals, 3)
```

it produces the table in Figure 22.9 and saves it in a file named `distances`.

As you probably expected, the distance between the rattlesnake and the boa constrictor is less than that between either of the snakes and the dart frog. Notice, by the way, that the dart frog is a bit closer to the rattlesnake than to the boa constrictor.

	rattlesnake	boa constrictor	dart frog
rattlesnake	--	1.414	4.243
boa constrictor	1.414	--	4.472
dart frog	4.243	4.472	--

Figure 22.9 Distances between three animals

Now, let's insert before the last line of the above code the lines

```
alligator = Animal('alligator', [1,1,0,1,4])
animals.append(alligator)
```

It produces the table in Figure 22.10.

	rattlesnake	boa constrictor	dart frog	alligator
rattlesnake	--	1.414	4.243	4.123
boa constrictor	1.414	--	4.472	4.123
dart frog	4.243	4.472	--	1.732
alligator	4.123	4.123	1.732	--

Figure 22.10 Distances between four animals

Perhaps you're surprised that the alligator is considerably closer to the dart frog than to either the rattlesnake or the boa constrictor. Take a minute to think about why.

The feature vector for the alligator differs from that of the rattlesnake in two places: whether it is poisonous and the number of legs. The feature vector for the alligator differs from that of the dart frog in three places: whether it is poisonous, whether it has scales, and whether it is cold-blooded. Yet according to our distance metric the alligator is more like the dart frog than like the rattlesnake. What's going on?

The root of the problem is that the different features have different ranges of values. All but one of the features range between 0 and 1, but the number of legs ranges from 0 to 4. This means that when we calculate the Euclidean distance the number of legs gets disproportionate weight. Let's see what happens if we turn the feature into a binary feature, with a value of 0 if the animal is legless and 1 otherwise.

	rattlesnake	boa constrictor	dart frog	alligator
rattlesnake	--	1.414	1.732	1.414
boa constrictor	1.414	--	2.236	1.414
dart frog	1.732	2.236	--	1.732
alligator	1.414	1.414	1.732	--

Figure 22.11 Distances using a different feature representation

This looks a lot more plausible.

Of course, it is not always convenient to use only binary features. In Section 23.4, we will present a more general approach to dealing with differences in scale among features.

---

[155] Samuel is probably best known as the author of a program that played checkers. The program, which he started working on in the 1950s and continued to work on into the 1970s, was impressive for its time, though not particularly good by modern standards. However, while working on it Samuel invented several techniques that are still used today. Among other things, Samuel's checker-playing program was quite possibly the first program ever written that improved based upon "experience."

[156] Much of the machine learning literature uses the word "class" rather than "label." Since we have used the word "class" for something else in this book, we will stick to using "label" for this concept.

[157] In case you are curious, Thomas Jefferson was 189 cm. tall.

[158] Unless your dinner date is exceedingly boring. In that case, your dinner date's conversation becomes the noise, and the conversation at the next table the signal.

[159] Since features are often strongly correlated with each other, this can lead to a large number of redundant features. There are more sophisticated feature elimination techniques, but we do not cover them in this book.

[160] Amnios are protective outer layers that allow eggs to be laid on land rather than in the water.

[161] This question is not quite as silly as it sounds. A naturalist and a toxicologist (or someone looking

to enhance the effectiveness of a blow dart) might give different answers to this question.

[162] Another popular distance metric is cosine similarity. This captures the difference in the angle of the two vectors rather than the difference in magnitude. It is is often useful for high-dimensional vectors.

[163] When $p < 1$, peculiar things happen. Consider, for example $p = 0.5$ and the points $A = (0,0)$, $B = (1,1)$, and $C = (0,1)$. If you compute the pairwise distances beween these points, you will discover that the distance from A to B is 4, the distance from A to C is 1, and the distance from C to B is 1. Common sense dictates that the distance from A to B via C cannot be less than the distance from A to B. (Mathematicians refer to this as the triangle inequality, which states that for any triangle the sum of the lengths of any two sides must not be less than the length of the third side.)

[164] Manhattan Island is the most densely populated borough of New York City. On most of the island, the streets are laid out in a rectangular grid, so using the Minkowski distance with $p = 1$ provides a good approximation of the distance one has to travel to walk from one place to another. Driving in Manhattan is a totally different story.

23 CLUSTERING

Unsupervised learning involves finding hidden structure in unlabeled data. The most commonly used unsupervised machine learning technique is clustering.

Clustering can be defined as the process of organizing objects into groups whose members are similar in some way. A key issue is defining the meaning of "similar." Consider the plot in Figure 23.1, which shows the height, weight, and shirt color for 13 people.

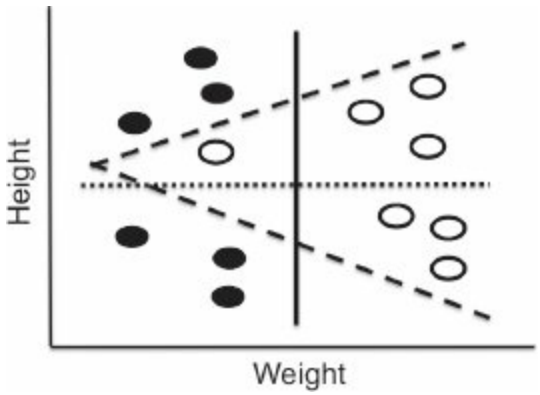

Figure 23.1 Height, weight, and kind of shirt

If we cluster people by height, there are two obvious clusters—delimited by the dotted horizontal line. If we cluster people by weight, there are two different obvious clusters—delimited by the solid vertical line. If we cluster people based on their shirts, there is yet a third clustering—delimited by the angled dashed lines. Notice, by the way, that this last division is not linear, i.e., we cannot separate the people by shirt color using a single straight line.

Clustering is an optimization problem. The goal is to find a set of clusters that optimizes an objective function, subject to some set of constraints. Given a distance metric that can be used to decide how close two examples are to each other, we need to define an objective function that minimizes the distance between examples in the same cluster, i.e., minimizes the dissimilarity of the examples within a cluster.

One measure, which we call variability, of how different the examples within a single cluster, c, are from each other is

$$variability(c) = \sum_{e \in c} distance(mean(c), e)^2$$

where $mean(c)$ is the mean of the feature vectors of all the examples in the cluster. The mean of a set

of vectors is computed component-wise. The corresponding elements are added, and the result divided by the number of vectors. If `v1` and `v2` are `arrays` of numbers, the value of the expression `(v1+v2)/2` is their **Euclidean mean**.

What we are calling variability is quite similar to the notion of variance presented in Chapter 15. The difference is that variability is not normalized by the size of the cluster, so clusters with more points are likely to look less cohesive according to this measure. If one wants to compare the coherence of two clusters of different sizes, one needs to divide the variability of each cluster by the size of the cluster.

The definition of variability within a single cluster, c, can be extended to define a dissimilarity metric for a set of clusters, C:

$$dissimilarity(C) = \sum_{c \in C} variability(c)$$

Notice that since we don't divide the variability by the size of the cluster, a large incoherent cluster increases the value of *dissimilarity(C)* more than a small incoherent cluster does. This is by design.

So, is the optimization problem to find a set of clusters, C, such that *dissimilarity(C)* is minimized? Not exactly. It can easily be minimized by putting each example in its own cluster. We need to add some constraint. For example, we could put a constraint on the minimum distance between clusters or require that the maximum number of clusters is some k.

In general, solving this optimization problem is computationally prohibitive for most interesting problems. Consequently, people rely on greedy algorithms that provide approximate solutions. In Section 23.2, we present one such algorithm, k-means clustering. But first we will introduce some abstractions that are useful for implementing that algorithm (and other clustering algorithms as well).

23.1 Class Cluster

Class `Example` will be used to build the samples to be clustered. Associated with each example is a name, a feature vector, and an optional label. The `distance` method returns the Euclidean distance between two examples.

```
class Example(object):

    def __init__(self, name, features, label = None):
        #Assumes features is an array of floats
        self.name = name
        self.features = features
        self.label = label

    def dimensionality(self):
        return len(self.features)

    def getFeatures(self):
        return self.features[:]

    def getLabel(self):
        return self.label

    def getName(self):
        return self.name

    def distance(self, other):
        return minkowskiDist(self.features, other.getFeatures(), 2)

    def __str__(self):
        return self.name +':'+ str(self.features) + ':'\
               + str(self.label)
```

Figure 23.2 Class `Example`

Class `Cluster`, Figure 23.3, is slightly more complex. A cluster is a set of examples. The two interesting methods in `Cluster` are `computeCentroid` and `variability`. Think of the **centroid** of a cluster as its center of mass. The method `computeCentroid` returns an example with a feature vector equal to the Euclidean mean of the feature vectors of the examples in the cluster. The method `variability` provides a measure of the coherence of the cluster.

```
class Cluster(object):

    def __init__(self, examples):
        """Assumes examples a non-empty list of Examples"""
        self.examples = examples
        self.centroid = self.computeCentroid()
```

```
    def update(self, examples):
        """Assume examples is a non-empty list of Examples
           Replace examples; return amount centroid has changed"""
        oldCentroid = self.centroid
        self.examples = examples
        self.centroid = self.computeCentroid()
        return oldCentroid.distance(self.centroid)

    def computeCentroid(self):
        vals = pylab.array([0.0]*self.examples[0].dimensionality())
        for e in self.examples: #compute mean
            vals += e.getFeatures()
        centroid = Example('centroid', vals/len(self.examples))
        return centroid

    def getCentroid(self):
        return self.centroid

    def variability(self):
        totDist = 0.0
        for e in self.examples:
            totDist += (e.distance(self.centroid))**2
        return totDist

    def members(self):
        for e in self.examples:
            yield e

    def __str__(self):
        names = []
        for e in self.examples:
            names.append(e.getName())
        names.sort()
        result = 'Cluster with centroid '\
               + str(self.centroid.getFeatures()) + ' contains:\n  '
        for e in names:
            result = result + e + ', '
        return result[:-2] #remove trailing comma and space
```

Figure 23.3 Class `Cluster`

23.2 K-means Clustering

K-means clustering is probably the most widely used clustering method.[165] Its goal is to partition a set of examples into k clusters such that

- Each example is in the cluster whose centroid is the closest centroid to that example, and
- The dissimilarity of the set of clusters is minimized.

Unfortunately, finding an optimal solution to this problem on a large data set is computationally intractable. Fortunately, there is an efficient greedy algorithm[166] that can be used to find a useful approximation. It is described by the pseudocode

```
randomly choose k examples as initial centroids of clusters while true:
```

1. `Create k clusters by assigning each example to closest centroid`
2. `Compute k new centroids by averaging the examples in each cluster`
3. `If none of the centroids differ from the previous iteration: return the current set of clusters`

The complexity of step 1 is O(k*n*d), where k is the number of clusters, n is the number of examples, and d the time required to compute the distance between a pair of examples. The complexity of step 2 is O(n), and the complexity of step 3 is O(k). Hence, the complexity of a single iteration is O(k*n*d). If the examples are compared using the Minkowski distance, d is linear in the length of the feature vector.[167] Of course, the complexity of the entire algorithm depends upon the number of iterations. That is not easy to characterize, but suffice it to say that it is usually small.

One problem with the k-means algorithm is that the value returned depends upon the initial set of randomly chosen centroids. If a particularly unfortunate set of initial centroids is chosen, the algorithm might settle into a local optimum that is far from the global optimum. In practice, this problem is typically addressed by running k-means multiple times with randomly chosen initial centroids. We then choose the solution with the minimum dissimilarity of clusters.

Figure 23.4 contains a function, `trykmeans`, that calls `kmeans` (see Figure 23.5) multiple times and selects the result with the lowest dissimilarity. If a trial fails because `kmeans` generated an empty cluster and therefore raised an exception, `trykmeans` merely tries again—assuming that eventually `kmeans` will choose an initial set of centroids that successfully converges.

```
def dissimilarity(clusters):
    totDist = 0.0
    for c in clusters:
        totDist += c.variability()
    return totDist

def trykmeans(examples, numClusters, numTrials, verbose = False):
    """Calls kmeans numTrials times and returns the result with the
          lowest dissimilarity"""
    best = kmeans(examples, numClusters, verbose)
    minDissimilarity = dissimilarity(best)
    trial = 1
    while trial < numTrials:
        try:
            clusters = kmeans(examples, numClusters, verbose)
        except ValueError:
            continue #If failed, try again
        currDissimilarity = dissimilarity(clusters)
        if currDissimilarity < minDissimilarity:
            best = clusters
            minDissimilarity = currDissimilarity
        trial += 1
    return best
```

Figure 23.4 Finding the best k-means clustering

Figure 23.5 contains a translation into Python of the pseudocode describing k-means. The only wrinkle is that it raises an exception if any iteration creates a cluster with no members. Generating an empty cluster is rare. It can't occur on the first iteration, but it can occur on subsequent iterations. It usually results from choosing too large a k or an unlucky choice of initial centroids. Treating an empty cluster as an error is one of the options used by Matlab. Another is creating a new cluster containing a single point—the point furthest from the centroid in the other clusters. We chose to treat it an error to keep the implementation relatively simple.

```
def kmeans(examples, k, verbose = False):
    #Get k randomly chosen initial centroids, create cluster for each
    initialCentroids = random.sample(examples, k)
    clusters = []
    for e in initialCentroids:
        clusters.append(Cluster([e]))
```

```
    #Iterate until centroids do not change
    converged = False
    numIterations = 0
    while not converged:
        numIterations += 1
        #Create a list containing k distinct empty lists
        newClusters = []
        for i in range(k):
            newClusters.append([])

        #Associate each example with closest centroid
        for e in examples:
            #Find the centroid closest to e
            smallestDistance = e.distance(clusters[0].getCentroid())
            index = 0
            for i in range(1, k):
                distance = e.distance(clusters[i].getCentroid())
                if distance < smallestDistance:
                    smallestDistance = distance
                    index = i
            #Add e to the list of examples for appropriate cluster
            newClusters[index].append(e)

        for c in newClusters: #Avoid having empty clusters
            if len(c) == 0:
                raise ValueError('Empty Cluster')

        #Update each cluster; check if a centroid has changed
        converged = True
        for i in range(k):
            if clusters[i].update(newClusters[i]) > 0.0:
                converged = False
        if verbose:
            print('Iteration #' + str(numIterations))
            for c in clusters:
                print(c)
            print('') #add blank line
    return clusters
```

Figure 23.5 K-means clustering

23.3 A Contrived Example

Figure 23.7 contains code that generates, plots, and clusters examples drawn from two distributions.

The function `genDistributions` generates a list of `n` examples with two-dimensional feature vectors. The values of the elements of these feature vectors are drawn from normal distributions.

The function `plotSamples` plots the feature vectors of a set of examples. It uses `pylab.annotate` to place text next to points on the plot. The first argument is the text, the second argument the point with which the text is associated, and the third argument the location of the text relative to the point with which it is associated.

The function `contrivedTest` uses `genDistributions` to create two distributions of ten examples (each with the same standard deviation but different means), plots the examples using `plotSamples`, and then clusters them using `trykmeans`.

The call `contrivedTest(1, 2, True)` produced the plot in Figure 23.6 and printed the lines in Figure 23.8.

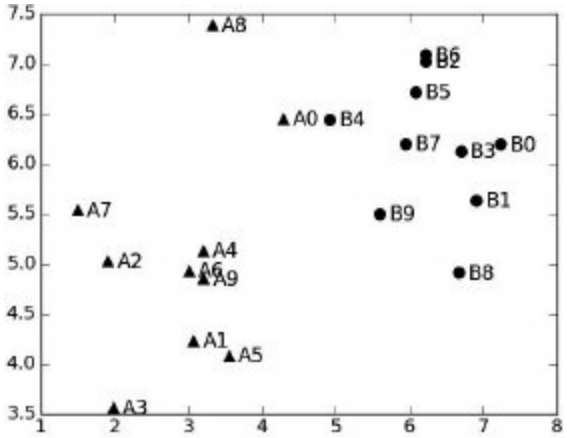

Figure 23.6 Examples from two distributions

```
def genDistribution(xMean, xSD, yMean, ySD, n, namePrefix):
    samples = []
    for s in range(n):
        x = random.gauss(xMean, xSD)
        y = random.gauss(yMean, ySD)
        samples.append(Example(namePrefix+str(s), [x, y]))
    return samples

def plotSamples(samples, marker):
    xVals, yVals = [], []
    for s in samples:
        x = s.getFeatures()[0]
        y = s.getFeatures()[1]
        pylab.annotate(s.getName(), xy = (x, y),
                       xytext = (x+0.13, y-0.07),
                       fontsize = 'x-large')
        xVals.append(x)
        yVals.append(y)
    pylab.plot(xVals, yVals, marker)

def contrivedTest(numTrials, k, verbose = False):
    xMean = 3
    xSD = 1
    yMean = 5
    ySD = 1
    n = 10
    d1Samples = genDistribution(xMean, xSD, yMean, ySD, n, 'A')
    plotSamples(d1Samples, 'k^')
    d2Samples = genDistribution(xMean+3, xSD, yMean+1, ySD, n, 'B')
    plotSamples(d2Samples, 'ko')
    clusters = trykmeans(d1Samples+d2Samples, k, numTrials, verbose)
    print('Final result')
    for c in clusters:
        print('', c)
```

Figure 23.7 A test of k-means

```
Iteration #1
Cluster with centroid [ 4.71113345  5.76359152] contains:
  A0, A1, A2, A4, A5, A6, A7, A8, A9, B0, B1, B2, B3, B4, B5, B6,
B7, B8, B9
Cluster with centroid [ 1.97789683  3.56317055] contains:
  A3

Iteration #2
Cluster with centroid [ 5.46369488  6.12015454] contains:
  A0, A4, A8, A9, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
Cluster with centroid [ 2.49961733  4.56487432] contains:
  A1, A2, A3, A5, A6, A7

Iteration #3
Cluster with centroid [ 5.84078727  6.30779094] contains:
  A0, A8, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
Cluster with centroid [ 2.67499815  4.67223977] contains:
  A1, A2, A3, A4, A5, A6, A7, A9

Iteration #4
Cluster with centroid [ 5.84078727  6.30779094] contains:
  A0, A8, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
Cluster with centroid [ 2.67499815  4.67223977] contains:
  A1, A2, A3, A4, A5, A6, A7, A9

Final result
 Cluster with centroid [ 5.84078727  6.30779094] contains:
  A0, A8, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
 Cluster with centroid [ 2.67499815  4.67223977] contains:
  A1, A2, A3, A4, A5, A6, A7, A9
```

Figure 23.8 Lines printed by call `contrivedTest(1, 2, True)`

Notice that the initial (randomly chosen) centroids led to a highly skewed clustering in which a single cluster contained all but one of the points. By the fourth iteration, however, the centroids had moved to places such that the points from the two distributions were reasonably well separated into two clusters. The only "mistakes" were made on `A0` and `A8`.

When we tried 50 trials rather than 1, by calling `contrivedTest(50, 2, False)`, it printed

```
Final result
 Cluster with centroid [ 2.74674403 4.97411447] contains:
  A1, A2, A3, A4, A5, A6, A7, A8, A9
```

```
Cluster with centroid [ 6.0698851 6.20948902] contains:
 A0, B0, B1, B2, B3, B4, B5, B6, B7, B8, B9
```

A0 is still mixed in with the B's, but A8 is not. If we try 1000 trials, we get the same result. That might surprise you, since a glance at Figure 23.6 reveals that if A0 and B0 are chosen as the initial centroids (which would probably happen with 1000 trials), the first iteration will yield clusters that perfectly separate the A's and B's. However, in the second iteration new centroids will be computed, and A0 will be assigned to a cluster with the B's. Is this bad? Recall that clustering is a form of unsupervised learning that looks for structure in unlabeled data. Grouping A0 with the B's is not unreasonable.

One of the key issues in using k-means clustering is choosing k. The function contrivedTest2 in Figure 23.9 generates, plots, and clusters points from three overlapping Gaussian distributions. We will use it to look at the results of clustering this data for various values of k. The data points are shows in Figure 23.10.

```
def contrivedTest2(numTrials, k, verbose = False):
    xMean = 3
    xSD = 1
    yMean = 5
    ySD = 1
    n = 8
    d1Samples = genDistribution(xMean,xSD, yMean, ySD, n, 'A')
    plotSamples(d1Samples, 'k^')
    d2Samples = genDistribution(xMean+3,xSD,yMean, ySD, n, 'B')
    plotSamples(d2Samples, 'ko')
    d3Samples = genDistribution(xMean, xSD, yMean+3, ySD, n, 'C')
    plotSamples(d3Samples, 'kx')
    clusters = trykmeans(d1Samples + d2Samples + d3Samples,
                         k, numTrials, verbose)
    pylab.ylim(0,11)
    print('Final result has dissimilarity',
          round(dissimilarity(clusters), 3))
    for c in clusters:
        print('', c)
```

Figure 23.9 Generating points from three distributions

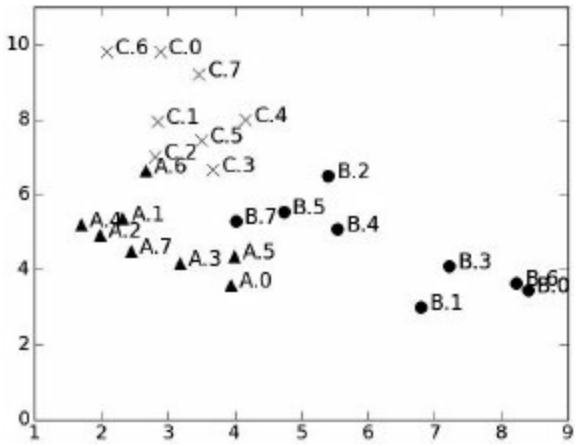

Figure 23.10 Points from three overlapping Gaussians

The invocation `contrivedTest2(40, 2)` prints

```
Final result has dissimilarity 90.128
 Cluster with centroid [ 5.5884966 4.43260236] contains:
  A.0, A.3, A.5, B.0, B.1, B.2, B.3, B.4, B.5, B.6, B.7
 Cluster with centroid [ 2.80949911 7.11735738] contains:
  A.1, A.2, A.4, A.6, A.7, C.0, C.1, C.2, C.3, C.4, C.5, C.6, C.7
```

The invocation `contrivedTest2(40, 3)` prints

```
Final result has dissimilarity 42.757
 Cluster with centroid [ 7.66239972 3.55222681] contains:
  B.0, B.1, B.3, B.6
 Cluster with centroid [ 3.56907939 4.95707576] contains:
  A.0, A.1, A.2, A.3, A.4, A.5, A.7, B.2, B.4, B.5, B.7
 Cluster with centroid [ 3.12083099 8.06083681] contains:
  A.6, C.0, C.1, C.2, C.3, C.4, C.5, C.6, C.7
```

And the invocation `contrivedTest2(40, 6)` prints

```
Final result has dissimilarity 11.441
 Cluster with centroid [ 2.10900238 4.99452866] contains:
  A.1, A.2, A.4, A.7
 Cluster with centroid [ 4.92742554 5.60609442] contains:
  B.2, B.4, B.5, B.7
 Cluster with centroid [ 2.80974427 9.60386549] contains:
  C.0, C.6, C.7
 Cluster with centroid [ 3.27637435 7.28932247] contains:
  A.6, C.1, C.2, C.3, C.4, C.5
 Cluster with centroid [ 3.70472053 4.04178035] contains:
  A.0, A.3, A.5
 Cluster with centroid [ 7.66239972 3.55222681] contains:
  B.0, B.1, B.3, B.6
```

The last clustering is the tightest fit, i.e., the clustering has the lowest dissimilarity (11.441). Does

this mean that it is the "best" clustering? Not necessarily. Recall that when we looked at linear regression in Section 18.1.1, we observed that by increasing the degree of the polynomial we got a more complex model that provided a tighter fit to the data. We also observed that when we increased the degree of the polynomial we ran the risk of finding a model with poor predictive value—because it overfit the data.

Choosing the right value for k is exactly analogous to choosing the right degree polynomial for a linear regression. By increasing k, we can decrease dissimilarity, at the risk of overfitting. (When k is equal to the number of examples to be clustered, the dissimilarity is 0!) If we have information about how the examples to be clustered were generated, e.g., chosen from m distributions, we can use that information to choose k. Absent such information, there are a variety of heuristic procedures for choosing k. Going into them is beyond the scope of this book.

23.4 A Less Contrived Example

Different species of mammals have different eating habits. Some species (e.g., elephants and beavers) eat only plants, others (e.g., lions and tigers) eat only meat, and some (e.g., pigs and humans) eat anything they can get into their mouths. The vegetarian species are called herbivores, the meat eaters are called carnivores, and those species that eat both plants and animals are called omnivores.

Over the millennia, evolution (or some other mysterious process) has equipped species with teeth suitable for consumption of their preferred foods.[168] That raises the question of whether clustering mammals based on their dentition produces clusters that have some relation to their diets.

Figure 23.11 shows the contents of a file listing some species of mammals, their dental formulas (the first 8 numbers), their average adult weight in pounds,[169] and a code indicating their preferred diet. The comments at the top describe the items associated with each mammal, e.g., the first item following the name is the number of top incisors.

```
#Name
#top incisors
#top canines
#top premolars
#top molars
#bottom incisors
#bottom canines
#bottom premolars
#bottom molars
#weight
#Label: 0=herbivore, 1=carnivore, 2=omnivore
Badger,3,1,3,1,3,1,3,2,10,1
Bear,3,1,4,2,3,1,4,3,278,2
Beaver,1,0,2,3,1,0,1,3,20,0
Brown bat,2,1,1,3,3,1,2,3,0.5,1
Cat,3,1,3,1,3,1,2,1,4,1
Cougar,3,1,3,1,3,1,2,1,63,1
```

```
Cow,0,0,3,3,3,1,2,1,400,0
Deer,0,0,3,3,4,0,3,3,200,0
Dog,3,1,4,2,3,1,4,3,20,1
Elk,0,1,3,3,3,1,3,3,500,0
Fox,3,1,4,2,3,1,4,3,5,1
Fur seal,3,1,4,1,2,1,4,1,200,1
Grey seal,3,1,3,2,2,1,3,2,268,1
Guinea pig,1,0,1,3,1,0,1,3,1,0
Human,2,1,2,3,2,1,2,3,150,2
Jaguar,3,1,3,1,3,1,2,1,81,1
Kangaroo,3,1,2,4,1,0,2,4,55,0
Lion,3,1,3,1,3,1,2,1,175,1
Mink,3,1,3,1,3,1,3,2,1,1
Mole,3,1,4,3,3,1,4,3,0.75,1
Moose,0,0,3,3,4,0,3,3,900,0
Mouse,1,0,0,3,1,0,0,3,0.3,2
Pig,3,1,4,3,3,1,4,3,50,2
Porcupine,1,0,1,3,1,0,1,3,3,0
Rabbit,2,0,3,3,1,0,2,3,1,0
Raccoon,3,1,4,2,3,1,4,2,40,2
Rat,1,0,0,3,1,0,0,3,.75,2
Red bat,1,1,2,3,3,1,2,3,1,1
Sea lion,3,1,4,1,2,1,4,1,415,1
Skunk,3,1,3,1,3,1,3,2,2,2
Squirrel,1,0,2,3,1,0,1,3,2,2
Wolf,3,1,4,2,3,1,4,3,27,1
Woodchuck,1,0,2,3,1,0,1,3,4,2
```

Figure 23.11 Mammal dentition

Figure 23.12 contains a function, `readMammalData`, for reading a file formatted in this way and processing the contents of the file to produce a set of examples representing the information in the file. It first processes the header information at the start of the file to get a count of the number of features to be associated with each example. It then uses the lines corresponding to each species to build three lists:

- `speciesNames` is a list of the names of the mammals.
- `labelList` is a list of the labels associated with the mammals.
- `featureVals` is a list of lists. Each element of `featureVals` contains the list of values, one for each mammal, for a single feature—for example, a list of weights. The value of the expression `featureVals[i][j]` is the $\texttt{i}^{th}$ feature of the $\texttt{j}^{th}$ mammal.

The last part of `readMammalData` uses the values in `featureVals` to create a list of feature vectors, one for each mammal. (The code could be simplified by not constructing `featureVals` and instead directly constructing the feature vectors for each mammal. However, we chose not to do that in anticipation of an enhancement to `readMammalData` that we make later in this section.)

The function `buildMammalExamples` in Figure 23.12 builds a list of examples from the data in

the lists created by `readMammalData`.

The function `testTeeth`, Figure 23.13, uses `trykmeans` to cluster the examples built by `buildMammalExamples`. It then reports the number of herbivores, carnivores, and omnivores in each cluster.

```
def readMammalData(fName):
    dataFile = open(fName, 'r')
    numFeatures = 0
    #Process lines at top of file
    for line in dataFile: #Find number of features
        if line[0:6] == '#Label': #indicates end of features
            break
        if line[0:5] != '#Name':
            numFeatures += 1
    featureVals = []

    #Produce featureVals, speciesNames, and labelList
    featureVals, speciesNames, labelList = [], [], []
    for i in range(numFeatures):
        featureVals.append([])

    #Continue processing lines in file, starting after comments
    for line in dataFile:
        #remove newline, then split
        dataLine = line[:-1].split(',')
        speciesNames.append(dataLine[0])
        classLabel = dataLine[-1]
        labelList.append(classLabel)
        for i in range(numFeatures):
            featureVals[i].append(float(dataLine[i+1]))

    #Use featureVals to build list containing the feature vectors
    #for each mammal
    featureVectorList = []
    for mammal in range(len(speciesNames)):
        featureVector = []
        for feature in range(numFeatures):
            featureVector.append(featureVals[feature][mammal])
        featureVectorList.append(featureVector)
    return featureVectorList, labelList, speciesNames

def buildMammalExamples(featureList, labelList, speciesNames):
    examples = []
    for i in range(len(speciesNames)):
```

```
        features = pylab.array(featureList[i])
        example = Example(speciesNames[i], features, labelList[i])
        examples.append(example)
    return examples
```

Figure 23.12 Read and process file

```
def testTeeth(numClusters, numTrials):
    features, labels, species = readMammalData('dentalFormulas.txt')
    examples = buildMammalExamples(features, labels, species)
    bestClustering = trykmeans(examples, numClusters, numTrials)
    for c in bestClustering:
        names = ''
        for p in c.members():
            names += p.getName() + ', '
        print('\n' + names[:-2]) #remove trailing comma and space
        herbivores, carnivores, omnivores = 0, 0, 0
        for p in c.members():
            if p.getLabel() == '0':
                herbivores += 1
            elif p.getLabel() == '1':
                carnivores += 1
            else:
                omnivores += 1
        print(herbivores, 'herbivores,', carnivores, 'carnivores,',
              omnivores, 'omnivores')
```

Figure 23.13 Clustering animals

When we executed the code `testTeeth(3, 40)` it printed

```
Bear, Cow, Deer, Elk, Fur seal, Grey seal, Lion, Sea lion
```

3 herbivores, 4 carnivores, 1 omnivores

Badger, Cougar, Dog, Fox, Guinea pig, Human, Jaguar, Kangaroo, Mink, Mole, Mouse, Pig, Porcupine, Rabbit, Raccoon, Rat, Red bat, Skunk, Squirrel, Wolf, Woodchuck

4 herbivores, 9 carnivores, 8 omnivores

Moose

1 herbivores, 0 carnivores, 0 omnivores

So much for our conjecture that the clustering would be related to the eating habits of the various species. A cursory inspection suggests that we have a clustering totally dominated by the weights of the animals. The problem is that the range of weights is much larger than the range of any of the other features. Therefore, when the Euclidean distance between examples is computed, the only feature that truly matters is weight.

We encountered a similar problem in Section 22.2 when we found that the distance between animals was dominated by the number of legs. We solved the problem there by turning the number of legs into a binary feature (legged or legless). That was fine for that data set, because all of the animals happened to have either zero or four legs. Here, however, there is no obvious way to turn weight into a single binary feature without losing a great deal of information.

This is a common problem, which is often addressed by scaling the features so that each feature has a mean of 0 and a standard deviation of 1,[170] as done by the function `zScaleFeatures` in Figure 23.14. It's easy to see why the statement `result = result - mean` ensures that the mean of the returned array will always be close to 0.[171] That the standard deviation will always be 1 is not obvious. It can be shown by a long and tedious chain of algebraic manipulations, which we will not bore you with. This kind of scaling is often called **z-scaling** because the normal distribution is sometimes referred to as the Z-distribution.

Another common approach to scaling is to map the minimum feature value to 0, map the maximum feature value to 1, and use linear interpolation in between, as done by the function `iScaleFeatures` in Figure 23.14.

```
def zScaleFeatures(vals):
    """Assumes vals is a sequence of floats"""
    result = pylab.array(vals)
    mean = sum(result)/len(result)
    result = result - mean
    return result/stdDev(result)

def iScaleFeatures(vals):
    """Assumes vals is a sequence of floats"""
    minVal, maxVal = min(vals), max(vals)
    fit = pylab.polyfit([minVal, maxVal], [0, 1], 1)
    return pylab.polyval(fit, vals)
```

Figure 23.14 Scaling attributes

Figure 23.15 contains a version of `readMammalData` that allows scaling of features using the function bound to the parameter `scale`. Notice that it depends upon that fact that we collect all of the values for a single feature into a single vector. The version of the function `testTeeth` in Figure 23.15 supplies the scaling function used by `readMammalData`. When `testTeeth` is called with only two arguments, it calls `readMammalData` with the identity function, which is equivalent to doing no scaling.

```
def readMammalData(fName, scale):
  Same code as in Figure 23.11

    #Produce featureVals, speciesNames, and labelList
    Same code as in Figure 23.11

    #Continue processing lines in file, starting after comments
    Same code as in Figure 23.11

    #Use featureVals to build list containing the feature vectors
    #for each mammal, scaling features as indicated
    for i in range(numFeatures):
        featureVals[i] = scale(featureVals[i])
    featureVectorList = []
    for mammal in range(len(speciesNames)):
        featureVector = []
        for feature in range(numFeatures):
            featureVector.append(featureVals[feature][mammal])
        featureVectorList.append(featureVector)
    return featureVectorList, labelList, speciesNames

def testTeeth(numClusters, numTrials, scale = lambda x: x):
    features, labels, species =\
              readMammalData('dentalFormulas.txt', scale)
    examples = buildMammalExamples(features, labels, species)

    ###Remainder of testTeeth is the same as in Figure 23.13###
```

Figure 23.15 Code that allows scaling of features

When we executed the code

```
random.seed(0) #so two clusterings starts with same seed
print('Clustering without scaling')
testTeeth(3, 40)
random.seed(0) #so two clusterings starts with same seed
print('\nClustering with z-scaling')
testTeeth(3, 40, zScaleFeatures)
print('\nClustering with i-scaling')
testTeeth(3, 40, iScaleFeatures)
```

it printed

Clustering without scaling

Bear, Cow, Deer, Elk, Fur seal, Grey seal, Lion, Sea lion

3 herbivores, 4 carnivores, 1 omnivores

Badger, Cougar, Dog, Fox, Guinea pig, Human, Jaguar, Kangaroo, Mink, Mole, Mouse, Pig, Porcupine, Rabbit, Raccoon, Rat, Red bat, Skunk, Squirrel, Wolf, Woodchuck

4 herbivores, 9 carnivores, 8 omnivores

Moose

1 herbivores, 0 carnivores, 0 omnivores

Clustering with z-scaling

Badger, Bear, Cougar, Dog, Fox, Fur seal, Grey seal, Human, Jaguar, Lion, Mink, Mole, Pig, Raccoon, Red bat, Sea lion, Skunk, Wolf

0 herbivores, 13 carnivores, 5 omnivores

Guinea pig, Kangaroo, Mouse, Porcupine, Rabbit, Rat, Squirrel, Woodchuck

4 herbivores, 0 carnivores, 4 omnivores

Cow, Deer, Elk, Moose

4 herbivores, 0 carnivores, 0 omnivores

Clustering with i-scaling

Cow, Deer, Elk, Moose

4 herbivores, 0 carnivores, 0 omnivores

Badger, Bear, Cougar, Dog, Fox, Fur seal, Grey seal, Human, Jaguar, Lion, Mink, Mole, Pig, Raccoon, Red bat, Sea lion, Skunk, Wolf

0 herbivores, 13 carnivores, 5 omnivores

Guinea pig, Kangaroo, Mouse, Porcupine, Rabbit, Rat, Squirrel, Woodchuck

4 herbivores, 0 carnivores, 4 omnivores

The clustering with scaling (the two methods of scaling yield the same clusters) does not perfectly partition the animals based upon their eating habits, but it is certainly correlated with what they eat. It does a good job of separating the carnivores from the herbivores, but there is no obvious pattern in where the omnivores appear. This suggests that perhaps features other than dentition and weight might be needed to separate omnivores from herbivores and carnivores.

---

[165] Though k-means clustering is probably the most commonly used clustering method, it is not the most appropriate method in all situations. Two other widely used methods, not covered in this book, are hierarchical clustering and EM-clustering.

[166] The most widely used k-means algorithm is attributed to James McQueen, and was first published in 1967. However, other approaches to k-means clustering were used as early as the 1950s.

[167] Unfortunately, in many applications we need to use a distance metric, e.g., earth-movers distance or dynamic-time-warping distance, that has a higher computational complexity.

[168] Or, perhaps, species have chosen food based on their dentition. As we pointed out in Section 21.4, correlation does not imply causation.

[169] We included the information about weight because the author has been told, on more than one occasion, that there is a relationship between weight and eating habits.

[170] A normal distribution with a mean of 0 and a standard deviation of 1 is called a **standard normal distribution**.

[171] We say "close," because floating point numbers are only an approximation to the reals.

24 CLASSIFICATION METHODS

The most common application of supervised machine learning is building classification models. A **classification model,** or classifier, is used to label an example as belonging to one of a finite set of categories. Deciding whether an email message is spam, for example, is a classification problem. In the literature, these categories are typically called **classes** (hence the name classification). Equivalently, one can describe an example as belonging to a class or a having **label.**

In **one-class learning,** the training set contains examples drawn from only one class. The goal is to learn a model that predicts whether an example belongs to that class. One-class learning is useful when it is difficult to find training examples that lie outside the class. One-class learning is frequently used for building anomaly detectors, e.g., detecting previously unseen kinds of attacks on a computer network.

In **two-class learning** (often called **binary classification**), the training set contains examples drawn from exactly two classes (typically called positive and negative), and the objective is to find a boundary that separates the two classes. **Multi-class learning** involves finding boundaries that separate more than two classes from each other.

In this chapter, we look at two widely used supervised learning methods for solving classification problems: k-nearest neighbors and regression. Before we do, we address the question of how to evaluate the classifiers produced by these methods.

24.1 Evaluating Classifiers

Those of you who read Chapter 18 might recall that part of that chapter addressed the question of choosing a degree for a linear regression that would 1) provide a reasonably good fit for the available data, and 2) have a reasonable chance of making good predictions about as yet unseen data. The same issues arise when using supervised machine learning to train a classifier.

We start by dividing our data into two sets, a training set and a **test set.** The training set is used to learn a model, and the test set is used to evaluate that model. When we train the classifier, we attempt to minimize **training error**, i.e., errors in classifying the examples in the training set, subject to certain constraints. The constraints are designed to increase the probability that the model will perform reasonably well on as yet unseen data. Let's look at this pictorially.

The chart on the left of Figure 24.1 shows a representation of voting patterns for sixty (simulated) American citizens. The x-axis is the distance of the voter's home from Boston, Massachusetts. The y-

axis is the age of the voter. The stars indicate voters who usually vote Democratic, and the triangles voters who usually vote Republican. The chart on the right in Figure 24.1 shows a training set containing a randomly chosen sample of thirty of those voters. The solid and dashed lines show two possible boundaries between the two populations. For the model based on the solid line, points below the line are classified as Democratic voters. For the model based on the dotted line, points to the left of the line are classified as Democratic voters.

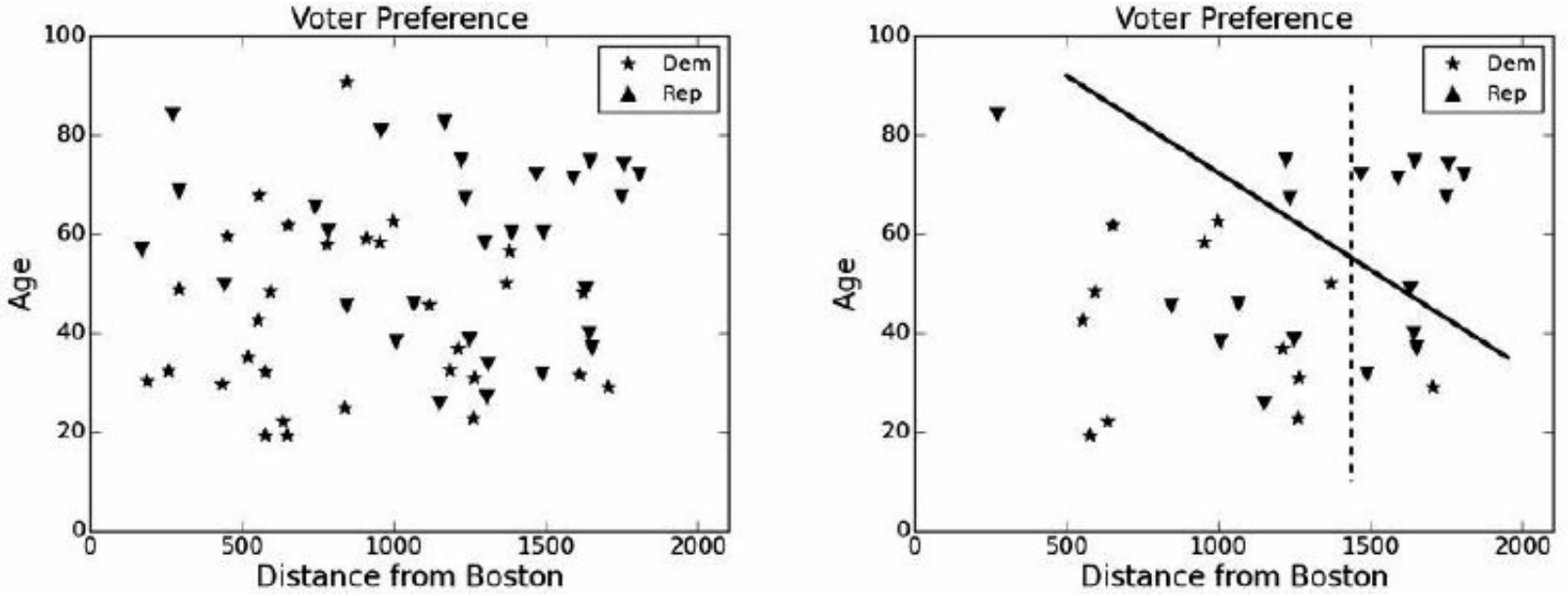

Figure 24.1 Plots of voter preferences

Neither boundary separates the training data perfectly. The training errors for the two models are shown in the **confusion matrices** in Figure 24.2. The top left corner of each shows the number of examples classified as Democratic that are actually Democratic, i.e., the true positives. The bottom left corner shows the number of examples classified as Democratic that are actually Republican, i.e., the false positives. The right-hand column shows the number of false negatives on the top and the number of true negatives on the bottom.

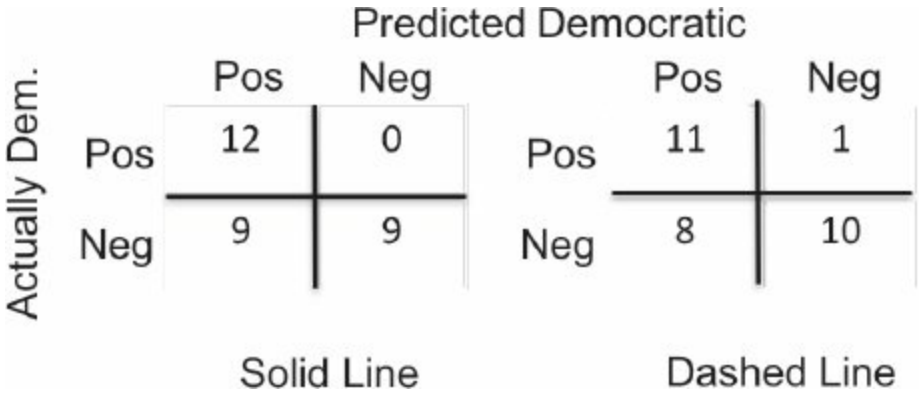

Figure 24.2 Confusion matrices

The **accuracy** of each classifier on the training data can be calculated as

$$accuracy = \frac{true\ positive + true\ negative}{true\ positive + true\ negative + false\ positive + false\ negative}$$

In this case, each classifier has an accuracy of 0.7. Which does a better job of fitting the training data? It depends upon whether one is more concerned about misclassifying Republicans as Democrats, or vice versa.

If we are willing to draw a more complex boundary, we can get a classifier that does a more accurate job of classifying the training data. The classifier pictured in Figure 24.3, for example, has an accuracy of about 0.83 on the training data, as depicted in the left plot of the figure. However, as we saw in our discussion of linear regression in Chapter 18, the more complicated the model, the higher the probability that it has been overfit to the training data. The right-hand plot in Figure 24.3 depicts what happens if we apply the complex model to the holdout set—the accuracy drops to 0.6.

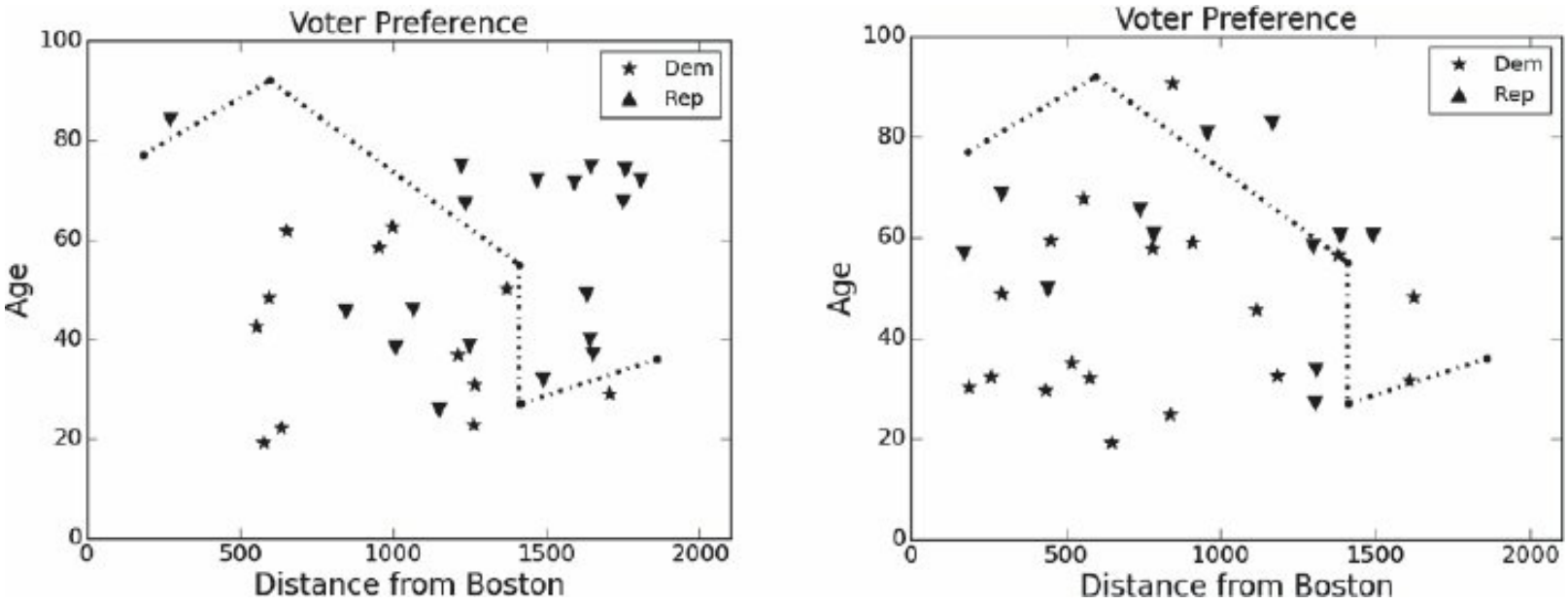

Figure 24.3 A more complex model

Accuracy is a reasonable way to evaluate a classifier when the two classes are of roughly equal size. It is a terrible way to evaluate a classifier when there is a large **class imbalance.** Imagine that you are charged with evaluating a classifier that predicts whether a person has a potentially fatal disease that occurs in about 0.1% of the population to be tested. Accuracy is not a particularly useful statistic, since 99.9% accuracy can be attained by merely declaring all patients disease-free. That classifier might seem great to those charged with paying for the treatment (nobody would get treated!), but it might not seem so great to those worried that they might have the disease.

Fortunately, there are statistics about classifiers that shed light when classes are imbalanced:

$$sensitivity = \frac{true\ positive}{true\ positive + false\ negative}$$

$$specificity = \frac{true\ negative}{true\ negative + false\ positive}$$

$$positive\ predictive\ value = \frac{true\ positive}{true\ positive + false\ positive}$$

$$negative\ predictive\ value = \frac{true\ negative}{true\ negative + false\ negative}$$

Sensitivity (called **recall** in some fields) is the true positive rate, i.e., the proportion of positives that are correctly identified as such. **Specificity** (called **precision** in some fields) is the true negative rate, i.e., the proportion of negatives that are correctly identified as such. **Positive predictive** value is the probability that an example classified as positive is truly positive. **Negative predictive** value is the probability that an example classified as negative is truly negative.

Implementations of these statistical measures and a function that uses them to generate some statistics are in Figure 24.4. We will use these functions later in this chapter.

```
def accuracy(truePos, falsePos, trueNeg, falseNeg):
    numerator = truePos + trueNeg
    denominator = truePos + trueNeg + falsePos + falseNeg
    return numerator/denominator

def sensitivity(truePos, falseNeg):
    try:
        return truePos/(truePos + falseNeg)
    except ZeroDivisionError:
        return float('nan')

def specificity(trueNeg, falsePos):
    try:
```

```
    try:
        return trueNeg/(trueNeg + falsePos)
    except ZeroDivisionError:
        return float('nan')

def posPredVal(truePos, falsePos):
    try:
        return truePos/(truePos + falsePos)
    except ZeroDivisionError:
        return float('nan')

def negPredVal(trueNeg, falseNeg):
    try:
        return trueNeg/(trueNeg + falseNeg)
    except ZeroDivisionError:
        return float('nan')

def getStats(truePos, falsePos, trueNeg, falseNeg, toPrint = True):
    accur = accuracy(truePos, falsePos, trueNeg, falseNeg)
    sens = sensitivity(truePos, falseNeg)
    spec = specificity(trueNeg, falsePos)
    ppv = posPredVal(truePos, falsePos)
    if toPrint:
        print(' Accuracy =', round(accur, 3))
        print(' Sensitivity =', round(sens, 3))
        print(' Specificity =', round(spec, 3))
        print(' Pos. Pred. Val. =', round(ppv, 3))
    return (accur, sens, spec, ppv)
```

Figure 24.4 Functions for evaluating classifiers

24.2 Predicting the Gender of Runners

Earlier in this book, we used data from the Boston Marathon to illustrate a number of statistical concepts. We will now use the same data to illustrate the application of various classification methods. The task is to predict the gender of a runner given the runner's age and finishing time.

The code in Figure 24.5 reads in the data from a file by calling the function `getBMData` defined in Figure 17.2, and then builds a set of examples. Each example is an instance of class Runner. Each runner has a label (gender) and a feature vector (age and finishing time). The only interesting method in `Runner` is `featureDist`. It returns the Euclidean distance between the feature vectors of two runners.

The next step is to split the examples into a training set and a held-out test set. As is frequently

done, we use 80% of the data for training, and test on the remaining 20%. This is done using the function `divide80_20` at the bottom of Figure 24.5. Notice that we select the training data at random. It would have taken less code to simply select the first 80% of the data, but that runs the risk of not being representative of the set as a whole. If the file had been sorted by finishing time, for example, we would get a training set biased towards the better runners.

We are now ready to look at different ways of using the training set to build a classifier that predicts the gender of a runner. Inspection reveals that 58% of the runners in the training set are male. So, if we guess male all the time, we should expect an accuracy of 58%. Keep this baseline in mind when looking at the performance of more sophisticated classification algorithms.

24.3 K-nearest Neighbors

K-nearest neighbors (KNN) is probably the simplest of all classification algorithms. The "learned" model is simply the training examples themselves. New examples are assigned a label based on how similar they are to examples in the training data.

```
class Runner(object):
    def __init__(self, gender, age, time):
        self.featureVec = (age, time)
        self.label = gender

    def featureDist(self, other):
        dist = 0.0
        for i in range(len(self.featureVec)):
            dist += abs(self.featureVec[i] - other.featureVec[i])**2
        return dist**0.5

    def getTime(self):
        return self.featureVec[1]
    def getAge(self):
        return self.featureVec[0]
    def getLabel(self):
        return self.label
    def getFeatures(self):
        return self.featureVec

    def __str__(self):
        return str(self.getAge()) + ', ' + str(self.getTime())\
               + ', ' + self.label

def buildMarathonExamples(fileName):
    data = getBMData(fileName)
```

```
    examples = []
    for i in range(len(data['age'])):
        a = Runner(data['gender'][i], data['age'][i],
                   data['time'][i])
        examples.append(a)
    return examples

def divide80_20(examples):
    sampleIndices = random.sample(range(len(examples)),
                                  len(examples)//5)
    trainingSet, testSet = [], []
    for i in range(len(examples)):
        if i in sampleIndices:
            testSet.append(examples[i])
        else:
            trainingSet.append(examples[i])
    return trainingSet, testSet
```

Figure 24.5 Build examples and divide data into training and test sets

Imagine that you and a friend are strolling through the park and spot a bird. You believe that it is a yellow-throated woodpecker, but your friend is pretty sure that it is a golden-green woodpecker. You rush home and dig out your cache of bird books (or, if you are under 35, go to your favorite search engine) and start looking at labeled pictures of birds. Think of these labeled pictures as the training set. None of the pictures is an exact match for the bird you saw, so you settle for selecting the five that look the most like the bird you saw (the five "nearest neighbors"). The majority of them are photos of a yellow-throated woodpecker—you declare victory.

A weakness of KNN classifiers is that they often give poor results when there is a large class imbalance. If the frequency of pictures of bird species in the book is the same as the frequency of that species in your neighborhood, KNN will probably work well. Suppose, however, that despite the species being equally common, your books contain 30 pictures of yellow-throated woodpeckers and only one of a golden-green woodpecker. If a simple majority vote is used to determine the classification, the yellow-throated woodpecker will be chosen even if the photos don't look much like the bird you saw. This problem can be mitigated by using a more complicated voting scheme in which the k-nearest neighbors are weighted based on their similarity to the example being classified.

The functions in Figure 24.6 implement a k-nearest neighbors classifier that predicts the gender of a runner based on the runner's age and finishing time. The implementation is brute force. The function `findKNearest` is linear in the number of examples in `exampleSet`, since it computes the feature distance between `example` and each element in `exampleSet`. The function `kNearestClassify` uses a simple majority-voting scheme to do the classification. The complexity of `kNearestClassify` is `O(len(training)*len(testSet))`, since it calls the function `findNearest` a total of `len(testSet)` times.

When the code

```
examples = buildMarathonExamples('bm_results2012.txt')
training, testSet = divide80_20(examples)
truePos, falsePos, trueNeg, falseNeg =\
      KNearestClassify(training, testSet, 'M', 9)
getStats(truePos, falsePos, trueNeg, falseNeg)
```

was run, it printed

```
Accuracy = 0.65
Sensitivity = 0.715
Specificity = 0.563
Pos. Pred. Val. = 0.684
```

```
def findKNearest(example, exampleSet, k):
    kNearest, distances = [], []
    #Build lists containing first k examples and their distances
    for i in range(k):
        kNearest.append(exampleSet[i])
        distances.append(example.featureDist(exampleSet[i]))
    maxDist = max(distances) #Get maximum distance
    #Look at examples not yet considered
    for e in exampleSet[k:]:
        dist = example.featureDist(e)
        if dist < maxDist:
            #replace farther neighbor by this one
            maxIndex = distances.index(maxDist)
            kNearest[maxIndex] = e
            distances[maxIndex] = dist
            maxDist = max(distances)
    return kNearest, distances

def KNearestClassify(training, testSet, label, k):
    """Assumes training and testSet lists of examples, k an int
       Uses a k-nearest neighbor classifier to predict
         whether each example in testSet has the given label
       Returns number of true positives, false positives,
          true negatives, and false negatives"""
    truePos, falsePos, trueNeg, falseNeg = 0, 0, 0, 0
    for e in testSet:
        nearest, distances = findKNearest(e, training, k)
        #conduct vote
        numMatch = 0
        for i in range(len(nearest)):
            if nearest[i].getLabel() == label:
```

```
                numMatch += 1
        if numMatch > k//2: #guess label
            if e.getLabel() == label:
                truePos += 1
            else:
                falsePos += 1
        else: #guess not label
            if e.getLabel() != label:
                trueNeg += 1
            else:
                falseNeg += 1
    return truePos, falsePos, trueNeg, falseNeg
```

Figure 24.6 Finding the k-nearest neighbors

Should we be pleased that we can predict gender with 65% accuracy given age and finishing time? One way to evaluate a classifier is to compare it to a classifier that doesn't even look at age and finishing time. The classifier in Figure 24.7 first uses the examples in `training` to estimate the probability of a randomly chosen example in `testSet` being from class `label`. Using this prior probability, it then randomly assigns a label to each example in `testSet`.

When we test `prevalenceClassify` on the same Boston Marathon data on which we tested KNN, it prints

```
Accuracy = 0.514
Sensitivity = 0.593
Specificity = 0.41
Pos. Pred. Val. = 0.57
```

indicating that we are reaping a considerable advantage from considering age and finishing time.

That advantage has a cost. If you run the code in Figure 24.6, you will notice that it takes a rather long time to finish. There are 17,233 training examples and 4,308 test examples, so there are nearly 75 million distances calculated. This raises the question of whether we really need to use all of the training examples. Let's see what happens if we simply **down sample** the training data by a factor of 10

If we run

```
reducedTraining = random.sample(training, len(training)//10)
truePos, falsePos, trueNeg, falseNeg =\
      KNearestClassify(reducedTraining, testSet, 'M', 9)
getStats(truePos, falsePos, trueNeg, falseNeg)
```

it completes in one-tenth the time, with little change in classification performance:

```
Accuracy = 0.643
Sensitivity = 0.726
Specificity = 0.534
Pos. Pred. Val. = 0.673
```

In practice, when people apply KNN to large data sets they do down sample the training data.[172]

```
def prevalenceClassify(training, testSet, label):
    """Assumes training and testSet lists of examples
       Uses a prevalence-based classifier to predict
         whether each example in testSet is of class label
       Returns number of true positives, false positives,
          true negatives, and false negatives"""
    numWithLabel = 0
    for e in training:
        if e.getLabel()== label:
            numWithLabel += 1
    probLabel = numWithLabel/len(training)
    truePos, falsePos, trueNeg, falseNeg = 0, 0, 0, 0
    for e in testSet:
        if random.random() < probLabel: #guess label
            if e.getLabel() == label:
                truePos += 1
            else:
                falsePos += 1
        else: #guess not label
            if e.getLabel() != label:
                trueNeg += 1
            else:
                falseNeg += 1
    return truePos, falsePos, trueNeg, falseNeg
```

Figure 24.7 Prevalence-based classifier

In the above experiments, we set k to 9. We did not choose this number for its role in science (the number of planets in our solar system),[173] its religious significance (the number of forms of the Hindu goddess Durga), or its sociological importance (the number of hitters in a baseball lineup). Instead, we learned k from the training data by using the code in Figure 24.8 to search for a good k.

The outer loop tests a sequence of values for k. We test only odd values to ensure that when the vote is taken in `kNearestClassify` there will always be a majority for one gender or the other.

The inner loop tests each value of k using **n-fold cross validation**. In each of the numFolds iterations of the loop, the original training set is split into a new training set/test set pair. We then compute the accuracy of classifying the new test set using k-nearest neighbors and the new training set. When we exit the inner loop, we calculate the average accuracy of the `numFolds` folds.

```
def findK(training, minK, maxK, numFolds, label):
    #Find average accuracy for range of odd values of k
    accuracies = []
    for k in range(minK, maxK + 1, 2):
        score = 0.0
        for i in range(numFolds):
            #downsample to reduce computation time
            fold = random.sample(training, min(5000, len(training)))
            examples, testSet = divide80_20(fold)
            truePos, falsePos, trueNeg, falseNeg =\
                KNearestClassify(examples, testSet, label, k)
            score += accuracy(truePos, falsePos, trueNeg, falseNeg)
        accuracies.append(score/numFolds)
    pylab.plot(range(minK, maxK + 1, 2), accuracies)
    pylab.title('Average Accuracy vs k (' + str(numFolds)\
                + ' folds)')
    pylab.xlabel('k')
    pylab.ylabel('Accuracy')

findK(training, 1, 21, 1, 'M')
```

Figure 24.8 Searching for a good k

When we ran the code, it produced the plot in Figure 24.9. As we can see, 17 was the value of k that led to the best accuracy across 5 folds. Of course, there is no guarantee that some value larger than 21 might not have been even better. However, once k reached 9, the accuracy fluctuated over a reasonably narrow range, so we chose to use 9.

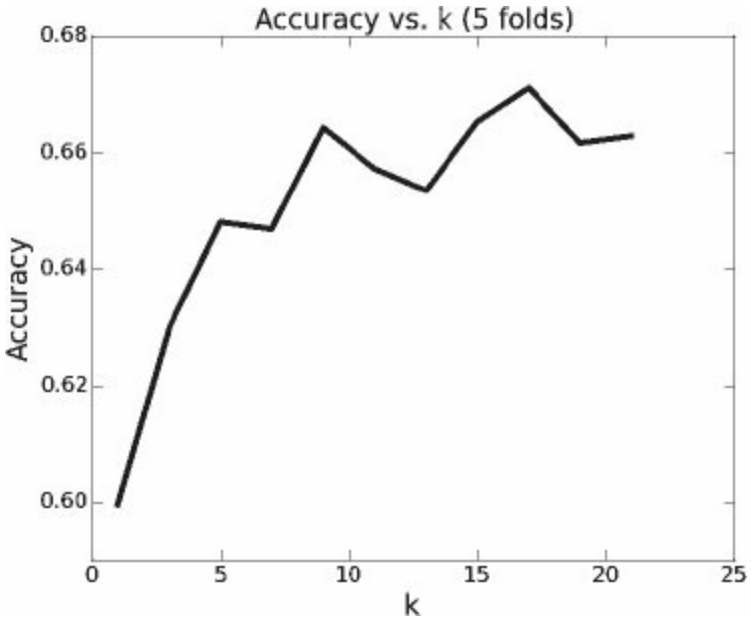

Figure 24.9 Choosing a value for k

24.4 Regression-based Classifiers

In Chapter 18 we used linear regression to build models of data. We do the same thing here, and use the training data to build separate models for the men and the women. The plot in Figure 24.11 was produced by the code in Figure 24.10.

```
#Build training sets for men and women
ageM, ageW, timeM, timeW = [], [], [], []
for e in training:
    if e.getLabel() == 'M':
        ageM.append(e.getAge())
        timeM.append(e.getTime())
    else:
        ageW.append(e.getAge())
        timeW.append(e.getTime())
#downsample to make plot of examples readable
ages, times = [], []
for i in random.sample(range(len(ageM)), 300):
    ages.append(ageM[i])
    times.append(timeM[i])
#Produce scatter plot of examples
pylab.plot(ages, times, 'yo', markersize = 6, label = 'Men')
ages, times = [], []
for i in random.sample(range(len(ageW)), 300):
    ages.append(ageW[i])
    times.append(timeW[i])
pylab.plot(ages, times, 'k^', markersize = 6, label = 'Women')
#Learn two first-degree linear regression models
mModel = pylab.polyfit(ageM, timeM, 1)
fModel = pylab.polyfit(ageW, timeW, 1)
#Plot lines corresponding to models
xmin, xmax = 15, 85
pylab.plot((xmin, xmax), (pylab.polyval(mModel,(xmin, xmax))),
           'k', label = 'Men')
pylab.plot((xmin, xmax), (pylab.polyval(fModel,(xmin, xmax))),
           'k--', label = 'Women')
pylab.title('Linear Regression Models')
pylab.xlabel('Age')
pylab.ylabel('Finishing time (minutes)')
pylab.legend()
```

Figure 24.10 Produce and plot linear regression models

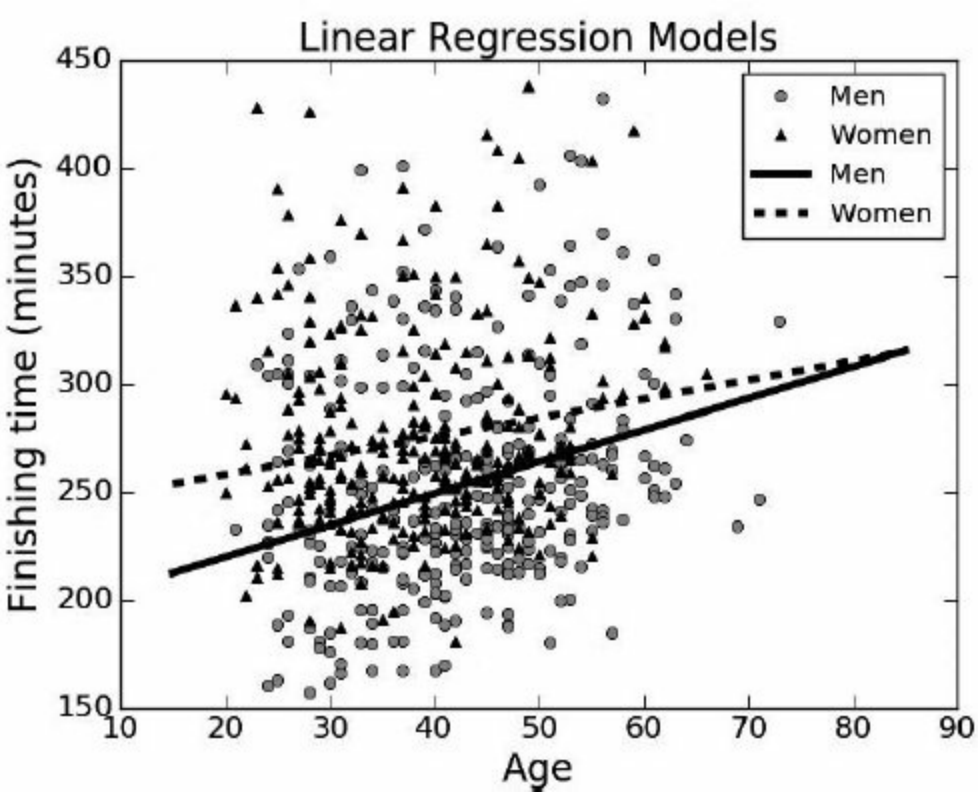

Figure 24.11 Linear regression models for men and women

A quick glance at Figure 24.11 is enough to see that the linear regression models explain only a small amount of the variance in the data.[174] Nevertheless, it is possible to use these models to build a classifier. Each model attempts to capture the relationship between age and finishing time. This relationship is different for men and women, a fact we can exploit in building a classifier. Given an example, we ask whether the relationship between age and finishing time is closer to the relationship predicted by the model for male runners (the solid line) or to the model for female runners (the dashed line). This idea is implemented in Figure 24.12.

When the code is run, it prints

```
Accuracy = 0.616
Sensitivity = 0.682
Specificity = 0.529
Pos. Pred. Val. = 0.657
```

The results are better than random, but a bit worse than for KNN.

```
truePos, falsePos, trueNeg, falseNeg = 0, 0, 0, 0
for e in testSet:
    age = e.getAge()
    time = e.getTime()
    if abs(time - pylab.polyval(mModel,age)) <\
       abs(time - pylab.polyval(fModel, age)):
        if e.getLabel() == 'M':
            truePos += 1
        else:
            falsePos += 1
    else:
        if e.getLabel() == 'F':
            trueNeg += 1
        else:
            falseNeg += 1
getStats(truePos, falsePos, trueNeg, falseNeg)
```

Figure 24.12 Using linear regression to build a classifier

You might be wondering why we took this indirect approach to using linear regression, rather than explicitly building a model using some function of age and time as the dependent variable and real numbers (say 0 for female and 1 for male) as the dependent variable.

We could easily build such a model using `polyfit` to map a function of age and time to a real number. However, what would it mean to predict that some runner is halfway between male and female? Were there some hermaphrodites in the race? Perhaps we can interpret the y-axis as the probability that a runner is male. Not really. There is not even a guarantee that applying `polyval` to the model will return a value between 0 and 1.

Fortunately, there is a form of regression, **logistic regression,**[175] designed explicitly for predicting the probability of an event. The Python library `sklearn`[176] provides a good implementation of logistic regression and of many other useful functions and classes related to machine learning.

The module `sklearn.linear_model` contains the class `LogisticRegression`. The `__init__` method of this class has a large number of parameters that control things such as the optimization algorithm used to solve the regression equation. They all have default values, and on most occasions it is fine to stick with those.

The central method of class `LogisticRegression` is `fit`. The method takes as arguments two sequences (tuples, lists, or arrays) of the same length. The first is a sequence of feature vectors and

the second a sequence of the corresponding labels. In the literature, these labels are typically called **outcomes**.

`The fit` method returns an object of type `LogisticRegression` for which coefficients have been learned for each feature in the feature vector. These coefficients, often called **feature weights,** capture the relationship between the feature and the outcome. A positive feature weight implies that there is a positive correlation between the feature and the outcome, and a negative feature weight implies a negative correlation. The absolute magnitude of the weight is related to the strength of the correlation.[177] The values of these weights can be accessed using the `coef_` attribute of `LogisticRegression`. Since it is possible to train a `LogisticRegression` object on multiple outcomes (called classes in the documentation for the package), the value of `coef_` is a sequence in which each element contains the sequence of weights associated with a single outcome. So, for example, the expression `model.coef[1][0]` denotes the value of the coefficient of the first feature for the second outcome.

Once the coefficients have been learned, the method `predict_proba` of the `LogisticRegression` class can be used to predict the outcome associated with a feature vector. The method `predict_proba` takes a single argument (in addition to `self`), a sequence of feature vectors. It returns an array of arrays, one per feature vector. Each element in the returned array contains a prediction for the corresponding feature vector. The reason that the prediction is an array is that it contains a probability for each label used in building `model`.

The code in Figure 24.13 contains a simple illustration of how this all works. It first creates a list of 100,000 examples, each of which has a feature vector of length 2 and is labeled either `'A'`, `'B'`, `'C'`, or `'D'`. The first two feature values for each example are drawn from a Gaussian with a standard deviation of 0.5, but the means vary depending upon the label. The value of third feature is chosen at random, and therefore should not be useful in predicting the label. After creating the examples, the code generates a logistic regression model, prints the feature weights, and finally the probabilities associated with four examples.

```
import sklearn.linear_model

featureVecs, labels = [], []
for i in range(25000): #create 4 examples in each iteration
    featureVecs.append([random.gauss(0, 0.5), random.gauss(0, 0.5),
                        random.random()])
    labels.append('A')
    featureVecs.append([random.gauss(0, 0.5), random.gauss(2, 0.),
                        random.random()])
    labels.append('B')
    featureVecs.append([random.gauss(2, 0.5), random.gauss(0, 0.5),
                        random.random()])
    labels.append('C')
    featureVecs.append([random.gauss(2, 0.5), random.gauss(2, 0.5),
                        random.random()])
    labels.append('D')
model = sklearn.linear_model.LogisticRegression().fit(featureVecs,
                                                      labels)
print('model.classes_ =', model.classes_)
for i in range(len(model.coef_)):
    print('For label', model.classes_[i],
          'feature weights =', model.coef_[i])
print('[0, 0] probs =', model.predict_proba([[0, 0, 1]])[0])
print('[0, 2] probs =', model.predict_proba([[0, 2, 2]])[0])
print('[2, 0] probs =', model.predict_proba([[2, 0, 3]])[0])
print('[2, 2] probs =', model.predict_proba([[2, 2, 4]])[0])
```

Figure 24.13 Using `sklearn` to do multi-class logistic regression

When the code in Figure 24.13 was run, it printed

```
model.classes_ = [’A' 'B' 'C' 'D']
For label A feature weights = [-4.65720783 -4.38351299 -0.00722845]
For label B feature weights = [-5.17036683 5.82391837 0.04706108]
For label C feature weights = [ 3.95940539 -3.97854738 -0.04480206]
For label D feature weights = [ 4.37529465 5.40639909 -0.09434664]
[0, 0] probs = [ 9.90019074e-01 4.66294343e-04 9.51434182e-03
2.90294956e-07]
[0, 2] probs = [ 8.72562747e-03 9.78468475e-01 3.18006160e-06
1.28027180e-02]
[2, 0] probs = [ 5.22466887e-03 1.69995686e-08 9.93218655e-01
1.55665885e-03]
```

```
[2, 2] probs = [ 7.88542473e-07 1.97601741e-03 7.99527347e-03
9.90027921e-01]
```

Let's look first at the feature weights. The first line tells us that the first two features have roughly the same weight and are negatively correlated with the probability of an example having label `'A'`.[178] I.e., the larger the value of the first two features, the less likely that the example is of type `'A'`. The third feature, which we expect to have little value in predicting the label, has a small value relative to the other two values, indicating that it is relatively unimportant. The second line tells us that the probability of an example having the label `'B'` is negatively correlated with value of the first feature, but positively with the second feature. Again, the third feature has a relatively small value. The third and four lines are mirror images of the first two lines.

Now, let's look at the probabilities associated with the four examples. The order of the probabilities corresponds to the order of the outcomes in the attribute `model.classes_`. As you would hope, when we predict the label associated with the feature vector `[0, 0]`, `'A'` has a very high probability and `'D'` a very low probability. Similarly, `[2, 2]` has a very high probability for `'D'` and a very low one for `'A'`. The probabilities associated with the middle two examples are also as expected.

The example in Figure 24.14 is similar to the one in Figure 24.13, except that we create examples of only two classes, `'A'` and `'D'`, and don't include the irrelevant third feature.

```
featureVecs, labels = [], []
for i in range(20000):
    featureVecs.append([random.gauss(0, 0.5), random.gauss(0, 0.5)])
    labels.append('A')
    featureVecs.append([random.gauss(2, 0.5), random.gauss(2, 0.5)])
    labels.append('D')
model = sklearn.linear_model.LogisticRegression().fit(featureVecs,
                                                      labels)
print('model.coef =', model.coef_)
print('[0, 0] probs =', model.predict_proba([[0, 0]])[0])
print('[0, 2] probs =', model.predict_proba([[0, 2]])[0])
print('[2, 0] probs =', model.predict_proba([[2, 0]])[0])
print('[2, 2] probs =', model.predict_proba([[2, 2]])[0])
```

Figure 24.14 Example of two-class logistic regression

When we run the code in Figure 24.14 it printed

```
model.coef = [[ 5.79284554 5.68893473]]
[0, 0] probs = [ 9.99988836e-01 1.11643397e-05]
[0, 2] probs = [ 0.50622598 0.49377402]
[2, 0] probs = [ 0.45439797 0.54560203]
[2, 2] probs = [ 9.53257749e-06 9.99990467e-01]
```

Notice that there is only one set of weights in `coef_`. When `fit` is used to produce a model for a binary classifier, it only produces weights for one label. This is sufficient because once `proba` has calculated the probability of an example being in either of the classes, the probability of it being in the other class is determined—since the probabilities must add up to 1. To which of the two labels do the weights in `coef_` correspond? Since the weights are negative, they must correspond to `'D'`, since we know that the larger the values in the feature vector, the more likely the example is of class `'D'`. Traditionally, binary classification uses the labels 0 and 1, and the classifier uses the weights for 1. In this case, `coef_` contains the weights associated with largest label, as defined by the `>` operator for type `str`.

Let's return to the Boston Marathon example. The code in Figure 24.15 uses the `LogisticRegression` class to build and test a model for our Boston Marathon data. The function `applyModel` takes four arguments:

- `model`: an object of type `LogisticRegression` for which a fit has been constructed
- `testSet`: a sequence of examples. The examples have the same kinds of features and labels used in constructing the fit for `model`.
- `label`: The label of the positive class. The confusion matrix information returned by `applyModel` is relative to this label.
- `prob`: the probability threshold to be used in deciding which label to assign to an example in `testSet`. The default value is `0.5`. Because it is not a constant, `applyModel` can be used to investigate the tradeoff between false positives and false negatives.

The implementation of `applyModel` first uses list comprehension (Section 5.3.2) to build a list whose elements are the feature vectors of the examples in `testSet`. It then calls `model.predict_proba` to get an array of pairs corresponding to the prediction for each feature vector. Finally, it compares the prediction against the label associated with the example with that feature vector, and keeps track of and returns the number of true positives, false positives, true negatives, and false negatives.

```
def applyModel(model, testSet, label, prob = 0.5):
    #Create vector containing feature vectors for all test examples
    testFeatureVecs = [e.getFeatures() for e in testSet]
    probs = model.predict_proba(testFeatureVecs)
    truePos, falsePos, trueNeg, falseNeg = 0, 0, 0, 0
    for i in range(len(probs)):
        if probs[i][1] > prob:
            if testSet[i].getLabel() == label:
                truePos += 1
            else:
                falsePos += 1
        else:
            if testSet[i].getLabel() != label:
                trueNeg += 1
            else:
                falseNeg += 1
    return truePos, falsePos, trueNeg, falseNeg

examples = buildMarathonExamples('bm_results2012.txt')
training, test = divide80_20(examples)

featureVecs, labels = [], []
for e in training:
    featureVecs.append([e.getAge(), e.getTime()])
    labels.append(e.getLabel())
model = sklearn.linear_model.LogisticRegression().fit(featureVecs,
                                                      labels)
print('Feature weights for label M:',
      'age =', str(round(model.coef_[0][0], 3)) + ',',
      'time =', round(model.coef_[0][1], 3))
truePos, falsePos, trueNeg, falseNeg = \
                        applyModel(model, test, 'M', 0.5)
getStats(truePos, falsePos, trueNeg, falseNeg)
```

Figure 24.15 Use logistic regression to predict Gender

When the code is run, it prints

```
Feature weights for label M: age = 0.055, time = -0.011
 Accuracy = 0.635
 Sensitivity = 0.831
 Specificity = 0.377
```

```
Pos. Pred. Val. = 0.638
```

Let's compare these results to what we got when we used KNN:

```
Accuracy = 0.65
Sensitivity = 0.715
Specificity = 0.563
Pos. Pred. Val. = 0.684
```

The accuracies and positive predictive values are similar, but logistic regression has a much higher sensitivity and a much lower specificity. That makes the two methods hard to compare. We can address this problem by adjusting the probability threshold used by `applyModel` so that it has approximately the same sensitivity as KNN. We can find that probability by iterating over values of `prob` until we get a sensitivity close to that we got using KNN.

If we call `applyModel` with `prob = 0.578` instead of `0.5`, we get the results

```
Accuracy = 0.659
Sensitivity = 0.714
Specificity = 0.586
Pos. Pred. Val. = 0.695
```

I.e., the models have similar performance.

Since it is easy to explore the ramifications of changing the decision threshold for a linear regression model, people often use something called the **receiver operating characteristic curve,**[179] or **ROC curve** to visualize the tradeoff between sensitivity and specificity. The curve plots the true positive rate (sensitivity) against the false positive rate (1 – specificity) for multiple decision thresholds.

ROC curves are often compared to one another by computing the area under the curve (**AUROC**). This area is equal to the probability that the model will assign a higher probability of being positive to a randomly chosen positive example than to a randomly chosen negative example. This is known as the **discrimination** of the model. It is important to keep in mind that discrimination says nothing about the accuracy, often called the **calibration**, of the probabilities. One could, for example, divide all of the estimated probabilities by 2 without changing the discrimination—but it would certainly change the accuracy of the estimates.

The code in Figure 24.16 plots the ROC curve for the logistic regression classifier as a solid line, Figure 24.17. The dotted line is the ROC for a random classifier—a classifier that chooses the label randomly. We could have computed the AUROC by first interpolating (because we have only a discrete number of points) and then integrating the ROC curve, but we got lazy and simply called the function `sklearn.metrics.auc`.

```
def buildROC(model, testSet, label, title, plot = True):
    xVals, yVals = [], []
    p = 0.0
    while p <= 1.0:
        truePos, falsePos, trueNeg, falseNeg =\
                                applyModel(model, testSet, label, p)
        xVals.append(1.0 - specificity(trueNeg, falsePos))
        yVals.append(sensitivity(truePos, falseNeg))
        p += 0.01
    auroc = sklearn.metrics.auc(xVals, yVals, True)
    if plot:
        pylab.plot(xVals, yVals)
        pylab.plot([0,1], [0,1,], '--')
        pylab.title(title +  ' (AUROC = '\
                    + str(round(auroc, 3)) + ')')
        pylab.xlabel('1 - Specificity')
        pylab.ylabel('Sensitivity')
    return auroc

buildROC(model, test, 'M', 'ROC for Predicting Gender')
```

Figure 24.16 Construct ROC curve and find AUROC

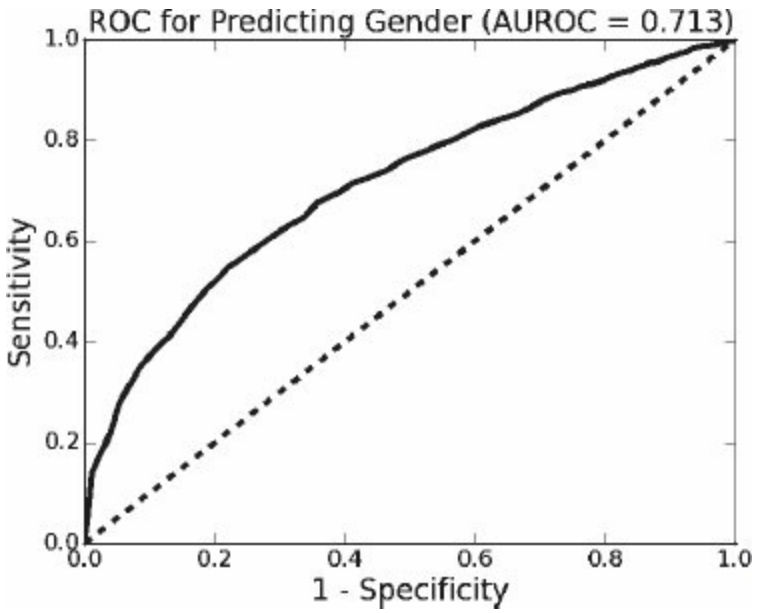

Figure 24.17 ROC curve and AUROC

Finger exercise: Write code to plot the ROC curve and compute the AUROC when the model built in Figure 24.15 is tested on 200 randomly chosen competitors. Use that code to investigate the impact of the number of training examples (try varying it from 10 to 1010 in increments of 50) on the AUROC.

24.5 Surviving the Titanic

On the morning of April 15, 1912, the RMS Titanic hit an iceberg and sank in the North Atlantic. Of the roughly 1,300 passengers on board, 832 perished in the disaster. There were many factors contributing to the disaster, including navigational error, inadequate lifeboats, and the slow response of a nearby ship. Whether or not individual passengers survived had an element of randomness, but was far from completely random. In fact, it is possible to make a reasonably good model for predicting survival using information from the ship's passenger manifest.

In this section, we build a classification model from a data set containing information for 1046 passengers.[180] Each line of the file contains information about a single passenger: cabin class (1st, 2nd, or 3rd), age, gender, whether the passenger survived the disaster, and the passenger's name.

We build the model using logistic regression. We chose to use logistic regression because

- It is the most commonly used classification method.
- By examining the weights produced by logistic regression we can gain some insight into why some passengers were more likely to have survived than others.

Figure 24.18 defines class `Passenger`. The only thing of interest in this code is the encoding of cabin class. Though the file encodes the cabin class as a integer, it is really shorthand for a category. Cabin classes do not behave like numbers, e.g., a first class cabin plus a second class cabin does not equal a third class cabin. We encode cabin class using three binary features (one per possible cabin class). For each passenger, exactly one of these variables is set to `1`, and the other two are set to `0`.

This is an example of an issue that frequently arises in machine learning. Categorical (sometimes called nominal) features are the natural way to describe many things, e.g., the home country of a runner. It's easy to replace these by integers, e.g., we could choose a representation for countries based on their ISO 3166-1 numeric code,[181] e.g., 076 for Brazil, 826 for the United Kingdom, and 862 for Venezuela. The problem with doing this is that the regression will treat these as numerical variables, thus using a nonsensical ordering on the countries in which Venezuela would be closer to the UK than it is to Brazil.

This problem can be avoided by converting categorical variables to binary variables, as we did with cabin class. One potential problem with doing this is that it can lead to very long and sparse feature vectors. For example, if a hospital dispenses 2000 different drugs, we would convert one categorical variable into 2000 binary variables, one for each drug.

```
class Passenger(object):
    features = ('C1', 'C2', 'C3', 'age', 'male gender')
    def __init__(self, pClass, age, gender, survived, name):
        self.name = name
        self.featureVec = [0, 0, 0, age, gender]
        self.featureVec[pClass - 1] = 1
        self.label = survived
        self.cabinClass = pClass
    def distance(self, other):
        return minkowskiDist(self.veatureVec, other.featureVec, 2)
    def getClass(self):
        return self.cabinClass
    def getAge(self):
        return self.featureVec[3]
    def getGender(self):
        return self.featureVec[4]
    def getName(self):
        return self.name
    def getFeatures(self):
        return self.featureVec[:]
    def getLabel(self):
        return self.label
```

Figure 24.18 Class `Passenger`

Figure 24.19 contains code that reads the data from a file and builds a set of examples from the data about the Titanic.

```
def testModels(examples, numTrials, printStats, printWeights):
    survived = 1 #value of label indicating survived
    stats, weights = [], [[], [], [], [], []]
    for i in range(numTrials):
        training, testSet = divide80_20(examples)
        featureVecs, labels = [], []
        for e in training:
            featureVecs.append(e.getFeatures())
            labels.append(e.getLabel())
        featureVecs = pylab.array(featureVecs)
        labels = pylab.array(labels)
        model =\
          sklearn.linear_model.LogisticRegression().fit(featureVecs,
                                                        labels)
        for i in range(len(Passenger.features)):
            weights[i].append(model.coef_[0][i])
        truePos, falsePos, trueNeg, falseNeg =\
                          applyModel(model, testSet, survived, 0.5)
        auroc = buildROC(model, testSet, survived, None, False)
        tmp = getStats(truePos, falsePos, trueNeg, falseNeg, False)
        stats.append(tmp + (auroc,))
    print('Averages for', numTrials, 'trials')
    if printWeights:
        for feature in range(len(weights)):
            featureMean = sum(weights[feature])/numTrials
            featureStd = stdDev(weights[feature])
            print(' Mean weight of', Passenger.features[feature],
              '=', str(round(featureMean, 3)) + ',',
              '95% confidence interval =', round(1.96*featureStd, 3))
    if printStats:
        summarizeStats(stats)
```

Figure 24.19 Read Titanic data and build list of example

Now that we have the data, we can build a logistic regression model using the same code we used to build a model of the Boston Marathon data. However, because the data set has a relatively small number of examples, we need to be concerned about using the evaluation method we employed earlier. It is entirely possible to get an unrepresentative 80-20 split of the data, and then generate misleading results.

To ameliorate the risk, we create many different 80-20 splits (each split is created using the `divide80_20` function defined in Figure 24.5), build and evaluate a classifier for each, and then report mean values and 95% confidence intervals, using the code in Figure 24.20 and Figure 24.21.

```
def testModels(examples, numTrials, printStats, printWeights):
    stats, weights = [], [[], [], [], [], []]
    for i in range(numTrials):
        training, testSet = divide80_20(examples)
        xVals, yVals = [], []
        for e in training:
            xVals.append(e.getFeatures())
            yVals.append(e.getLabel())
        xVals = pylab.array(xVals)
        yVals = pylab.array(yVals)
        model = sklearn.linear_model.LogisticRegression().fit(xVals,
                                                               yVals)
        for i in range(len(Passenger.features)):
            weights[i].append(model.coef_[0][i])
        truePos, falsePos, trueNeg, falseNeg =\
                         applyModel(model, testSet, 1, 0.5)
        auroc = buildROC(model, testSet, 1, None, False)
        tmp = getStats(truePos, falsePos, trueNeg, falseNeg, False)
        stats.append(tmp + (auroc,))
    print('Averages for', numTrials, 'trials')
    if printWeights:
        for feature in range(len(weights)):
            featureMean = sum(weights[feature])/numTrials
            featureStd = stdDev(weights[feature])
            print(' Mean weight of', Passenger.features[feature],
              '=', str(round(featureMean, 3)) + ',',
              '95% confidence interval =', round(1.96*featureStd, 3))
    if printStats:
        summarizeStats(stats)
```

Figure 24.20 Test models for Titanic Survival

```
def summarizeStats(stats):
    """assumes stats a list of 5 floats: accuracy, sensitivity,
       specificity, pos. pred. val, ROC"""
    def printStat(X, name):
        mean = round(sum(X)/len(X), 3)
        std = stdDev(X)
        print(' Mean', name, '=', str(mean) + ',',
              '95% confidence interval =', round(1.96*std, 3))
    accs, sens, specs, ppvs, aurocs = [], [], [], [], []
    for stat in stats:
        accs.append(stat[0])
        sens.append(stat[1])
        specs.append(stat[2])
        ppvs.append(stat[3])
        aurocs.append(stat[4])
    printStat(accs, 'accuracy')
    printStat(sens, 'sensitivity')
    printStat(accs, 'specificity')
    printStat(sens, 'pos. pred. val.')
    printStat(aurocs, 'AUROC')
```

Figure 24.21 Print statistics about classifiers

The call `testModels(examples, 100, True, False)` printed

```
Averages for 100 trials
 Mean accuracy = 0.783, 95% confidence interval = 0.046
 Mean sensitivity = 0.699, 95% confidence interval = 0.099
 Mean specificity = 0.783, 95% confidence interval = 0.046
 Mean pos. pred. val. = 0.699, 95% confidence interval = 0.099
 Mean AUROC = 0.839, 95% confidence interval = 0.051
```

It appears that this small set of features is sufficient to do a reasonably good job of predicting survival. To see why, let's take a look at the weights of the various features. We can do that with the call

```
testModels(examples, 100, False, True)
```

which printed

```
Averages for 100 trials
 Mean weight of C1 = 1.648, 95% confidence interval = 0.156
 Mean weight of C2 = 0.449, 95% confidence interval = 0.095
 Mean weight of C3 = -0.499, 95% confidence interval = 0.112
 Mean weight of age = -0.031, 95% confidence interval = 0.006
 Mean weight of male gender = -2.367, 95% confidence interval = 0.144
```

When it comes to surviving a shipwreck, it seems useful to be rich,[182] young, and female.

24.6 Wrapping Up

In the last three chapters, we've barely scratched the surface of machine learning.

The same could be said about many of the other topics presented in the second half of this book. We've tried to give you a taste of the kind of thinking involved in using computation to better understand the world—in the hope that you will find ways to pursue the topic on your own. You probably found some topics less interesting than others. But we do hope that you encountered at least a few topics you are looking forward to learning more about.

---

[172] They often use more sophisticated methods than random choice in constructing the sample.

[173] Some of us still believe in planet Pluto.

[174] Though we fit the models to the entire training set, we choose to plot only a small subset of the training points. When we plotted all of them, the result was a blob in which it was hard to see any useful detail.

[175] It's called logistic regression because the optimization problem being solved involves an objective function based on the log of an odds ratio. Such functions are called logit functions, and their inverses are call logistic functions.

[176] This toolkit comes preinstalled with some Python IDE's, e.g., Anaconda. To learn more about this library and find out out how to install it, go to `http://scikit-learn.org`.

[177] This relationship is complcated by the fact that features are often correlated with each other. For example, age and finishing time are positively correlated. When features are correlated, the magnitudes of the weights are not independent of each other.

[178] The slight difference in the absolute values of the weights is attributable to the fact that our sample size is finite.

[179] It is called the "receiver operating characteristic" for historical reasons. It was first developed during World War II as way to evaluate the operating characteristics of devices receiving radar signals.

[180] The data was extracted from a data set constructed by R.J. Dawson, and used in "The 'Unusual

Episode' Data Revisted," *Journal of Statistics Education*, v. 3, n. 3, 1995.

[181] ISO 3166-1 numeric is part of the ISO 3166 standard published by the International Organization for Standardization.

[182] A first class cabin on the Titanic cost the equivalent of about $70,000 in today's U.S. dollars.

PYTHON 3.5 QUICK REFERENCE

Common operations on numerical types

i+j is the sum of i and j.
i-j is i minus j.
i*j is the product of i and j.
i//j is integer division.
i/j is floating point division.
i%j is the remainder when the int i is divided by the int j.
ij** is i raised to the power j.
x += y is equivalent to x = x + y. ***=** and **-=** work the same way.

Comparison and Boolean operators

x == y returns True if x and y are equal.
x != y returns True if x and y are not equal.
<, >, <=, >= have their usual meanings.
a and b is True if both a and b are True, and False otherwise.
a or b is True if at least one of a or b is True, and False otherwise.
not a is True if a is False, and False if a is True.

Common operations on sequence types

seq[i] returns the i^{th} element in the sequence.
len(seq) returns the length of the sequence.
seq1 + seq2 concatenates the two sequences. (Not available for ranges.)
n*seq returns a sequence that repeats seq n times. (Not available for ranges.)
seq[start:end] returns a slice of the sequence.
e in seq tests whether e is contained in the sequence.
e not in seq tests whether e is not contained in the sequence.
for e in seq iterates over the elements of the sequence.

Common string methods

s.count(s1) counts how many times the string s1 occurs in s.
s.find(s1) returns the index of the first occurrence of the substring s1 in s; returns -1 if s1 is not in s.
s.rfind(s1) same as find, but starts from the end of s.
s.index(s1) same as find, but raises an exception if s1 is not in s.
s.rindex(s1) same as index, but starts from the end of s.
s.lower() converts all uppercase letters to lowercase.

s.replace(old, new) replaces all occurrences of string old with string new.
s.rstrip() removes trailing white space.
s.split(d) Splits s using d as a delimiter. Returns a list of substrings of s.

Common list methods

L.append(e) adds the object e to the end of L.
L.count(e) returns the number of times that e occurs in L.
L.insert(i, e) inserts the object e into L at index i.
L.extend(L1) appends the items in list L1 to the end of L.
L.remove(e) deletes the first occurrence of e from L.
L.index(e) returns the index of the first occurrence of e in L. Raises ValueError if e not in L.
L.pop(i) removes and returns the item at index i; i defaults to -1. Raises IndexError if L is empty.
L.sort() has the side effect of sorting the elements of L.
L.reverse() has the side effect of reversing the order of the elements in L.

Common operations on dictionaries

len(d) returns the number of items in d.
d.keys() returns a view of the keys in d.
d.values() returns a view of the values in d.
k in d returns True if key k is in d.
d[k] returns the item in d with key k. Raises KeyError if k is not in d.
d.get(k, v) returns d[k] if k in d, and v otherwise.
d[k] = v associates the value v with the key k. If there is already a value associated with k, that value is replaced.
del d[k] removes element with key k from d. Raises KeyError if k is not in d.
for k in d iterates over the keys in d.

Common input/output mechanisms

input(msg) prints msg and then returns the value entered as a string.
print(s1, …, sn) prints strings s1, …, sn separated by spaces.
open('fileName', 'w') creates a file for writing.
open('fileName', 'r') opens an existing file for reading.
open('fileName', 'a') opens an existing file for appending.
fileHandle.read() returns a string containing contents of the file.
fileHandle.readline() returns the next line in the file.
fileHandle.readlines() returns a list containing lines of the file.
fileHandle.write(s) write the string s to the end of the file.
fileHandle.writelines(L) writes each element of L to the file as a separate line.
fileHandle.close() closes the file.

INDEX