Chapter 10: Preparing the Data Set

Overview

In this chapter, the focus of attention is on the data set itself. Using the term "data set" places emphasis on the interactions *between* the variables, whereas the term "data" has implied focusing on individual variables and their instance values. In all of the data preparation techniques discussed so far, care has been taken to expose the information content of the individual variables to a modeling tool. The issue now is how to make the information content of the data set itself most accessible. Here we cover using sparsely populated variables, handling problems associated with excessive dimensionality, determining an appropriate number of instances, and balancing the sample.

These are all issues that focus on the data set and require restructuring it as a whole, or at least, looking at groups of variables instead of looking at the variables individually.

10.1 Using Sparsely Populated Variables

Why use sparsely populated variables? When originally choosing the variables to be included in the data set, any in which the percentage of missing values is too high are usually discarded. They are discarded as simply not having enough information to be worth retaining. Some forms of analysis traditionally discard variables if 10 or 20% of the values are missing. Very often, when data mining, this discards far too many variables, and the threshold is set far lower. Frequently, the threshold is set to only exclude variables with more than 80 or 90% of missing values, if not more. Occasionally, a miner is constrained to use extremely sparsely populated variables—even using variables with only fractions of 1% of the values present. Sometimes, almost all of the variables in a data set are sparsely populated. When that is the case, it is these sparsely populated variables that carry the only real information available. The miner either uses them or gives up mining the data set.

For instance, one financial brokerage data set contained more than 700 variables. A few were well populated: account balance, account number, margin account balance, and so on. The variables carrying the information to be modeled, almost all of the variables present, were populated at below 10%, more than half below 2%; and a full one-third of all the variables present were populated below 1%—that is, with less than 1% of the values present. These were fields like "trades-in-corn-last-quarter," "open-contracts-oil," "June-open-options-hogs-bellies," "number-stop-loss-per-cycle," and other specialized information. How can such sparsely populated variables be used?

The techniques discussed up to this point in this book will not meet the need. For the brokerage data set just mentioned, for instance, the company wanted to predict portfolio trading proclivity so that the brokers could concentrate on high-value clients. Traditional modeling techniques have extreme difficulty in using such sparse data. Indeed, the brokerage analysts had difficulty in estimating trading proclivity with better than a 0.2 correlation. The full data preparation tool set, of which a demonstration version is on the accompanying CD-ROM, produced models with somewhat better than a 0.4 correlation when very sparsely populated variables were not included. When such variables were included, prepared as usual (see Chapter 8), the correlation increased to just less than 0.5. However, when these variables were identified and prepared as very sparsely populated, the correlation climbed to better than 0.7.

Clearly, there are occasions when the sparsely populated variables carry information that must be used. The problem is that, unless somehow concentrated, the information density is too low for mining tools to make good use of it. The solution is to increase the information density to the point where mining tools can use it.

10.1.1 Increasing Information Density Using Sparsely Populated Variables

When using very sparsely populated variables, missing-value replacement is not useful. Even when the missing values are replaced so they can be used, the dimensionality of state space increases by every sparsely populated variable included. Almost no information is gained in spite of this increase, since the sparsely populated variable simply does not carry much information. (Recall that replacing a missing value adds no information to the data set.) However, sometimes, and for some applications, variables populated with extreme sparsity have to be at least considered for use.

One solution, which has proved to work well when sparsely populated variables have to be used, collapses the sparse variables into fewer composite variables. Each composite variable carries information from several to many sparsely populated variables. If the sparsely populated variables are alpha, they are left in that form. If they are not alpha, categories are created from the numerical information. If there are many discrete numeric values, their number may have to be reduced. One method that works well is to “bin” the values and assign an alpha label to each bin. Collapsing the numeric information needs a situation-specific solution. Some method needs to be devised by the miner, together with a domain expert if necessary, to make sure that the needed information is available to the model.

It may be that several categories can occur simultaneously, so that simply creating a label for each individual variable category is not enough. Labels have to be created for each category combination that occurs, not just the categories themselves.

10.1.2 Binning Sparse Numerical Values

Binning is not a mysterious process. It only involves dividing the range of values into subranges and using subrange labels as substitutes for the actual values. Alpha labels are used to identify the subranges. This idea is very intuitive and widely used in daily life. Coffee temperature, say, may be binned into the categories “scalding,” “too hot,” “hot,” “cool,” and “cold.” These five alpha labels each represent part of the temperature range of coffee. These bins immediately translate into alpha labels. If the coffee temperature is in the range of “hot,” then assign the label “hot.” Figure 10.1 shows how binning works for coffee temperature.

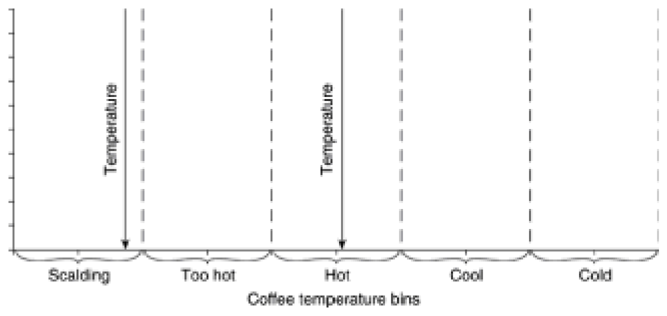


Figure 10.1 Binning coffee temperature to assign alpha labels. The left bar represents coffee assigned the label “Scalding” even though it falls close to the edge of the bin, near the boundary between ”Scalding” and “Too hot.” Likewise, the centrally located bar is assigned the label “Hot.”

This method of assigning labels to numerical values extends to any numerical variable. Domain knowledge facilitates bin boundaries assignment, appropriately locating where meaningful boundaries fall. For coffee temperature, both the number of bins and the bin boundaries have a rationale that can be justified. Where this is not the case, arbitrary boundaries and bin count have to be assigned. When there is no rationale, it is a good idea to assign bin boundaries so that each bin contains approximately equal numbers of labels.

10.1.3 Present-Value Patterns (PVPs)

Chapter 8 discussed missing-value patterns (MVPs). MVPs are created when most of the values are present and just a few are missing. Very sparsely populated variables present almost exactly the reverse situation. Most of the values are missing, and just a few are present. There is one major difference. With MVPs, the values were either missing or present. With PVPs, it is not enough to simply note the fact of the presence of a label; the PVP must also note the indentity of the label. The miner needs to account for this difference. Instead of simply noting, say, “P” for present and “A” for absent, the “P” must be a label of some sort that reflects which label or labels are present in the sparse variables. The miner must map every unique combination of labels present to a unique label and use that second label. This collapses many variables into one. Figure 10.2 shows this schematically, although for illustration the density of values in each variable in the figure is enormously higher than this method would ever be applied to in practice.

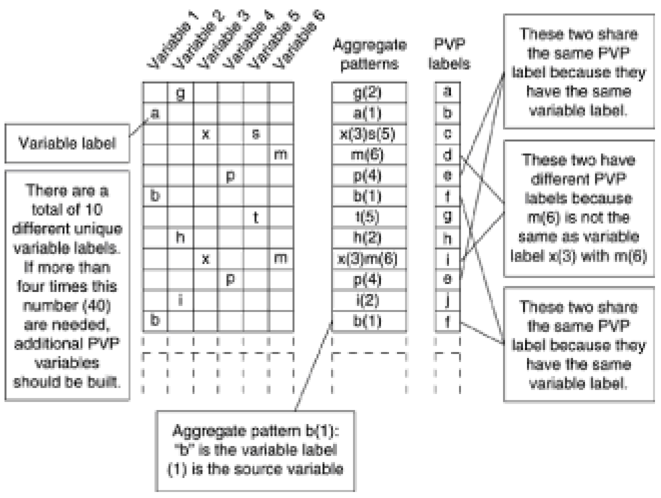

Variable 1
Variable 2
Variable 3
Variable 4
Variable 5
Variable 6
Aggregate patterns
PVP labels
Variable label
There are a total of 10 different unique variable labels. If more than four times this number (40) are needed, additional PVP variables should be built.
g
a
x
s
m
p
b
t
h
x
m
p
i
b
g(2)
a(1)
x(3)s(5)
m(6)
p(4)
b(1)
t(5)
h(2)
x(3)m(6)
p(4)
i(2)
b(1)
a
b
c
d
e
f
g
h
i
e
j
f
These two share the same PVP label because they have the same variable label.
These two have different PVP labels because m(6) is not the same as variable label x(3) with m(6)
These two share the same PVP label because they have the same variable label.
Aggregate pattern b(1): "b" is the variable label (1) is the source variable

Figure 10.2 Sparsely populated variables generate unique patterns for those values that are present. Every PVP has a unique label. The density of values shown in the figure is far above the fractional percent density for which this method of compression is designed.

Note that PVPs are only created for variables that are very sparsely populated—usually at less than 1%. Creating PVPs for more populous variables results in a proliferation of alpha labels that simply explodes! In fact, if there are too many PVPs in any one created variable, subsets of the sparsely populated variables should be collapsed so that the label count does not get too high. What are too many PVPs? As a rule of thumb, if the number of PVPs is more than four times the total number of individual variable labels, use multiple PVP variables. Where multiple PVP variables are needed, select groups of sparsely populated variables so that the PVP label count is minimized.

Building PVP patterns this way does lose some information. Binning itself discards information in the variables for a practical gain in usability. However, using PVPs makes much of the information in very sparsely populated variables available to the mining tool, where it would otherwise be completely discarded. The created PVP variable(s) is numerated exactly as any other alpha variable. Chapter 6 discusses numerating alpha values.

10.2 Problems with High-Dimensionality Data Sets

The dimensionality of a data set is really a count of the number of variables it contains. When discussing state space (Chapter 6), each of the variables was referred to as a "dimension." Very large state spaces—those with large numbers of dimensions—present problems for all mining tools. Why problems? For one reason, no matter how fast or powerful the mining tool, or the computer running the tool, there is always some level of dimensionality that will defeat any effort to build a comprehensive model. Even a massively parallel processor, totally dedicated to the project and running a highly advanced and optimized mining toolset, could not deal with a multiterabyte, 7000+ variable data set required on one mining project.

Another reason that high dimensionality presents difficulties for mining tools is that as the dimensionality increases, the size (multidimensional volume) of state space increases. This requires more data points to fill the space to any particular density. Low-density state spaces leave more "wiggle room" between the data points where the shape of the manifold is undefined. In these spaces there is more probability of overfitting than in more populous state spaces. (Chapter 3 discusses overfitting.) To reduce the risk of overfitting, more instances are needed—lots more instances. Just how many is discussed later in this chapter.

What seems to be another problem with increasing dimensionality, but is actually the same problem in different clothing, is the combinatorial explosion problem. "Combinatorial" here refers to the number of different ways that the values of the variables can be combined. The problem is caused by the number of possible unique system states increasing as the multiple of the individual variable states. For instance, if three variables have two, three, and four possible states each, there are 2 x 3 x 4 = 24 possible system states. When each variable can take tens, hundreds, thousands, or millions of meaningful discrete states, and there are hundreds or thousands of variables, the number of possible discrete, meaningful system states becomes very large—*very* large! And yet, to create a fully representative model, a mining tool ideally needs at least one example of each meaningful system state represented in state space. The number of instances required can very quickly become impractical, and shortly thereafter impossible to assemble.

There seem to be three separate problems here. First, the sheer amount of data defeats the hardware/software mining tools. Second, low density of population in a voluminous state space does not well define the shape of the manifold in the spaces between the data points. Third, the number of possible combinations of values requires an impossibly huge amount of data for a representative sample—more data than can actually be practically assembled. As if these three (apparently) separate problems weren't enough, high dimensionality brings with it other problems too! As an example, if variables are "colinear"—that is, they are so similar in information content as to carry nearly identical information—some tools, particularly those derived from statistical techniques, can have extreme problems dealing with some representations of such variables. The chance that two such variables occur together goes up tremendously as dimensionality increases. There are ways around this particular problem, and around many other problems. But it is much better to avoid them if possible.

What can be done to alleviate the problem? The answer requires somehow reducing the amount of data to be mined. Reducing the number of instances doesn't help since large state spaces need more, not less, data to define the shape of the manifold than small ones. The only other answer requires reducing the number of dimensions. But that seems to mean removing variables, and removing variables means removing information, and removing information is a poor answer since a good model needs all the information it can get. Even if removing variables is absolutely required in order to be able to mine at all, how should the miner select the variables to discard?

10.2.1 Information Representation

The real problem here is very frequently with the data representation, not really with high dimensionality. More properly, the problem is with *information* representation. Information representation is discussed more fully in Chapter 11. All that need be understood for the moment is that the values in the variables carry information. Some variables may duplicate all or part of the information that is also carried by other variables. However, the data set as a whole carries within it some underlying pattern of information distributed among its constituent variables. It is this information, carried in the weft and warp of the variables—the intertwining variability, distribution patterns, and other interrelationships—that the mining tool needs to access.

Where two variables carry identical information, one can be safely removed. After all, if the information carried by each variable is identical, there has to be a correlation of either +1 or –1 between them. It is easy to re-create one variable from the other with perfect fidelity. Note that although the information carried is identical, the form in which it is carried may differ. Consider the two times table. The instance values of the variable "the number to multiply" are different from the corresponding instance values of the variable "the answer." When connected by the relationship "two times table," both variables carry identical information and have a correlation of +1. One variable carries information to perfectly re-create instance values of the other, but the actual content of the variables is not at all similar.

What happens when the information shared between the variables is only partially duplicated? Suppose that several people are measured for height, weight, and girth, creating a data set with these as variables. Suppose also that any one variable's value can be derived from the other two, but not from any other one. There is, of course, a correlation between any two, probably a very strong one in this case, but not a perfect correlation. The height, weight, and girth measurements are all different from each other and they can all be plotted in a three-dimensional state space. But is a three-dimensional state space needed to capture the information? Since any two variables serve to completely specify the value of the third, one of the variables isn't actually needed. In fact, it only requires a two-dimensional state space to carry all of the information present. Regardless of which two variables are retained in the state space, a transformation function, suitably chosen, will perfectly give the value of the third. In this case, the information can be "embedded" into a two-dimensional state space without any loss of either predictive or inferential power. Three dimensions are needed to capture the variables' values—but only two dimensions to capture the information.

To take this example a little further, it is very unlikely that two variables will perfectly predict the third. Noise (perhaps as measurement errors and slightly different muscle/fat/bone ratios, etc.) will prevent any variable from being perfectly correlated with the other two. The noise adds some unique information to each variable—but is it wanted? Usually a miner wants to discard noise and is interested in the underlying relationship, not the noise relationship. The underlying relationship can still be embedded in two dimensions. The noise, in this example, will be small compared to the relationship but needs three dimensions. In multidimensional scaling (MDS) terms (see Chapter 6), projecting the relationship into two dimensions causes some, but only a little, stress. For this example, the stress is caused by noise, not by the underlying information.

Using MDS to collapse a large data set can be highly computationally intensive. In Chapter 6, MDS was used in the numeration of alpha labels. When using MDS to reduce data set dimensionality, instead of alpha label dimensionality, discrete system states have to be discovered and mapped into phase space. There may be a very large number of these, creating an enormous “shape.” Projecting and manipulating this shape is difficult and time-consuming. It can be a viable option. Collapsing a large data set is always a computationally intensive problem. MDS may be no slower or more difficult than any other option.

But MDS is an “all-or-nothing” approach in that only at the end is there any indication whether the technique will collapse the dimensionality, and by how much. From a practical standpoint, it is helpful to have an incremental system that can give some idea of what compression might achieve as it goes along. MDS requires the miner to choose the number of variables into which to attempt compression. (Even if the number is chosen automatically as in the demonstration software.) When compressing the whole data set, a preferable method allows the miner to specify a required level of confidence that the information content of the original data set has been retained, instead of specifying the final number of compressed variables. Let the required confidence level determine the number of variables instead of guessing how many might work.

10.2.2 Representing High-Dimensionality Data in Fewer Dimensions

There are dimensionality-reducing methods that work well for linear between-variable relationships. Methods such as principal components analysis and factor analysis are well-known ways of compressing information from many variables into fewer variables. (Statisticians typically refer to these as data reduction methods.)

Principal components analysis is a technique used for concentrating variability in a data set. Each of the dimensions in a data set possesses a variability. (Variability is discussed in many places; see, for example, Chapter 5.) Variability can be normalized, so that each dimension has a variability of 1. Variability can also be redistributed. A *component* is an artificially constructed variable that is fitted to all of the original variables in a data set in such a way that it extracts the highest possible amount of variability.

The total amount of variability in a specific data set is a fixed quantity. However, although each original variable contributes the same amount of variability as any other original variable, redistributing it concentrates data set variability in some components, reducing it in others. With, for example, 10 dimensions, the variability of the data set is 10. The first *component,* however, might have a variability not of 1—as each of the original variables has—but perhaps of 5. The second component, constructed to carry as much of the remaining variability as possible, might have a variability of 4. In principal components analysis, there are always in total as many components as there are original variables, but the remaining eight variables in this example now have a variability of 1 to share between them. It works out this way: there is a total amount of variability of 10/10 in the 10 original variables. The first two components carry 5/10 + 4/10 = 9/10, or 90% of the variability of the data set. The remaining eight components therefore have only 10% of the variability to carry between them.

Inasmuch as variability is a measure of the information content of a variable (discussed in Chapter 11), in this example, 90% of the information content has been squeezed into only two of the specially constructed variables called components. Capturing the full variability of the data set still requires 10 components, no change over having to use the 10 original variables. But it is highly likely that the later components carry noise, which is well ignored. Even if noise does not exist in the remaining components, the benefit gained in collapsing the number of variables to be modeled by 80% may well be worth the loss of information.

The problem for the miner with principal component methods is that they only work well for *linear* relationships. Such methods, unfortunately, actually damage or destroy nonlinear relationships—catastrophic and disastrous for the mining process! Some form of nonlinear principal components analysis seems an ideal solution. Such techniques are now being developed, but are extremely computationally intensive—so intensive, in fact, that they themselves become intractable at quite moderate dimensionalities. Although promising for the future, such techniques are not yet of help when collapsing information in intractably large dimensionality data sets.

Removing variables is a solution to dimensionality reduction. Sometimes this is required since no other method will suffice. For instance, in the data set of 7000+ variables mentioned before, removing variables was the only option. Such dimensionality mandates a reduction in the number of dimensions before it is practical to either mine or compress it with any technique available today. But when discarding variables is required, selecting the variables to discard needs a rationale that selects the least important variables. These are the variables least needed by the model. But how are the least needed variables to be discovered?

10.3 Introducing the Neural Network

One problem, then, is how to squash the information in a data set into fewer variables without destroying any nonlinear relationships. Additionally, if squashing the data set is impossible, how can the miner determine which are the least contributing variables so that they can be removed? There is, in fact, a tool in the data miner's toolkit that serves both dimensionality reduction purposes. It is a very powerful tool that is normally used as a modeling tool. Although data preparation uses the full range of its power, it is applied to totally different objectives than when mining. It is introduced here in general terms before examining the modifications needed for dimensionality reduction. The tool is the standard, back-propagation, artificial neural network (BP-ANN).

The idea underlying a BP-ANN is very simple. The BP-ANN has to learn to make predictions. The learning stage is called *training*. Inputs are as a pattern of numbers—one number per network input. That makes it easy to associate an input with a variable such that every variable has its corresponding input. Outputs are also a pattern of numbers—one number per output. Each output is associated with an output variable. Each of the inputs and outputs is associated with a "neuron," so there are input neurons and output neurons. Sandwiched between these two kinds of neurons is another set of neurons called the *hidden layer*, so called for the same reason that the cheese in a cheese sandwich is hidden from the outside world by the bread. So too are the hidden neurons hidden from the world by the input and output neurons. Figure 10.3 shows schematically a typical representation of a neural network with three input neurons, two hidden neurons, and one output neuron. Each of the input neurons connects to each of the hidden neurons, and each of the hidden neurons connects to the output neuron. This configuration is known as a fully connected ANN.

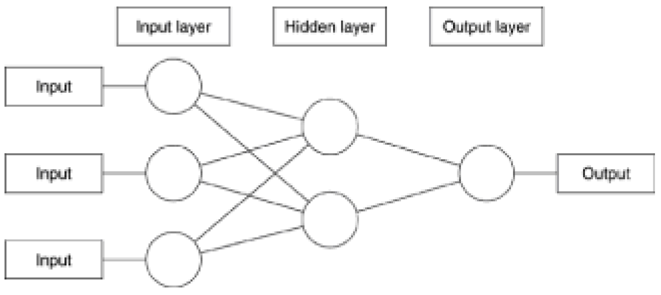


Figure 10.3 A three-input, one-output neural network with two neurons in the hidden layer.

The BP-ANN is usually in the form of a fully autonomous algorithm—often a compiled and ready-to-run computer program—which the miner uses. Use of a BP-ANN usually requires the miner only to select the input and output data that the network will train on, or predict about, and possibly some learning parameters. Seldom do miners write their own BP-ANN software today. The explanation here is to introduce the features and architecture of the BP-ANN that facilitate data compression and dimensionality reduction. This gives the miner an insight about why and how the information compression works, why the compressed output is in the form it is, and some insight into the limitations and problems that might be expected.

10.3.1 Training a Neural Network

Training takes place in two steps. During the first step, the network processes a set of input values and the matching output value. The network looks at the inputs and estimates the output—ignoring its actual value for the time being.

In the second step, the network compares the value it estimated and the actual value of the output. Perhaps there is some error between the estimated and actual values. Whatever it is, this error reflects back through the network, from output to inputs. The network adjusts itself so that, if those adjustments were used, the error would be made smaller. Since there are only neurons and connections, where are the adjustments made? Inside the neurons.

Each neuron has input(s) and an output. When training, it takes each of its inputs and multiplies them by a weight specific to that input. The weighted inputs merge together and pass out of the neuron as its response to these particular inputs. In the second step, back comes some level of error. The neuron adjusts its internal weights so that the actual neuron output, for these specific inputs, is closer to the desired level. In other words, it adjusts to reduce the size of the error.

This reflecting the output error backwards from the output is known as propagating the error backwards, or *back-propagation*. The back-propagation referred to in the name of the network only takes place during training. When predicting, the weights are frozen, and only the forward-propagation of the prediction takes place.

Neural networks, then, are built from neurons and interconnections between neurons. By continually adjusting its internal neuron weightings to reduce the error of each neuron’s predictions, the neural network eventually learns the correct output for any input, if it is possible. Sometimes, of course, the output is not learnable from the information contained in the input. When it is possible, the network learns (in its neurons) the relationship between inputs and output. In many places in this book, those relationships are described as curved manifolds in state space. Can a neural network learn any conceivable manifold shape? Unfortunately not. The sorts of relationship that a neural network can learn are those that can be described by a function—but it is potentially any function! (A *function* is a mathematical device that produces a single output value for every set of input values. See Chapter 6 for a discussion of functions, and relationships not describable by functions.) Despite the limitation, this is remarkable! How is it that changing the weights inside neurons, connected to other neurons in layers, can create a device that can learn what may be complex nonlinear functions? To answer that question, we need to take a much closer look at what goes on inside an artificial neuron.

10.3.2 Neurons

Neurons are so called because, to some extent, they are modeled after the functionality of units of the human brain, which is built of biochemical neurons. The neurons in an artificial neural network copy some of the simple but salient features of the way biochemical neurons are believed to work. They both perform the same essential job. They take several inputs and, based on those inputs, produce some output. The output reflects the state and value of the inputs, and the error in the output is reduced with training.

For an artificial neuron, the input consists of a number. The input number transfers across the inner workings of the neuron and pops out the other side altered in some way. Because of this, what is going on inside a neuron is called a *transfer function*. In order for the network as a whole to learn nonlinear relationships, the neuron's transfer function has to be nonlinear, which allows the neuron to learn a small piece of an overall nonlinear function. Each neuron finds a small piece of nonlinearity and learns how to duplicate it—or at least come as close as it can. If there are enough neurons, the network can learn enough small pieces in its neurons that, as a whole, it learns complete, complex nonlinear functions.

There are a wide variety of neuron transfer functions. In practice, by far the most popular transfer function used in neural network neurons is the logistic function. (See the Supplemental Material section at the end of Chapter 7 for a brief description of how the logistic function works.) The logistic function takes in a number of any value and produces as its output a number between 0 and 1. But since the exact shape of the logistic curve can be changed, the exact number that comes out depends not only on what number was put in, but on the particular shape of the logistic curve.

10.3.3 Reshaping the Logistic Curve

First, a brief note about nomenclature. A function can be expressed as a formula, just as the formula for determining the value of the logistic function is

$$g = \frac{1}{1 + e^{-y}}$$

For convenience, this whole formula can be taken as a given and represented by a single letter, say *g*. This letter *g* stands for the logistic function. Specific values are input into the logistic function, which returns some other specific value between 0 and 1. When using this sort of notation for a function, the input value is shown in brackets, thus:

$y = g(10)$

This means that *y* gets whatever value comes out of the logistic function, represented by *g*, when the value 10 is entered. A most useful feature of this shorthand notation is that any valid expression can be placed inside the brackets. This nomenclature is used to indicate that the value of the expression inside the brackets is input to the logistic function, and the logistic function output is the final result of the overall expression. Using this notation removes much distraction, making the expression in brackets visually prominent.

10.3.4 Single-Input Neurons

A neuron uses two internal weight types: the *bias weight* and *input weights*. As discussed elsewhere, a bias is an offset that moves all other values by some constant amount. (Elsewhere, bias has implied noise or distortion—here it only indicates offsetting movement.) The bias weight moves, or biases, the position of the logistic curve. The input weight modifies an input value—effectively changing the shape of the logistic curve. Both of these weight types are adjustable to reduce the back-propagated error.

The formula for this arrangement of weights is exactly the formula for a straight line:

$$y_n \text{ x } a_0 + b_n x_n$$

So, given this formula, exactly what effect does adjusting these weights have on the logistic function's output? In order to understand each weight's effects, it is easiest to start by looking at the effect of each type of weight separately. In the following discussion a one-input neuron is used so there is a single-bias weight and a single-input weight. First, the bias weight.

Figure 10.4 shows the effect on the logistic curve for several different bias weights. Recall that the curve itself represents, on the *y* (vertical) axis, values that come out of the logistic function when the values on the *x* (horizontal) axis represent the input values. As the bias weight changes, the position of the logistic curve moves along the horizontal *x*-axis. This does not change the range of values that are translated by the logistic function—essentially it takes a range of 10 to take the function from 0 to 1. (The logistic function never reaches either 0 or 1, but, as shown, covers about 99% of its output range for a change in input of 10, say –5 to +5 with a bias of 0.)

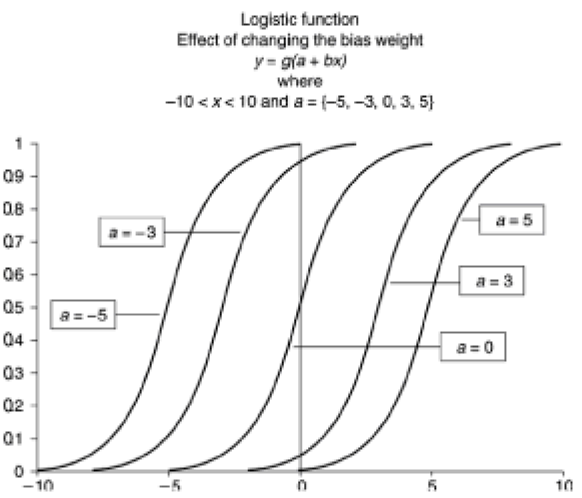


Figure 10.4 Changing the bias weight *a* moves the center of the logistic curve along the *x*-axis. The center of the curve, value 0.5, is positioned at the value of the bias weight.

The bias displaces the range over which the output moves from 0 to 1. In actual fact, it moves the center of the range, and why it is important that it is the center that moves will be seen in a moment. The logistic curves have a central value of 0.5, and the bias weight positions this point along the *x*-axis.

The input weight has a very different effect. Figure 10.5 shows the effect of changing the input weight. For ease of illustration, the bias weight remains at 0. In this image the shape of the curve stretches over a larger range of values. The smaller the input weight, the more widely the translation range stretches. In fact, although not shown, for very large values the function is essentially a “step,” suddenly switching from 0 to 1. For a value of 0, the function looks like a horizontal line at a value of 0.5.

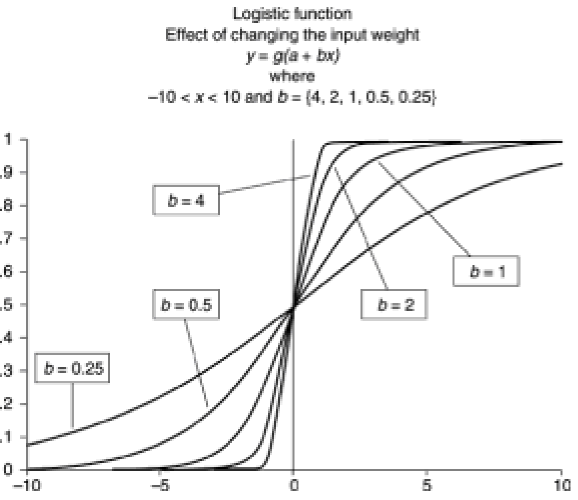


Figure 10.5 Holding the bias weight at 0 and changing the input weight *b*

changes the transition range of the logistic function.

Figure 10.6 has similar curves except that they all move in the opposite direction! This is the result of using a negative input weight. With positive weights, the output values translate from 0 to 1 as the input moves from negative to positive values of *x*. With negative input weights, the translation moves from 1 toward 0, but is otherwise completely adjustable exactly as for positive weights.

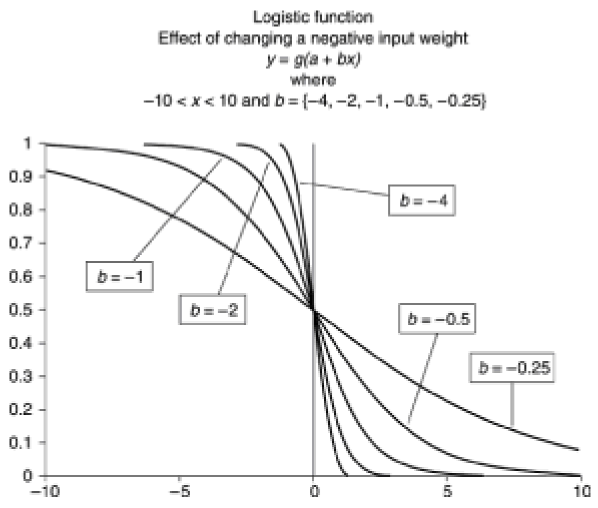


Figure 10.6 When the input weight is negative, the curve is identical in shape to a positively weighted curve, except that it moves in the opposite direction—positive to negative instead of negative to positive.

The logistic curve can be positioned and shaped as needed by the use of the bias and input weights. The range, slope, and center of the curve are fully adjustable. While the characteristic shape of the curve itself is not modified, weight modification positions the center and range of the curve wherever desired.

This is indeed what a neuron does. It moves its transfer function around so that whatever output it actually gives best matches the required output—which is found by back-propagating the errors.

Well, it can easily be seen that the logistic function is nonlinear, so a neuron can learn at least that much of a nonlinear function. But how does this become part of a complex nonlinear function?

10.3.5 Multiple-Input Neurons

So far, the neuron in the example has dealt with only one input. Whether the hidden layer neurons have multiple inputs or not, the output neuron of a multi-hidden-node network must deal with multiple inputs. How does a neuron weigh multiple inputs and pass them across its transfer function?

Figure 10.7 shows schematically a five-input neuron. Looking at this figure shows that the bias weight, *a0*, is common to all of the inputs. Every input into this neuron shares the effect of this common bias weight. The input weights, on the other hand, *bn*, are specific to each input. The input value itself is denoted by *xn*.

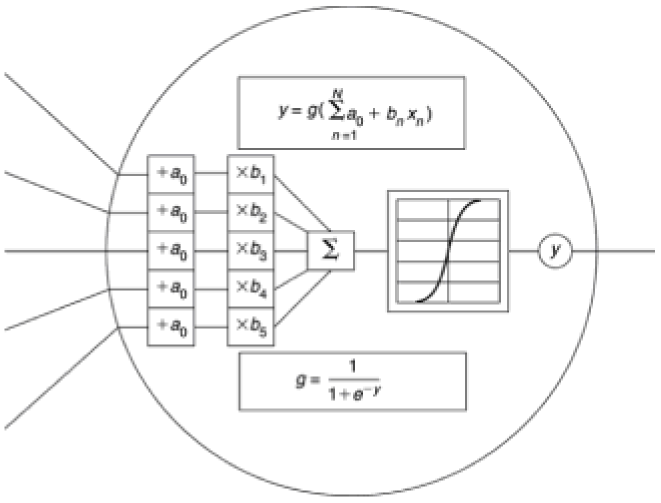


Figure 10.7 The “Secret Life of Neurons”! Inside a neuron, the common bias weight (*a*0®MDNM¯) is added to all inputs, but each separate input is multiplied by its own input weight (*bn*). The summed result is applied to the transfer function, which produces the neuron’s output (*y*).

There is an equation specific to each of the five inputs:

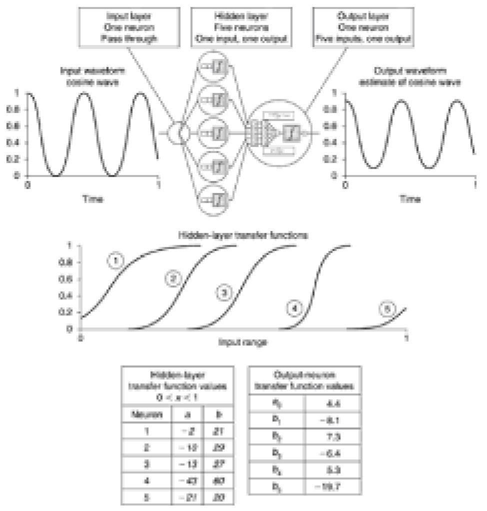
$$y_n = a_0 + b_n x_n$$

where *n* is the number of the input. In this example, *n* ranges from 1 to 5. The neuron code evaluates the equations for specific input values and sums the results. The expression in the top box inside the neuron indicates this operation. The logistic function (shown in the neuron’s lower box) transfers the sum, and the result is the neuron’s output value.

Because each input has a separate weight, the neuron can translate and move each input into the required position and direction of effect to approximate the actual output. This is critical to approximating a complex function. It allows the neuron to use each input to estimate part of the overall output and assembles the whole range of the output from these component parts.

10.3.6 Networking Neurons to Estimate a Function

Figure 10.8 shows a complete one-input, five-hidden-neuron, one-output neural network. There are seven neurons in all. The network has to learn to reproduce the 2 cycles of cosine wave shown as input to the network.



Hidden-layer transfer function values $0 < x < 1$		
Neuron	a	b
1	−2	21
2	−10	29
3	−13	27
4	−43	60
5	−21	20

Output neuron transfer function values	
b_0	4.4
b_1	−8.1
b_2	7.3
b_3	−6.4
b_4	5.3
b_5	−19.7

Figure 10.8 A neural network learning the shape of a cosine waveform. The input neuron splits the input to the hidden neurons. Each hidden neuron learns part of the overall wave shape, which the output neuron reassembles when prediction is required.

The input neuron itself serves only as a placeholder. It has no internal structure, serving only to represent a single input point. Think of it as a “splitter” that takes the single input and splits it between all of the neurons in the hidden layer. Each hidden-layer neuron “sees” the whole input waveform, in this case the 2 1/4 cosine wave cycles. The amplitude of the cosine waveform is 1 unit, from 0 to 1, corresponding to the input range for the logistic transfer function neurons. The limit in output range of 0–1 requires that the input range be limited too. Since the neuron has to try to duplicate the input as its output, then the input has to be limited to the range the neuron actually can output. The “time” range for the waveform is also normalized to be across the range 0–1, again matching the neuron output requirements.

The reexpression of the time is necessary because the network has to learn to predict the value of the cosine wave at specific times. When predicting with this network, it will be asked, in suitably anthropomorphic form, “What is the value of the function at time *x*?” where *x* is a number between 0 and 1.

Each hidden-layer neuron will learn part of the overall waveform shape. Figure 10.9 shows why five neurons are needed. Each neuron can move and modify the exact shape of its logistic transfer function, but it is still limited to fitting the modified logistic shape to part of the pattern to be learned as well as it can. The cosine waveform has five roughly logistic-function-shaped pieces, and so needs five hidden-layer neurons to learn the five pieces.

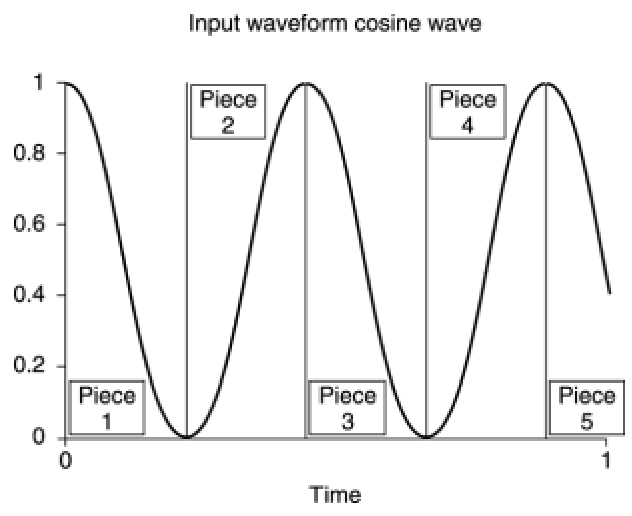


Figure 10.9 Learning this waveform needs at least five neurons. Each neuron can only learn an approximately logistic-function-shaped piece of the overall waveform. There are five such pieces in this wave shape.

10.3.7 Network Learning

During network setup, the network designer takes care to set all of the neuron weights at random. This is an important part of network learning. If the neuron weights are all set identically, for instance, each neuron tries to learn the same part of the input waveform as all of the other neurons. Since identical errors are then back-propagated to each, they all continue to be stuck looking at one small part of the input, and no overall learning takes place. Setting the weights at random ensures that, even if they all start trying to approximate the same part of the input, the errors will be different. One of the neurons predominates and the others wander off to look at approximating other parts of the curve. (The algorithm uses sophisticated methods of ensuring that the neurons do all wander to different parts of the overall curve, but they do not need to be explored here.)

Training the network requires presenting it with instances one after the other. These instances, of course, comprise the miner-selected training data set. For each instance of data presented, the network predicts the output based on the state of its neuron weights. At the output there is some error (difference between actual value and predicted value)—even if in a particular instance the error is 0. These errors are accumulated, not fed back on an instance-by-instance basis. A complete pass through the training data set is called an *epoch*. Adequately training a neural network usually requires many epochs. Back-propagation only happens at the end of each epoch. Then, each neuron adjusts its weights to better modify and fit the logistic curve to the shape of its input. This ensures that each neuron is trying to fit its own curve to some “average” shape of the overall input waveform.

Overall, each neuron tries to modify and fit its logistic function as well as possible to some part of the curve. It may succeed well, or it may do very poorly, but when training is complete, each approximates a part of the input as well as possible. The miner determines the criteria that determine training to be "complete." Usually, training stops either when the input wave shape can be re-created with less than some selected level of error, say, 10%, or when a selected number of epochs have passed without any improvement in the prediction.

It is usual to reserve a test data set for use during training. The network learns the function from the training set, but fitting the function to the test data determines that training is complete. As training begins, and the network better estimates the needed function in the training data set, the function improves its fit with the test data too. When the function learned in the training data begins to fit the test data less well, training is halted. This helps prevent learning noise. (Chapter 2 discusses sources of noise, Chapter 3 discusses noise and the need for multiple data sets when training, and Chapter 9 discusses noise in time series data, and waveforms.)

10.3.8 Network Prediction—Hidden Layer

So what has the network learned, and how can the cosine waveform be reproduced? Returning to Figure 10.8, after training, each hidden-layer neuron learned part of the waveform. The center graph shows the five transfer functions of the individual hidden-layer neurons. But looking at these transfer functions, it doesn't appear that putting them together will reproduce a cosine waveform!

Observe, however, that the transfer functions for each neuron are each in a separate position of the input range, shown on the (horizontal) *x*-axis. None of the transfer functions seems to be quite the same shape as any other, as well as being horizontally shifted. The actual weights learned for each hidden neuron are shown in the lower-left box. It is these weights that modify and shape the transfer function. For any given input value (between 0 and 1), the five neurons will be in some characteristic state.

Suppose the value 0.5 is input—what will be the state of the hidden-layer neurons? Hidden neurons 1 and 2 will both produce an output close in value to 1. Hidden neuron 3 is just about in the middle of its range and will produce an output close to 0.5. Hidden neurons 4 and 5 will produce an output close to 0. So it is for any specific input value—the hidden neurons will each produce a specific value.

But these outputs are not yet similar to the original cosine waveform. How can they be assembled to resemble the input cosine waveform?

10.3.9 Network Prediction—Output Layer

The task of the output neuron involves taking as input the various values output by the hidden layer and reproducing the input waveform from them. This, of course, is a multiple-input neuron. The lower-right box in Figure 10.8 shows the learned values for its inputs. The bias weight (a_0) is common, but the input weights are each separate. Careful inspection shows that some of them are negative. Negative input weights, recall, have the effect of “flipping” the direction in which the transfer function moves. In fact, the first, third, and fifth weights (b_1, b_3, b_5) are all negative. During the part of the input range when these hidden-layer neurons are changing value, their positive going change will be translated at the output neuron into a negative going change. It is these weights that change the direction of the hidden-layer transfer functions.

The output layer sums the inputs, transfers the resulting value across its own internal function, and produces the output shown. Clearly the network did not learn to reproduce the input perfectly. More training would improve the *shape* of the output. In fact, with enough training, it is possible to come as close as the miner desires to the original shape. But there is another distortion. The range of the original input was 0 to 1. The smallest input value was actually 0, while the largest was actually 1. The output seems to span a range of about 0.1 to 0.9. Is this an error? Can it be corrected?

Unfortunately, the logistic function cannot actually reach values of 0 or 1. Recall that it is this feature that makes it so useful as a squashing function (Chapter 7). To actually reach values of 0 or 1, the input to the logistic function has to be infinitely negative or infinitely positive. This allows neural networks to take input values of any size during modeling. However, the network will only actually “see” any very significant change over the linear part of the transfer function—and it will only produce output over the range it “sees” in the input.

10.3.10 Stochastic Network Performance

A neural network is a stochastic device. *Stochastic* comes from a Greek word meaning “to aim at a mark, to guess.” Stochastic devices work by making guesses and improving their performance, often based on error feedback. Their strength is that they usually produce approximate answers very quickly. Approximate can mean quite close to the precise answer (should one exist) or having a reasonably high degree of confidence in the answer given. Actually producing exact answers requires unlimited repetitions of the feedback cycle—in other words, a 100% accurate answer (or 100% confidence in the answer) takes, essentially, forever.

This makes stochastic devices very useful for solving a huge class of real-world problems. There are an enormous number of problems that are extremely difficult, perhaps impossible, to solve exactly, but where a good enough answer, quickly, is far better than an exact answer at some very remote time.

Humans use stochastic techniques all the time. From grocery shopping to investment analysis, it is difficult, tedious, and time-consuming, and most likely impossible in practice, to get completely accurate answers. For instance, exactly—to the nearest whole molecule—how much coffee will you require next week? Who knows? Probably a quarter of a pound or so will do (give or take several trillions of molecules). Or again—compare two investments: one a stock mutual fund and the other T-bills. Precisely how much—to the exact penny, and including all transaction costs, reinvestments, bonuses, dividends, postage, and so on—will each return over the next 10 years (to the nearest nanosecond)? Again, who knows, but stocks typically do better over the long haul than T-bills. T-bills are safer. But only safer stochastically. If a precise prediction was available, there would be no uncertainty.

With both of these examples, more work will give more accurate results. But there comes a point at which good enough is good enough. More work is simply wasted. A fast, close enough answer is useable now. A comprehensive and accurate answer is not obtainable in a useful time frame.

Recall that at this stage in the data preparation process, all of the variables are fully prepared—normalized, redistributed, and no missing values—and all network input values are known. Because of this, the dimensionality collapse or reduction part of data preparation doesn't use another enormously powerful aspect of stochastic techniques. Many of them are able to make estimates, inferences, and predictions when the input conditions are uncertain or unknown. Future stock market performance, for instance, is impossible to accurately predict—this is intrinsically unknowable information, not just unknown-but-in-principle-knowable information. Stochastic techniques can still estimate market performance even with inadequate, incomplete, or even inaccurate inputs.

The point here is that while it is not possible for a neural network to produce 100% accurate predictions in any realistic situation, it will quickly come to some estimate and converge, ever more slowly, never quite stopping, toward its final answer. The miner must always choose some acceptable level of accuracy or confidence as a stopping criterion. That accuracy or confidence must, of necessity, always be less than 100%.

10.3.11 Network Architecture 1—The Autoassociative Network

There are many varieties of neural networks. Many networks work on slightly different principles than the BP-ANN described here, and there are an infinite variety of possible network architectures. The architecture, in part, defines the number, layout, and connectivity of neurons within a network. Data preparation uses a class of architectures called *autoassociative networks*.

One of the most common neural network architectures is some variant of that previously shown in Figure 10.3. This type of network uses input neurons, some lesser number of hidden neurons fully connected to the input layer, and one, or at most a few, output neurons fully connected to the hidden layer. Such networks are typically used to predict the value of an output(s) when the output value(s) is not included in the input variables—for example, predicting the level of tomorrow's stock market index given selected current information.

An autoassociative network has a very different architecture, as shown in Figure 10.10. A key point is that the number of inputs and outputs are identical. Not only is the number identical, but all of the values put into the network are also applied as outputs! This network is simply learning to predict the value of its own inputs. This would ordinarily be a trivial task, and as far as predicting values for the outputs goes, it is trivial. After all, the value of the inputs is known, so why predict them?

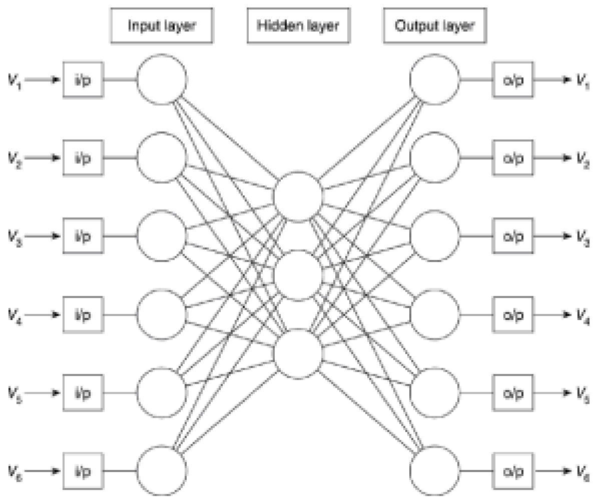


Figure 10.10 An autoassociative neural network has the same number of inputs and outputs. Each value applied as an input is also an output. The network learns to duplicate all of the inputs as outputs.

The key to this answer lies in the hidden layer. The hidden layer has less neurons than either input or output. The importance of this is discussed in a moment.

10.3.12 Network Architecture 2—The Sparsely Connected Network

The networks discussed so far were all fully connected. That is to say, every neuron in the input layer connects to every neuron in the hidden layer, and every neuron in the hidden layer connects to every neuron in the output layer. For a fully connected autoassociative neural network, the number of interconnections rises quite steeply as the dimensionality of the data set to be modeled increases. The actual number of interconnections is twice the number of dimensions multiplied by the number of neurons in the hidden layer:

$c = 2(n_i \times n_h)$

where

c	is number of interconnections
n_i	is number of inputs
n_h	is number of hidden neurons

For the example shown in Figure 10.10, that number is 2(6 x 3)= 36. For a 100-input, 50-hidden neuron network, the number is already 10,000. With 1000 inputs (not untypical with data compression problems), a 50% reduction in the size of the hidden layer requires 1,000,000 connections in a fully interconnected neural network. Each interconnection requires a small calculation for each instance value. When multiplied by the number of instances in a representative high-dimensionality data set, multiplied again by the number of training epochs, the number of calculations required to converge a large autoassociative network becomes truly vast.

Fortunately, there is an easy way around the problem. Any inputs to a neuron that are not being used to influence the transfer function can be ignored. A neuron with 1000 inputs may be using only a small fraction of them, and the others need not be calculated. A caveat to such a connection reduction method requires each unused interconnection to be sampled occasionally to see if it makes a difference if used. As the neuron moves its transfer function, it may be able to use other information from additional inputs. Dropping internal connections in this way can lead to a 90% reduction in interconnections—with a concomitant increase in training speed. When so configured, this is called a *sparsely connected network*. Although the predictive power of the network does degrade as the interconnectivity level falls, it is a graceful degradation. Such networks frequently retain 90% of their power with only 10% of the fully interconnected connections. (The exact performance depends very highly on the complexity of the function required, and so on.) A sparsely connected network that is making a good estimation of the required output values also strongly indicates that the required function can be approximated well with less hidden neurons.

10.4 Compressing Variables

The basic data compression tool, and the one included with the demonstration software on the accompanying CD-ROM, is the sparsely connected autoassociative neural network (SCANN). Its one task is to learn to predict its own inputs to some selected level of confidence—but with less neurons in the hidden layer than there are in the input and output layers. What does this achieve?

If the connections were “straight through,” that is, input connected directly to output, the task would be easy. Each input would perfectly predict the output, since the required prediction for the output value is the input value! But the fact is that there is not a direct connection. Whatever information is contained in the input has to be compressed into the hidden neurons. These neurons seek a relationship such that a few of them can predict, as accurately as needed, many outputs. And here, when the hidden neurons number less than the inputs, the information is squeezed into the hidden neurons. This is compressed information. Inasmuch as the hidden neurons can, at the outputs, re-create all of the values at the input, so they hold all of the information at the input, but in less neurons.

Suppose that a trained SCANN network is split between the hidden layer and the output layer. As instance values of a data set are applied to the input, the hidden-layer outputs are recorded in a file—one variable per hidden neuron. This “hidden-layer” file has less variables than the original file, yet holds information sufficient to re-create the input file. Applying the captured data to the output neurons re-creates the original file. This combination of outputs from the hidden neurons, together with the transform in the output neurons, shows that the information from many variables is compressed into fewer.

Compression serves to remove much of the redundancy embedded in the data. If several variables can be predicted from the same embedded information, that information only needs to be recognized once by the hidden neurons. Once the relationship is captured, the output neurons can use the single relationship recognized by the hidden neurons to re-create the several variables.

10.4.1 Using Compressed Dimensionality Data

When the miner and domain expert have a high degree of confidence that the training data set for the compression model is fully representative of the execution data, compression works well. A problem with compression is that the algorithm, at execution time, cannot know if the data being compressed can be recovered. If the execution data moves outside of the training sample parameters, the compression will not work, and the original values cannot be recovered. There are ways to establish a confidence level that the execution data originates from the same population distribution as the training sample (discussed in Chapter 11; see “Novelty Detection”), and these need to be used along with compression techniques.

Compressing data can offer great benefits. Compression is used because the dimensionality of the execution data set is too high for any modeling method to actually model the uncompressed data. Since a representative subset of the whole data set is used to build the compression model (part of the PIE described in Chapter 2), the compression model can be built relatively quickly. Using that compression model, the information in the full data set is quickly compressed for modeling. Compression, if practicable, reduces an intractable data set and puts it into tractable form. The compressed data can be modeled using any of the usual mining tools available to the miner, whereas the original data set cannot.

Creating a predictive model requires that the variable(s) to be predicted are kept out of the compressed data set. Because at execution time the prediction variable's value is unknown, it cannot be included for compression. If there are a very large number of prediction variables, they could be compressed separately and predicted in their compressed form. The decompression algorithm included in the PIE-O (see Chapter 2) will recover the actual predicted values.

Such compression models have been used successfully with both physical and behavioral data. Monitoring large industrial processes, for example, may produce data streams from high hundreds to thousands of instrumented monitoring points throughout the process. Since many instrumentation points very often turn out to be correlated (carry similar information), such as flow rates, temperature, and pressure, from many points, it is possible to compress such data for modeling very effectively. Compression models of the process require data from both normal and abnormal operations, as well as careful automated monitoring of the data stream to ensure that the data remains with the model's limits. This allows successful, real-time modeling and optimization of vast industrial processes.

Compressing large telecommunications data sets has also been successful for problem domains in which customer behavior changes relatively slowly over time. Huge behavioral data sets can be made tractable to model, particularly as many of the features are correlated, although often not highly. In very high dimensionality data sets there is enough redundancy in the information representation to facilitate good compression ratios.

10.5 Removing Variables

This is a last-ditch, when-all-else-fails option for the miner. However, sometimes there is nothing else to do if the dimensionality exceeds the limits of the compression or modeling tool and computer resources available.

The ideal solution requires removing redundant variables only. Redundant variables show a high degree of possibly nonlinear correlation. The solution, then, seems to be to remove variables that are most highly nonlinearly correlated. (Note that linearly correlated variables are also correlated when using nonlinear estimates. Linear correlation is, in a sense, just a special case of nonlinear correlation.) Unfortunately, there are many practical problems with discovering high-dimensionality nonlinear correlations. One of the problems is that there are an unlimited number of degrees of nonlinearity. However deep a so-far-fruitless search, it's always possible some yet greater degree of nonlinearity will show a high correlation. Another problem is deciding whether to compare single correlation (one variable with another one at a time) or multiple correlation (one variable against several at a time in combinations). Conducting such a search for massive dimensionality input spaces can be just as large a problem as trying to model the data—which couldn't be done, hence the search for variables to remove in the first place!

Once again the SCANN steps forward. Due to the sparse interconnections within the SCANN, it will build models of very high dimensionality data sets. The models will almost certainly be poor, but perhaps surprisingly, it is not the models themselves that are of interest when removing variables.

Recall that the weights of a SCANN are assigned random values when the network is set up. Even identically configured networks, training on identical data sets, are initiated with different neuron weights from training run to training run. Indeed, which neuron learns which piece of the overall function will very likely change from run to run. The precise way that two architecturally identical networks will represent a function internally may be totally different even though they are identically trained—same data, same number of epochs, and so on. Random initialization of weights ensures that the starting conditions are different.

Caution: Do not confuse a training cycle with an epoch. An epoch is a single pass through the training set. A training cycle is start-to-finish training to some degree of network convergence selected by the miner, usually consisting of many epochs.

However, variables that are important to the network will very likely remain important from training session to training session, random initialization or not. More significant from the standpoint of removing variables is that variables that remain unimportant from session to session are indeed unimportant. Whether highly correlated with other variables or not, the network does not use them much to create the input-to-output relationship. If these unimportant variables can be detected, they can be removed. The question is how to detect them.

10.5.1 Estimating Variable Importance 1: What Doesn’t Work

There is great danger in talking about using neuron weights to estimate variable importance because the values of the weights themselves are not a measure of the variable’s importance. This is a very important point and bears repeating: The importance of a variable to estimating the input-to-output relationship cannot be determined from inspecting the value of the weights. This may seem counterintuitive, and before looking at what can help, it is easier to frame the problem by looking at why inspecting weight values does not work.

The problem with looking at weight values is that the effects of different weights can be indirect and subtle. Suppose an input weight is very heavy—does it have a large effect on the output function? Not if the output neurons it connects to ignore it with a low weight. Or again, suppose that an input is very lightly weighted—is it unimportant? Not if all of the output neurons weight it heavily, thus amplifying its effect since it participates in every part of the function estimation.

With highly correlated inputs, just the sort that we are looking for, it may well be that the weights for two of them are very high—even if together they make no effect on the output! It could be that one of them has a large weight for the sole and exclusive purpose of nullifying the effect of the other large weight. The net result on the input-to-output function is nil.

If trying to untangle the interactions between input and hidden layer is hard, determining the effect over the whole input range of the network at the output neuron(s) is all but impossible.

There is no doubt that the values of the weights do have great significance, but only in terms of the entire structure of interactions within the network. And that is notoriously impenetrable to human understanding. The problem of explaining what a particular network configuration "means" is all but impossible for a network of any complexity.

10.5.2 Estimating Variable Importance 2: Clues

So if looking at the neuron weights doesn't help, what's left? Well, actually, it's the neuron weights. But it is not the level of the weights, it's the change in the weight from training cyle to training cycle.

Recall that, even for high-dimensionality systems, a SCANN trains relatively quickly. This is particularly so if the hidden layer is small relative to the input count. Speedy training allows many networks to be trained in a reasonable time. It is the "many networks" that is the key here. The automatic algorithm estimating variable importance starts by capturing the state of the initialized weights before each training cycle. After each training cycle is completed, the algorithm captures the adjusted state of the weights. It determines the difference between initial setting and trained setting, and accumulates that difference for every input weight. (The input neurons don't have any internal structure; the algorithm captures the input weights at the hidden layer.)

Looking at the total distance that any input weight has moved over the total number of training cycles gives an indication of importance. If the network always finds it necessary to move a weight (unless by happenstance it was set near to the correct weight when set up), it is clearly using the input for something. If consistently used, the input may well be important. More significantly, when looking for unimportant variables, if the network doesn't move the weight for an input much from cycle to cycle, it is definitely not using it for much. Since each weight is initially set at random and therefore takes on a range of values, if nonmoving during training, it clearly does not really matter what weighting the input is given. And if that is true, it won't matter much if the weighting is 0—which definitely means it isn't used.

Here is where to find clues to which variables are important. It is very unlikely that any variable has weights that are never moved by the network. It is the nature of the network to at least try many different configurations, so all weights will almost certainly be moved somewhat over a number of training cycles. However, when data set dimensionality is high, removing variables with small weight movement at least removes the variables that the network uses the least during many training cycles.

10.5.3 Estimating Variable Importance 3: Configuring and Training the Network

Configuring the training parameters for assessing variable importance differs from other training regimes. The quality of the final model is not important. It is important that the network does begin to converge. It is only when converging that the network demonstrates that it is learning. Network learning is crucial, since only when learning (improving performance) are the input variables accurately assessed for their importance. However, it is only important for variable importance assessment that significant convergence occurs.

Some rules of thumb. These are purely empirical, and the miner should adjust them in the light of experience. Actual performance depends heavily on the complexity of the relationship actually enfolded within the data set and the type and nature of the distributions of the variables, to mention only two factors. These are clearly not hard-and-fast rules, but they are a good place to start:

1. Initially configure the SCANN to have hidden-layer neurons number between 5% and 10% of the number of input variables.

2. Test train the network and watch for convergence. Convergence can be measured many ways. One measure is to create a pseudo-correlation coefficient looking simultaneously at network prediction error for all of the input/outputs from epoch to epoch. Ensure that the pseudo-r^2 (pr^2) measure improves by at least 35–50%. (See Chapter 9 for a description of r^2. Since this is a pseudo-correlation, it has a pseudo-r^2.) Failing that, use any measure of convergence that is convenient, but make sure the thing converges!

3. If the network will not converge, increase the size of the hidden layer until it does.

4. With a network configuration that converges during training, begin the Importance Detection Training (IDT) cycles.

5. Complete at least as many IDT cycles as the 0.6th power of the number of inputs (IDT cycles = $\text{Inputs}^{0.6}$).

6. Complete each IDT cycle when convergence reaches a 35–50% pr^2 improvement.

7. Cut as many input variables as needed up to 33% of the total in use.

8. If the previous cut did not reduce the dimensionality sufficiently for compression, go round again from the start.

9. Don’t necessarily cut down to modeling size; compress when possible.

10. Do it all again on the rejected variables. If true redundancy was eliminated from the first set, both sets of variables should produce comparable models.

Following this method, in one application the automated data reduction schedule was approximately as follows:

1. Starting with the 7000+-variable, multiterabyte data set mentioned earlier, a representative sample was extracted. About 2000 variables were collapsed into 5 as highly sparse variables. Roughly 5000 remained. The SCANN was initially configured with 500 hidden neurons, which was raised to 650 to aid convergence. 35% pr^2 convergence was required as an IDT terminating criterion. (Twenty-five epochs with less than 1% improvement was also a terminating criterion, but was never triggered.) After about 170 IDT cycles ($5000^{0.6} = 166$), 1500 variables were discarded.

2. Restarting with 3500 variables, 400 hidden neurons, 35% pr^2 convergence, and 140 IDT cycles, 1000 variables were discarded.

3. Restarting with 2500 variables, 300 hidden neurons, 35% pr^2 convergence, and 120 IDT cycles, 1000 variables were discarded.

4. The remaining 1500 variables were compressed into 700 with a minimum 90% confidence for data retention.

5. For the discarded 3500 variables left after the previous extraction, 2000 were eliminated using a similar method as above. This produced two data sets comprised of different sets of variables.

6. From both extracted variable data sets, separate predictive models were constructed and compared. Both models produced essentially equivalent results.

This data reduction methodology was largely automated, requiring little miner intervention once established. Running on parallel systems, the data reduction phase took about six days before modeling began. In a manual data reduction run at the same time, domain experts selected the variables considered significant or important, and extracted a data set for modeling. For this particular project, performance was measured in terms of “lift”—how much better the model did than random selection. The top 20% of selections for the domain expert model achieved a lift approaching three times. The extracted, compressed model for the top 20% of its selections produced a lift of better than four times.

The problem with this example is that it is impossible to separate the effect of the data reduction from the effect of data compression. As an academic exercise, the project suffers from several other shortcomings since much was not controlled, and much data about performance was not collected. However, as a real-world, automated solution to a particularly intractable problem requiring effective models to be mined from a massive data set to meet business needs, this data reduction method proved invaluable.

In another example, a data set was constructed from many millions of transaction records by reverse pivoting. The miners, together with domain experts, devised a number of features hypothesized to be predictive of consumer response to construct from the transaction data. The resulting reverse pivot produced a source data set for mining with more than 1200 variables and over 6,000,000 records. This data set, although not enormous by many standards (totaling something less than half a terabyte), was nonetheless too large for the mining tool the customer had selected, causing repeated mining software failures and system crashes during mining.

The data reduction methodology described above reduced the data set (no compression) to 35 variables in less than 12 hours total elapsed time. The mining tool was able to digest the reduced data set, producing an effective and robust model. Several other methods of selecting variables were tried on the same data set as a validation, but no combination produced a model as effective as the original combination of automatically selected variables.

Further investigation revealed that five of the reduction system selected variables carried unique information that, if not used, prevented any other model not using them from being as effective. The automatic reduction technique clearly identified these variables as of highest importance, and they were included in the primary model. Since they had been selected by the SCANN, they weren't used in the check data set made exclusively from the discards. Without these key variables, no check model performed as well. However, SCANN importance ranking singled them out, and further investigation revealed these five as the key variables. In this case, the discarded information was indeed redundant, but was not sufficient to duplicate the information retained.

10.6 How Much Data Is Enough?

Chapter 5 examined the question of how much data was enough in terms of determining how much data was needed to prepare individual variables. Variability turned out to be a part of the answer, as did establishing suitable confidence that the variability was captured. When considering the data set, the miner ideally needs to capture the variability of the whole data set to some acceptable degree of confidence, not just the variability of each variable individually. Are these the same thing?

There is a difference between individual variability and joint variability. Joint variability was discussed previously—for instance, in Chapter 6, maintaining joint variability between variables was used as a way of determining a suitable replacement value for missing values. Recall that joint variability is a measure of how the values of one (or more) variable's values change as another (or several) variable's values also change. The joint variability of a data set is a measure of how all of the values change based on all of the other variables in the data set.

Chapter 5 showed that because of the close link between distribution and variability, assuring that the distribution of a variable was captured to a selected degree of confidence assured that the variability of the variable was captured. By extension of the same argument, the problem of capturing enough data for modeling can be described in terms of capturing, to some degree of confidence, the joint distribution of the data set.

10.6.1 Joint Distribution

Joint distributions extend the ideas of individual distribution. Figure 10.11 shows a two-dimensional state space (Chapter 6 discusses state space) showing the positions of 100 data points. The distribution histograms for the two variables are shown above and to the right of the graphical depiction of state space. Each histogram is built from bars that cover part of the range. The height of each bar indicates the number of instance values for the variable in the range the bar covers. The continuous curved lines on the histograms trace a normal distribution curve for the range of each variable. The normal curve measures nothing about the variable—it is there only for comparison purposes, so that it is easier to judge how close the histogram distribution is to a normal distribution. Comparing each histogram with the normal curve shows that the variables are at least approximately normally distributed, since the bar heights roughly approximate the height of the normal curve. Looking at the state space graph shows, exactly as expected with such individual distributions, that the density of the data points is higher around the center of this state space than at its borders. Each individual variable distribution has maximum density at its center, so the joint density of the state space too is greatest at its center.

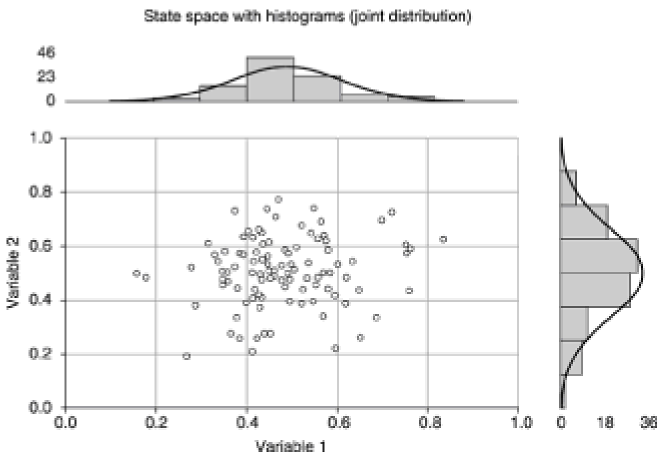


Figure 10.11 One hundred data points of two variables mapped into a 2D state

space with histograms showing the distribution of the individual variables compared with the normal curve.

The density of a one-dimensional distribution can be shown by the height of a continuous line, rather than histogram bars. This is exactly what the normal curve shows. Such a continuous line is, of course, a manifold. By extension, for two dimensions the density-mapping manifold is a surface.

Figure 10.12 shows the density manifold for the state space shown in Figure 10.11. The features in the manifold, shown by the unevenness (lumps and bumps), are easy to see. Every useful distribution has such features that represent variance in the density, regardless of the number of dimensions. (Distributions that have no variance also carry no useable information.) By extension from the single variable case (Chapter 5), as instance count in a sample increases, density structures begin to emerge. The larger the sample, the better defined and more stable become the features. With samples that are not yet representative of the population's distribution, adding more samples redefines (moves or changes) the features. When the sample is representative, adding more data randomly selected from the population essentially reinforces, but does not change, the features of the multidimensional distribution.

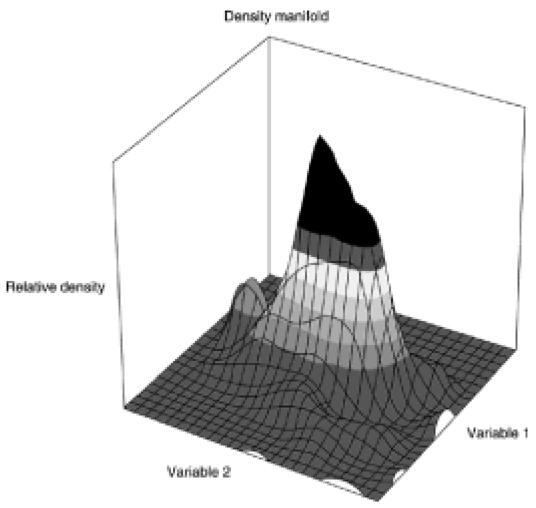


Figure 10.12 Density manifold for data points in a 2D state space. The manifold clearly shows density features ("lumps and bumps" in the manifold), as the density does not vary uniformly across state space.

It might seem that fully capturing the variability of individual variables is enough to capture the variability of the population. This, unfortunately, is not the case. A simple example shows that capturing individual variable variability is not necessarily sufficient to capture joint variability.

Suppose that two variables each have three possible values: 1, 2, and 3. The proportions in which each value occurs in the population of each variable are shown in Table 10.1.

TABLE 10.1   Value frequency for two variables.TABLE 10.1

Variable A values	Variable A proportion	Variable B values	Variable B proportion
1	10%	1	10%
2	20%	2	40%
3	70%	3	50%

Table 10.1 shows, for example, that 10% of the values of variable A are 1, 40% of the values of variable B are 2, and so on. It is very important to note that the table does not imply which values of variable A occur with which values of variable B. For instance, the 10% of variable A instances that have the value 1 may be paired with any value of variable B—1, 2, or 3. The table does not imply that values are in any way matched with each other.

Figure 10.13 shows distributions in samples of each variable. Recall that since each of the histograms represents the distribution of a sample of the population, each distribution approximates the distribution of the population and is not expected to exactly match the population distribution. Two separate samples of the population are shown, for convenience labeled “sample 1” and “sample 2.” The figure clearly shows that the sample 1 and sample 2 distributions for variable A are very similar to each other, as are those for variable B.

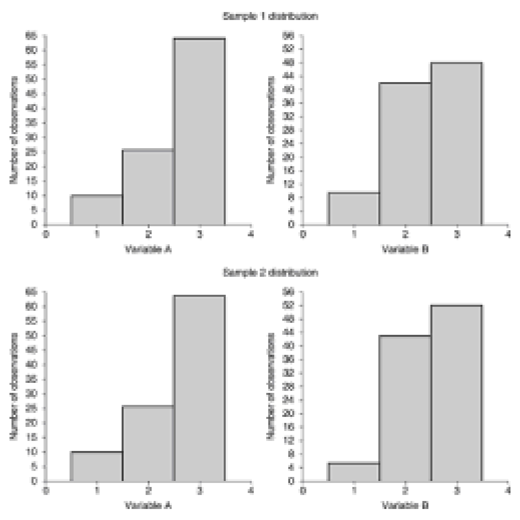


Figure 10.13 Individual distributions of two variables (A and B) from two representative samples (1 and 2) show very similar individual distributions in each sample.

If it is true that capturing the individual variability of each variable also captures the joint variability of both, then the joint variability of both samples should be very similar. Is it? Figure 10.14 shows very clearly that they are totally different. The histograms above and to the right of each joint distribution plot show the individual variable distributions. These distributions for both samples are exactly as shown in the previous figure. Yet the joint distribution, shown by the positions of the small circles, are completely different. The circles show where joint values occur—for instance, the circle at variable A = 3 and variable B = 3 shows that there are instances where the values of both variables are 3. It is abundantly clear that while the individual distributions of the variables in the two samples are essentially identical, the joint distributions are totally different.

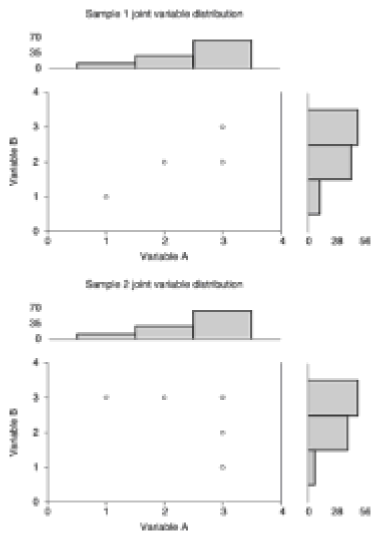


Figure 10.14 Although the individual variables' distributions are effectively

identical in each sample, the joint distribution, shown by the position of the circles, is markedly different between the two samples.

Figure 10.15 shows the joint histogram distributions for the two samples. The height of the columns is proportional to the number of instances having the A and B values shown. Looking at these images, the difference between the joint distributions is plain to see, although it is not so obvious that the individual distributions of the two variables are almost unchanged in the two samples. The column layout for the two samples is the same as the point layout in the previous figure. The joint histogram, Figure 10.15, shows the relative differences in joint values by column height. Figure 10.14 showed which joint values occurred, but without indicating how many of each there were.

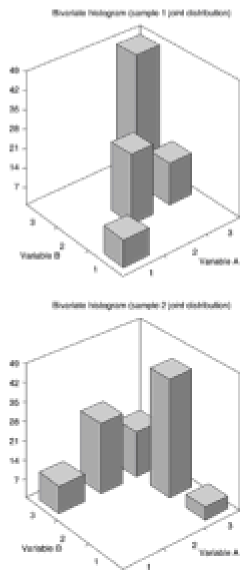


Figure 10.15 Joint distribution histograms for variables A and B are clearly seen as different between the two samples. The height of the column represents the number of instances having a particular joint combination of values.

The key point to understand from this example is that capturing individual variable variability to any degree of confidence does not in itself provide any degree of confidence that joint variability is captured.

10.6.2 Capturing Joint Variability

The clue to capturing joint variability has already been shown in Figure 10.12. This figure showed a relative-density plot. It mapped the density of a two-dimensional distribution. Very clearly it showed peaks and valleys—high points and low—representing the density of the distribution at any point on the map. Clearly, the exact shape of the surface reflects the underlying joint distribution of the sample, since the distribution is only an expression of the density of instance values in state space.

Just as a population has a specific, characteristic distribution for each individual variable (discussed in Chapter 5), so too the population has some particular characteristic joint distribution. As a sample gets larger, it better reflects the actual population distribution. At first, as instance values are added to the sample, the density manifold will change its shape—the peaks and valleys will move in state space. New ones may appear, and old ones disappear. When the sample is representative of the joint distribution, then the shape of the surface will change little as more data from the same population is added. This is no more than a multidimensional description of the way that a single variable was sampled in Chapter 5. By extension of the same discussion, multidimensional variability can be captured, to any selected degree of confidence, using density manifold stability.

But here is where data preparation steps into the data survey. The data survey (Chapter 11) examines the data set as a whole from many different points of view. Two of the questions a miner may need answers to are

1. Given a data set, what is the justifiable level of confidence that the joint variability has been captured?

2. Given a data set, how much more data is estimated to be needed to yield a given level of confidence that the sample is representative of the population?

Both of these questions are addressed during the survey. Both questions directly address the level of confidence justifiable for particular inferential or predictive models. The answers, of course, depend entirely on estimating the confidence for capturing joint variability.

Why is estimating joint variability confidence part of the survey, rather than data preparation? Data preparation concentrates on transforming and adjusting variables' values to ensure maximum information exposure. Data surveying concentrates on examining a prepared data set to glean information that is useful to the miner. Preparation manipulates values; surveying answers questions.

In general, a miner has limited data. When the data is prepared, the miner needs to know what level of confidence is justified that the sample data set is representative. If more data is available, the miner may ask how much more is needed to establish some greater required degree of confidence in the predictions or inferences. But both are questions about a prepared data set; this is not preparation of the data. Thus the answers fall under the bailiwick of data survey.

10.6.3 Degrees of Freedom

Degrees of freedom measure, approximately, how many things there are to change in a system that can affect the outcome. The more of them that there are, the more likely it is that, purely by happenstance, some particular, but actually meaningless, pattern will show up. The number of variables in a data set, or the number of weights in a neural network, all represent things that can change. So, yet again, high-dimensionality problems turn up, this time expressed as degrees of freedom. Fortunately for the purposes of data preparation, a definition of degrees of freedom is not needed as, in any case, this is a problem previously encountered in many guises. Much discussion, particularly in this chapter, has been about reducing the dimensionality/combinatorial explosion problem (which is degrees of freedom in disguise) by reducing dimensionality. Nonetheless, a data set always has some dimensionality, for if it does not, there is no data set! And having some particular dimensionality, or number of degrees of freedom, implies some particular chance that spurious patterns will turn up. It also has implications about how much data is needed to ensure that any spurious patterns are swamped by valid, real-world patterns. The difficulty is that the calculations are not exact because several needed measures, such as the number of significant system states, while definable in theory, seem impossible to pin down in practice. Also, each modeling tool introduces its own degrees of freedom (weights in a neural network, for example), which may be unknown to the minere .mi..

The ideal, if the miner has access to software that can make the measurements (such as data surveying software), requires use of a multivariable sample determined to be representative to a suitable degree of confidence. Failing that, as a rule of thumb for the minimum amount of data to accept, for mining (as opposed to data preparation), use *at least* twice the number of instances required for a data preparation representative sample. The key is to have enough representative instances of data to swamp the spurious patterns. Each significant system state needs sufficient representation, and having a truly representative sample of data is the best way to assure that.

10.7 Beyond Joint Distribution

So far, so good. Capturing the multidimensional distribution captures a representative sample of data. What more is needed? On to modeling!

Unfortunately, things are not always quite so easy. Having a representative sample in hand is a really good start, but it does not assure that the data set is modelable! Capturing a representative sample is an essential minimum—that, and knowing what degree of confidence is justified in believing the sample to be representative. However, the miner needs a modelable representative sample, and the sample simply being representative of the population may not be enough. How so?

Actually, there are any number of reasons, all of them domain specific, why the minimum representative sample may not suffice—or indeed, why a nonrepresentative sample is needed. (Heresy! All this trouble to ensure that a fully representative sample is collected, and now we are off after a nonrepresentative sample. What goes on here?)

Suppose a marketing department needs to improve a direct-mail marketing campaign. The normal response rate for the random mailings so far is 1.5%. Mailing rolls out, results trickle in. A (neophyte) data miner is asked to improve response. “Aha!,” says the miner, “I have just the thing. I’ll whip up a quick response model, infer who’s responding, and redirect the mail to similar likely responders. All I need is a genuinely representative sample, and I’ll be all set!” With this terrific idea, the miner applies the modeling tools, and after furiously mining, the best prediction is that no one at all will respond! Panic sets in; staring failure in the face, the neophyte miner begins the balding process by tearing out hair in chunks while wondering what to do next.

Fleeing the direct marketers with a modicum of hair, the miner tries an industrial chemical manufacturer. Some problem in the process occasionally curdles a production batch. The exact nature of the process failure is not well understood, but the COO just read a business magazine article extolling the miraculous virtues of data mining. Impressed by the freshly minted data miner (who has a beautiful certificate attesting to skill in mining), the COO decides that this is a solution to the problem. Copious quantities of data are available, and plenty more if needed. The process is well instrumented, and continuous chemical batches are being processed daily. Oodles of data representative of the process are on hand. Wielding mining tools furiously, the miner conducts an onslaught designed to wring every last confession of failure from the recalcitrant data. Using every art and artifice, the miner furiously pursues the problem until, staring again at failure and with desperation setting in, the miner is forced to fly from the scene, yet more tufts of hair flying.

Why has the now mainly hairless miner been so frustrated? The short answer is that while the data is representative of the population, it isn’t representative of the *problem*. Consider the direct marketing problem. With a response rate of 1.5%, any predictive system has an accuracy of 98.5% if it uniformly predicts “No response here!” Same thing with the chemical batch processing—lots of data in general, little data about the failure conditions.

Both of these examples are based on real applications, and in spite of the light manner of introducing the issue, the problem is difficult to solve. The feature to be modeled is insufficiently represented for modeling in a data set that is representative of the population. Yet, if the mining results are to be valid, the data set mined must be representative of the population or the results will be biased, and may well be useless in practice. What to do?

10.7.1 Enhancing the Data Set

When the density of the feature to be modeled is very low, clearly the density of that feature needs to be increased—but in a way that does least violence to the distribution of the population as a whole. Using the direct marketing response model as an example,