19 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 has, alas, not doubled every three years. One approach that the world is using to attempt to exploit what has come to be known as "big data" is **statistical machine learning**.

Machine learning is hard to define. One of the earliest definitions was proposed by the American electrical engineer and computer scientist Arthur Samuel,[126] who defined it as a "Field of study that gives computers the ability to learn without being explicitly programmed." Of course, in some sense, every useful program learns something. For example, an implementation of Newton's method learns the roots of a polynomial.

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 to conclude that *all* politicians are likely to make decisions intended to enhance their chances of staying in office.

When computer scientists speak about machine learning, they most often mean the field of writing programs that automatically learn to make useful inferences from implicit patterns in data. For example, linear regression (see Chapter 15) 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.

In general, machine learning involves observing a set of examples that represent incomplete information about some statistical phenomenon, and then attempting to infer something about the process that generated those examples. The examples are frequently called **training data**.

---

[126] Samuel is probably best known as the author of 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."

Suppose, for example, you were given the following two sets of people:

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

Now, suppose that you were provided with the following partial descriptions of each of them:

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

Based on this incomplete information about these historical figures, you might infer that the process that assigned these examples to the set labeled A or the set labeled B involved separating tall presidents from shorter ones.

The incomplete information is typically called a **feature vector**. Each element of the vector describes some aspect (i.e., feature) of the example.

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/label pairs.[127] The goal is to derive from these examples a rule that predicts the label associated with a previously unseen feature vector. For example, given the sets `A` and `B`, 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

```
Thomas Jefferson: American, President, 189 cm.
```

it would then 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. The best algorithms are quite sophisticated, and understanding them requires a level of mathematical sophistication well beyond that assumed for this book. Consequently, we will not cover them here.

---

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

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.

The most popular unsupervised learning techniques are designed to find clusters of similar feature vectors. Geneticists, for example, use clustering to find groups of related genes. Many popular clustering methods are surprisingly simple. We will present the most widely used algorithm later in this chapter. First, however, we want to say a few words about feature extraction.

19.1 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.[128] 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 gender 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 extraction 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 introduces two kinds of problems:

1. Irrelevant features can lead to a bad model. The danger of this is particularly high when the dimensionality of the data (i.e., the number of different features) is large relative to the number of samples.

2. Irrelevant features can greatly slow the learning process. Machine learning algorithms are often computationally intensive, and complexity grows with both the number of examples and the number of features.

The goal of feature extraction is to reduce the vast amount of information that might be available in examples 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 likes to drink wine. Some attributes, e.g., age and the nation in which they live, are likely to be relevant. Other attributes, e.g., whether they are left-handed, are less likely to be relevant.

Feature extraction is difficult. In the context of supervised learning, one can try to select those features that are correlated with the labels of the examples. In

---

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

unsupervised learning, the problem is 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.

Consider Figure 19.1, 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 19.1 Name, features and labels for assorted animals

A supervised machine learning algorithm (or a human) given only the information about cobras 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 is 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,[129] 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 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.

19.2 Distance Metrics

In Figure 19.1 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, e.g., to ask, is a boa constrictor more similar to a rattlesnake or to a dart frog?[130]

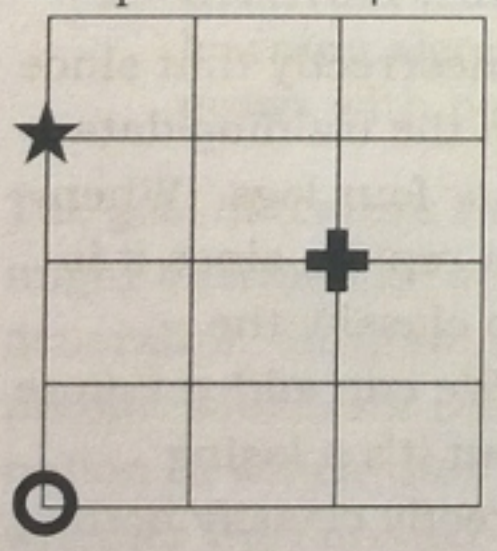
The first step in doing this kind of comparison is converting the features for each animal into a sequence of numbers. If we say `True = 1` and `False = 0`, we get the following feature vectors:

```
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**:

$$distance(V1, V2, p) = (\sum_{i=1}^{len} abs(V1_i - V2_i)^p)^{1/p}$$

The parameter p defines the kinds of paths that can be followed in traversing the distance between the vectors $V1$ and $V2$. This can be mostly easily visualized if the vectors are of length two, and represent Cartesian coordinates. Consider the picture on the left. Is the circle in the bottom left corner closer to the cross or 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



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

[130] 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.

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 one has to stay on the streets to get from one place to another. In that case, the star remains 3 units from the circle, but the cross is now 4 units away. These distances are called **Manhattan distances**,[131] and correspond to using the Minkowski distance with p = 1.

Figure 19.2 contains an implementation of the Minkowski distance.

```
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**(1.0/p)
```

Figure 19.2 Minkowski distance

Figure 19.3 contains class `Animal`. It defines the distance between two animals as the Euclidean distance between the feature vectors associated with the animals.

```
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 19.3 Class `Animal`

Figure 19.4 contains a function that compares a list of animals to each other, and produces a table showing the pairwise distances.

---

[131] Manhattan Island is the most densely populated borough of New York City. On most of the island, the streets are laid out in a 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 (say the Museum of Modern Art at 53rd Street and 6th Avenue) to another (say the American Folk Art Museum at 66th Street and 9th, also called Columbus Avenue). Driving in Manhattan is a totally different story.

```
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.axis('off') #Don't display x and y-axes
    pylab.savefig('distances')
```

Figure 19.4 Build table of distances between pairs of animals

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).

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 a figure containing the table

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

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 does seem to be a bit closer to the rattlesnake than to the boa.

Now, let's add to the bottom of the above code the lines

```
alligator = Animal('alligator', [1,1,0,1,4])
animals.append(alligator)
compareAnimals(animals, 3)
```

It produces the table

	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	–

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	–

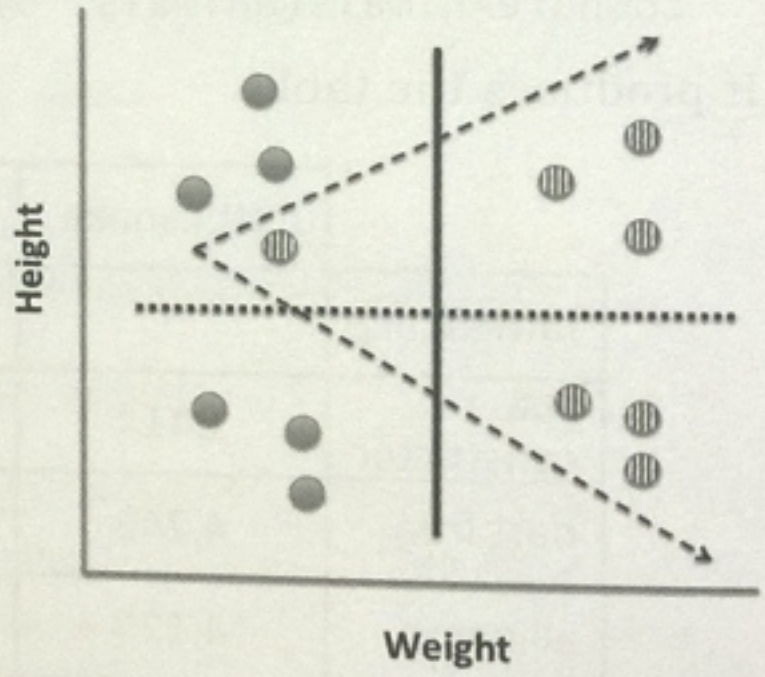
This looks a lot more plausible.

Of course, it is not always convenient to use only binary features. In Section 19.7 we will present a more general approach to dealing with differences in scale among features.

19.3 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 on the right, which shows the height, weight, and whether or not they are wearing a striped shirt for 13 people.



If we want to cluster people by height, there are two obvious clusters—delimited by the dotted horizontal line. If we want to cluster people by weight there are two different obvious clusters—delimited by the solid vertical line. If we want to cluster people based on their shirt, there is yet a third clustering—delimited by the angled dotted arrows. Notice, by the way, that this last division is not linear, i.e., we cannot separate the people wearing striped shirts from the others 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 clusters, i.e., minimizes the dissimilarity of the examples within a cluster.

As we will see later, the exact definition of the objective function can greatly influence the outcome.

A good measure of how close the examples within a single cluster, c, are to each other is variance. To compute the variance of the examples within a cluster, we

first compute the mean of the feature vectors of all the examples in the cluster. If `V` is a list of feature vectors each of which is an array of numbers, the mean (more precisely the Euclidean mean) is the value of the expression `sum(V)/float(len(V))`. Given the mean and a metric for computing the distance between feature vectors, the **variance** of a cluster is

$$variance(c) = \sqrt[2]{\sum_{e \in c} distance(mean(c), e)^2}$$

Notice that the variance 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 variance of each by the size of the cluster.

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

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

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

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 distance between clusters or require that the maximum number of clusters is *k*.

In general, solving this optimization problem is computationally prohibitive for most interesting problems. Consequently, people rely on greedy algorithms that provide approximate solutions. Later in this chapter, 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).

19.4 Types Example and 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 numbers
        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 19.5 Class `Example`

Class `Cluster`, Figure 19.6, is slightly more complex. Think of a cluster as a set of examples. The two interesting methods in `Cluster` are `computeCentroid` and `variance`. 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 `variance` provides a measure of the coherence of the cluster.

```
class Cluster(object):

    def __init__(self, examples, exampleType):
        """Assumes examples is a list of example of type exampleType"""
        self.examples = examples
        self.exampleType = exampleType
        self.centroid = self.computeCentroid()

    def update(self, examples):
        """Replace the examples in the cluster by new examples
           Return how much the centroid has changed"""
        oldCentroid = self.centroid
        self.examples = examples
        if len(examples) > 0:
            self.centroid = self.computeCentroid()
            return oldCentroid.distance(self.centroid)
        else:
            return 0.0

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

    def size(self):
        return len(self.examples)

    def getCentroid(self):
        return self.centroid

    def computeCentroid(self):
        dim = self.examples[0].dimensionality()
        totVals = pylab.array([0.0]*dim)
        for e in self.examples:
            totVals += e.getFeatures()
        centroid = self.exampleType('centroid',
                                    totVals/float(len(self.examples)))
        return centroid

    def variance(self):
        totDist = 0.0
        for e in self.examples:
            totDist += (e.distance(self.centroid))**2
        return totDist**0.5

    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]
```

Figure 19.6 Class `Cluster`

19.5 K-means Clustering

K-means clustering is probably the most widely used clustering method.[132] Its goal is to partition a set of examples into k clusters such that

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

Unfortunately, finding an optimal solution to this problem on a large dataset is computationally intractable. Fortunately, there is an efficient greedy algorithm[133] that can be used to find a useful approximation. It is described by the pseudocode

```
randomly choose k examples as initial centroids
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.[134] 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 it is nondeterministic—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 19.7 contains a straightforward translation of the pseudocode describing k-means into Python. It uses `random.sample(examples, k)` to get the initial centroids. This invocation returns a list of k randomly chosen distinct elements from the list `examples`.

---

[132] 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 coverd in this book, are hierarchical clustering and EM-clustering.

[133] 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.

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

```
def kmeans(examples, exampleType, k, verbose):
    """Assumes examples is a list of examples of type exampleType,
         k is a positive int, verbose is a Boolean
       Returns a list containing k clusters. If verbose is
         True it prints result of each iteration of k-means"""
    #Get k randomly chosen initial centroids
    initialCentroids = random.sample(examples, k)

    #Create a singleton cluster for each centroid
    clusters = []
    for e in initialCentroids:
        clusters.append(Cluster([e], exampleType))

    #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 the appropriate cluster
            newClusters[index].append(e)

        #Upate each cluster; check if a centroid has changed
        converged = True
        for i in range(len(clusters)):
            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 19.7 K-means clustering

Figure 19.8 contains a function, `trykmeans`, that calls `kmeans` multiple times and selects the result with the lowest dissimilarity.

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

def trykmeans(examples, exampleType, numClusters, numTrials,
              verbose = False):
    """Calls kmeans numTrials times and returns the result with the
         lowest dissimilarity"""
    best = kmeans(examples, exampleType, numClusters, verbose)
    minDissimilarity = dissimilarity(best)
    for trial in range(1, numTrials):
        clusters = kmeans(examples, exampleType, numClusters, verbose)
        currDissimilarity = dissimilarity(clusters)
        if currDissimilarity < minDissimilarity:
            best = clusters
            minDissimilarity = currDissimilarity
    return best
```

Figure 19.8 Finding the best k-means clustering

19.6 A Contrived Example

Figure 19.9 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 another PyLab plotting feature that we have not yet seen: the function `annotate` is used 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`.

```
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):
    random.seed(0)
    xMean = 3
    xSD = 1
    yMean = 5
    ySD = 1
    n = 10
    d1Samples = genDistribution(xMean, xSD, yMean, ySD, n, '1.')
    plotSamples(d1Samples, 'b^')
    d2Samples = genDistribution(xMean+3, xSD, yMean+1, ySD, n, '2.')
    plotSamples(d2Samples, 'ro')
    clusters = trykmeans(d1Samples + d2Samples, Example, k,
                         numTrials, verbose)
    print 'Final result'
    for c in clusters:
        print '', c
```

Figure 19.9 A test of k-means

When executed, the call `contrivedTest(1, 2, True)` produced the plot in Figure 19.10.

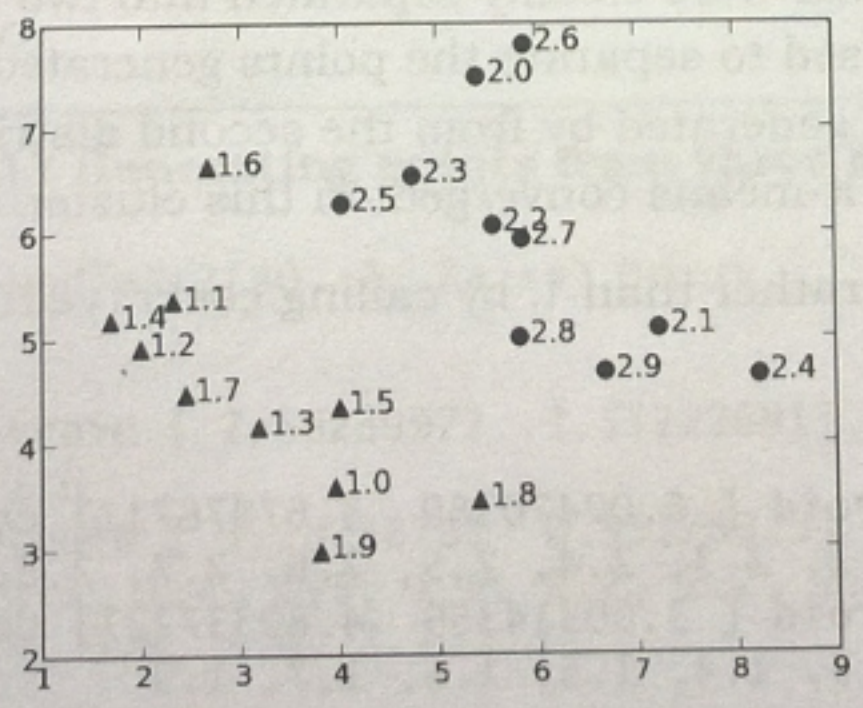


Figure 19.10 Examples from two distributions

and printed

```
Iteration 1
 Cluster with centroid [ 4.57800047  5.35921276] contains:
  1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 2.0, 2.1, 2.2, 2.3,
2.4, 2.5, 2.6, 2.7, 2.8, 2.9
 Cluster with centroid [ 3.79646584  2.99635148] contains:
  1.9

Iteration 2
 Cluster with centroid [ 4.80105783  5.73986393] contains:
  1.1, 1.2, 1.4, 1.5, 1.6, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7,
2.8, 2.9
 Cluster with centroid [ 3.75252146  3.74468698] contains:
  1.0, 1.3, 1.7, 1.8, 1.9

Iteration 3
 Cluster with centroid [ 5.6388835   6.02296994] contains:
  1.6, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9
 Cluster with centroid [ 3.19452848  4.28541384] contains:
  1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.7, 1.8, 1.9

Iteration 4
 Cluster with centroid [ 5.93613865  5.96069975] contains:
  2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9
 Cluster with centroid [ 3.14170883  4.52143963] contains:
  1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9

Iteration 5
 Cluster with centroid [ 5.93613865  5.96069975] contains:
  2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9
 Cluster with centroid [ 3.14170883  4.52143963] contains:
  1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9

Final result
 Cluster with centroid [ 5.93613865  5.96069975] contains:
  2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9
 Cluster with centroid [ 3.14170883  4.52143963] contains:
  1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9
```

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 fifth iteration, however, the centroids had moved to places such that the points from the two distributions were cleanly separated into two clusters. Given that a straight line can be used to separate the points generated from the first distribution from those generated by from the second distribution, it is not terribly surprising that k-means converged on this clustering.

When we tried 40 trials rather than 1, by calling `contrivedTest(40, 2, False)`, it printed

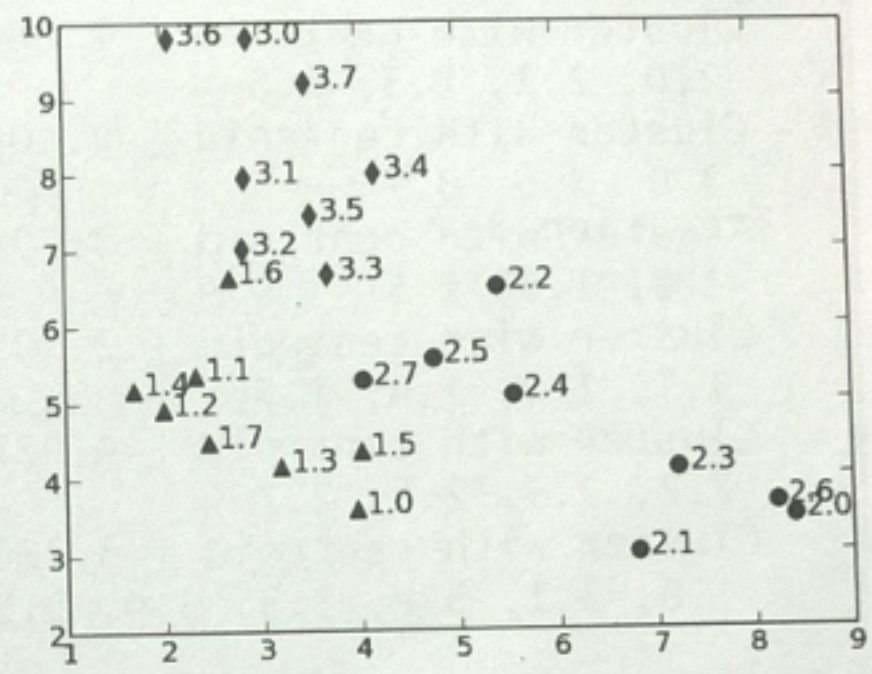
```
Final result
 Cluster with centroid [ 6.07470389  5.67876712] contains:
  1.8, 2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9
 Cluster with centroid [ 3.00314359  4.80337227] contains:
  1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.9
```

This indicates that the solution found using 1 trial, despite perfectly separating the examples by the distribution from which they were chosen, was not as good

(with respect to minimizing the objective function) as one of the solutions found using 40 trials.

Finger exercise: Draw lines on Figure 19.10 to show the separations found by our two attempts to cluster the points. Do you agree that the solution found using 40 trials is better than the one found using 1 trial?

One of the key issues in using k-means clustering is choosing k. Consider the points in the plot on the right, which were generated using `contrivedTest2`, Figure 19.11. This function generates and clusters points from three overlapping Gaussian distributions.




```
def contrivedTest2(numTrials, k, verbose):
    random.seed(0)
    xMean = 3
    xSD = 1
    yMean = 5
    ySD = 1
    n = 8
    d1Samples = genDistribution(xMean,xSD, yMean, ySD, n, '1.')
    plotSamples(d1Samples, 'b^')
    d2Samples = genDistribution(xMean+3,xSD,yMean, ySD, n, '2.')
    plotSamples(d2Samples, 'ro')
    d3Samples = genDistribution(xMean, xSD, yMean+3, ySD, n, '3.')
    plotSamples(d3Samples, 'gd')
    clusters = trykmeans(d1Samples + d2Samples + d3Samples,
                         Example, k, numTrials, verbose)
    print 'Final result'
    for c in clusters:
        print '', c
```

Figure 19.11 Generating points from three distributions

The invocation `contrivedTest2(40, 2, False)` prints

```
Final result
 Cluster with centroid [ 7.66239972  3.55222681] contains:
  2.0, 2.1, 2.3, 2.6
 Cluster with centroid [ 3.36736761  6.35376823] contains:
  1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 2.2, 2.4, 2.5, 2.7, 3.0,
3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7
```

The invocation contrivedTest2(40, 3, False) prints

```
Final result
 Cluster with centroid [ 7.66239972  3.55222681] contains:
  2.0, 2.1, 2.3, 2.6
 Cluster with centroid [ 3.10687385  8.46084886] contains:
  3.0, 3.1, 3.2, 3.4, 3.5, 3.6, 3.7
 Cluster with centroid [ 3.50763348  5.21918636] contains:
  1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 2.2, 2.4, 2.5, 2.7, 3.3
```

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

```
Final result
 Cluster with centroid [ 7.66239972  3.55222681] contains:
  2.0, 2.1, 2.3, 2.6
 Cluster with centroid [ 2.80974427  9.60386549] contains:
  3.0, 3.6, 3.7
 Cluster with centroid [ 3.70472053  4.04178035] contains:
  1.0, 1.3, 1.5
 Cluster with centroid [ 2.10900238  4.99452866] contains:
  1.1, 1.2, 1.4, 1.7
 Cluster with centroid [ 4.92742554  5.60609442] contains:
  2.2, 2.4, 2.5, 2.7
 Cluster with centroid [ 3.27637435  7.28932247] contains:
  1.6, 3.1, 3.2, 3.3, 3.4, 3.5
```

The last clustering is the tightest fit, i.e., the clustering has the lowest dissimilarity. Does this mean that it is the "best" fit? Recall that when we looked at linear regression in Section 15.1.1, 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 zero!) If we have some 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.

19.7 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 are called omnivores.

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

The table on the right shows the contents of a file listing some species of mammals, their dental formulas (the first 8 numbers), their average adult weight in pounds,[135] 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
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
Elk,0,1,3,3,3,1,3,3,500,0
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
Porcupine,1,0,1,3,1,0,1,3,3,0
Pig,3,1,4,3,3,1,4,3,50,2
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
Woodchuck,1,0,2,3,1,0,1,3,4,2
Wolf,3,1,4,2,3,1,4,3,27,1
```

Figure 19.12 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. The value of the expression `featureVals[i][j]` is the i^{th} feature of the j^{th} mammal.

---

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

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. We chose not to do that in anticipation of an enhancement to readMammalData that we make later in this section.)

```
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:
        dataLine = string.split(line[:-1], ',') #remove newline; then split
        speciesNames.append(dataLine[0])
        classLabel = float(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
```

Figure 19.12 Read and process file

The function testTeeth in Figure 19.13 uses trykmeans to cluster the examples built by the other function, buildMammalExamples, in Figure 19.13. It then reports the number of herbivores, carnivores, and omnivores in each cluster.

```
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

def testTeeth(numClusters, numTrials):
    features, labels, species = readMammalData('dentalFormulas.txt')
    examples = buildMammalExamples(features, labels, species)
    bestClustering =\
                trykmeans(examples, Example, 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 19.13 Clustering animals

When we executed the code `testTeeth(3, 20)` it printed

```
Cow, Elk, Moose, Sea lion
3 herbivores, 1 carnivores, 0 omnivores

Badger, Cougar, Dog, Fox, Guinea pig, Jaguar, Kangaroo, Mink, Mole,
Mouse, Porcupine, Pig, Rabbit, Raccoon, Rat, Red bat, Skunk, Squirrel,
Woodchuck, Wolf
4 herbivores, 9 carnivores, 7 omnivores

Bear, Deer, Fur seal, Grey seal, Human, Lion
1 herbivores, 3 carnivores, 2 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 19.2 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 way to binarize weight 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, as done by the function scaleFeatures in Figure 19.14.

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

Figure 19.14 Scaling attributes

To see the effect of scaleFeatures, let's look at the code below.

```
v1, v2 = [], []
for i in range(1000):
    v1.append(random.gauss(100, 5))
    v2.append(random.gauss(50, 10))
v1 = scaleFeatures(v1)
v2 = scaleFeatures(v2)
print 'v1 mean =', round(sum(v1)/len(v1), 4),\
      'v1 standard deviation', round(stdDev(v1), 4)
print 'v2 mean =', round(sum(v2)/len(v2), 4),\
      'v1 standard deviation', round(stdDev(v2), 4)
```

The code generates two normal distributions with different means (100 and 50) and different standard deviations (5 and 10). It then scales each and prints the means and standard deviations of the results. When run, it prints

```
v1 mean = -0.0 v1 standard deviation 1.0
v2 mean = 0.0 v1 standard deviation 1.0
```

[136]

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[137]. 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.

Figure 19.15 contains a version of readMammalData that allows scaling of features. The new version of the function testTeeth in the same figure shows the result of clustering with and without scaling.

---

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

[137] We say "close," because floating point numbers are only an approximation to the reals and the result will not always be exactly 0.

```
def readMammalData(fName, scale):
    """Assumes scale is a Boolean.  If True, features are scaled"""

    #start of code is same as in previous version

    #Use featureVals to build list containing the feature vectors
    #for each mammal scale features, if needed
    if scale:
        for i in range(numFeatures):
            featureVals[i] = scaleFeatures(featureVals[i])

    #remainder of code is the same as in previous version

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

    #remainder of code is the same as in the previous version
```

Figure 19.15 Code that allows scaling of features

When we execute the code

```
print 'Cluster without scaling'
testTeeth(3, 20, False)
print '\nCluster with scaling'
testTeeth(3, 20, True)
```

it prints

```
Cluster without scaling

Cow, Elk, Moose, Sea lion
3 herbivores, 1 carnivores, 0 omnivores

Badger, Cougar, Dog, Fox, Guinea pig, Jaguar, Kangaroo, Mink, Mole,
Mouse, Porcupine, Pig, Rabbit, Raccoon, Rat, Red bat, Skunk, Squirrel,
Woodchuck, Wolf
4 herbivores, 9 carnivores, 7 omnivores

Bear, Deer, Fur seal, Grey seal, Human, Lion
1 herbivores, 3 carnivores, 2 omnivores

Cluster with scaling

Cow, Deer, Elk, Moose
4 herbivores, 0 carnivores, 0 omnivores

Guinea pig, Kangaroo, Mouse, Porcupine, Rabbit, Rat, Squirrel,
Woodchuck
4 herbivores, 0 carnivores, 4 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
```

```
def readMammalData(fName, scale):
    """Assumes scale is a Boolean.  If True, features are scaled"""

    #start of code is same as in previous version

    #Use featureVals to build list containing the feature vectors
    #for each mammal scale features, if needed
    if scale:
        for i in range(numFeatures):
            featureVals[i] = scaleFeatures(featureVals[i])

    #remainder of code is the same as in previous version

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

    #remainder of code is the same as in the previous version
```

Figure 19.15 Code that allows scaling of features

When we execute the code

```
print 'Cluster without scaling'
testTeeth(3, 20, False)
print '\nCluster with scaling'
testTeeth(3, 20, True)
```

it prints

```
Cluster without scaling

Cow, Elk, Moose, Sea lion
3 herbivores, 1 carnivores, 0 omnivores

Badger, Cougar, Dog, Fox, Guinea pig, Jaguar, Kangaroo, Mink, Mole,
Mouse, Porcupine, Pig, Rabbit, Raccoon, Rat, Red bat, Skunk, Squirrel,
Woodchuck, Wolf
4 herbivores, 9 carnivores, 7 omnivores

Bear, Deer, Fur seal, Grey seal, Human, Lion
1 herbivores, 3 carnivores, 2 omnivores

Cluster with scaling

Cow, Deer, Elk, Moose
4 herbivores, 0 carnivores, 0 omnivores

Guinea pig, Kangaroo, Mouse, Porcupine, Rabbit, Rat, Squirrel,
Woodchuck
4 herbivores, 0 carnivores, 4 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
```

The clustering with scaling does not perfectly partition the animals based upon their eating habits, but it is certainly correlated with what the animals 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.[138]

19.8 Wrapping Up

In this chapter, we've barely scratched the surface of machine learning. We've tried to give you a taste of the kind of thinking involved in using machine learning—in the hope that you will find ways to pursue the topic on your own.

The same could be said about many of the other topics presented in this book. We've covered a lot more ground than is typical of introductory computer science courses. 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.

[138] Eye position might be a useful feature, since both omnivores and carnivores typically have eyes in the front of their head, whereas the eyes of herbivores are typically located more towards the side. Among the mammals, only mothers of humans have eyes in the back of their head.