CHAPTER

8

Predictive Modeling

This chapter explains key ideas behind algorithms used in predictive modeling. The algorithms included in the chapter are those that are found most commonly in the leading predictive analytics software packages. There are, of course, many more algorithms available than those included here, including some of my favorites.

Predictive modeling algorithms are supervised learning methods, meaning that the algorithms try to find relationships that associate inputs to one or more target variables. The target variable is key: Good predictive models need target variables that summarize the business objectives well.

Consider the data used for the 1998 KDD Cup Competition (`http://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html`). The business objective is to identify lapsed donors that can be recovered with a future mailing. There are two potential target variables for this task: TARGET_B, the binary (flag) variable indicating whether or not a donor responded to the recovery mailing, or TARGET_D, the amount of the donation the donor gave if he or she responded to the recovery campaign. The algorithm you use depends in part on the target variable type: classification for categorical target variables and regression for continuous target variables.

Assume that the business objective states that a classification model should be built to predict the likelihood a lapsed donor will respond to a mailing. In this

case, you can build TARGET_B models with a classification algorithm. But which ones? What are the strengths and weaknesses of the different approaches? This chapter describes, primarily qualitatively, how the predictive modeling algorithms work and which settings you can change to improve predictive accuracy.

Predictive modeling algorithms can be placed into a few categories: Some algorithms are local estimators (decision trees, nearest neighbor), some are global estimators that do not localize (linear and logistic regression), and still others are functional estimators that apply globally but can localize their predictions (neural networks).

There are dozens of good predictive modeling algorithms, and dozens of variations of these algorithms. This chapter, however, describes only the algorithms most commonly used by practitioners and found in predictive analytics software. Moreover, only the options and features of the algorithms that have the most relevance for practitioners using software to build the models are included in the discussion.

Decision Trees

Decision trees are among the most popular predicting modeling techniques in the analytics industry. The 2009, 2010, 2011, and 2013 Rexer Analytics Data Miner surveys, the most comprehensive public surveys of the predictive analytics community, showed decision trees as the number 1 or number 2 algorithm used by practitioners, with the number of users ranging between 67 and 74 percent. Of the supervised learning algorithms, only Regression was used often by even more than one-third of the respondents.

Why such popularity? First, decision trees are touted as easy to understand. All trees can be read as a series of "if-then-else" rules that ultimately generate a predicted value, either a proportion of a categorical variable value likely to occur or an average value of the target variable. These are much more accessible to decision-makers than mathematical formulas. In addition, decision trees can be very easy to deploy in rule-based systems or in SQL.

A second reason that decision trees are popular is that they are easy to build. Decision tree learning algorithms are very efficient and scale well as the number of records or fields in the modeling data increase.

Third, most decision tree algorithms can handle both nominal and continuous inputs. Most algorithms either require all inputs to be numeric (linear regression, neural networks, k-nearest neighbor) or all inputs to be categorical (as Naïve Bayes).

Fourth, decision trees have a built-in variable selection. Coupled with their ability to scale with large numbers of inputs, they are excellent for data exploration with hundreds of variables of unknown power in predicting the target variable.

Fifth, decision trees are non-parametric, making no assumptions about distributions for inputs *or* the target variable. They can therefore be used in a wide variety of datasets without any transformations of inputs or outputs. Continuous variables do not have to conform to any particular distribution, can be multimodal, and can have outliers.

Sixth, many decision tree algorithms can handle missing data automatically rather than requiring the modeler to impute missing values prior to building the models. The method of imputation depends on the decision tree algorithm.

This chapter begins with an overview of the kinds of decision trees currently used by practitioners. It then describes how decision trees are built, including splitting criteria and pruning options. Last, it enumerates typical options available in software to customize decision trees.

The Decision Tree Landscape

Decision trees are relatively new to the predictive modeler's toolbox, coming from the Machine Learning world and entering into data mining software in the 1990s. The first tree algorithm still in use was the AID algorithm (Automatic Interaction Detection, 1963), later adding the chi-square test to improve variable selection in what became the CHAID algorithm (Chisquare Automatic Interaction Detection, 1980).

Meanwhile, in independent research, two other algorithms were developed. Ross Quinlan developed an algorithm called ID3 in the 1970s (documented in 1986), which used Information Gain as the splitting criterion. In 1993, he improved the algorithm with the development of C4.5 (1993), which used Gain Ratio (normalized Information Gain) as the splitting criterion. C4.5 was subsequently improved further in the algorithm Quinlan called C5.0, including improvements in misclassification costs, cross-validation, and boosting (ensembles). It also significantly improved the speed in building trees and built them with fewer splits while maintaining the same accuracy. The algorithm is sometimes referred to as just C5.

Another approach to building decision trees occurred concurrent with the development of ID3. The CART algorithm is described fully by Breiman, Friedman, Olshen, and Stone in the 1984 book *Classification and Regression Trees* (Chapman and Hall/CRC). CART uses the Gini Index as the primary splitting criterion and has been emulated in every major data mining and predictive analytics software package.

These are the three most common algorithms in use in predictive analytics software packages, although other tree algorithms are sometimes included,

such as the QUEST algorithm (Quick, Unbiased and Efficient Statistical Tree), which splits based on quadratic discriminant analysis, and other versions of trees that include fuzzy or probabilistic splits.

Decision tree terminology is very much like actual trees except that decision trees are upside down: Roots of decision trees are at the top, and leaves are at the bottom.

A *split* is a condition in the tree where data is divided into two or more subgroups. These are "if" conditions, such as "if petal width is <= 0.6, then go left, otherwise go right." The first split in a tree is called the *root split* or root node and is often considered the most important split in the tree. After the data is split into subgroups, trees formed on each of these subgroups are called a "branch" of the tree.

When the decision tree algorithm completes the building of a tree, terminal or leaf nodes are collectors of records that match the rule of the tree above the terminal node. Every record in the training dataset must ultimately end up in one and only one of the terminal nodes.

In Figure 8-1, you see a simple tree built using the classic Iris dataset, which has 150 records and 3 classes of 50 records each. Only two of the four candidate inputs in the Iris data are used for this tree: Petal length and petal width, sometimes abbreviated as pet_length and pet_width. The root node of the tree splits on petal width <= 0.6. When this condition is true, one goes down the left path of the tree and ends up in a terminal node with all 50 records belonging to the class setosa. On the right side of the root split, you encounter a second condition: petal width > 1.7. If this is also true, records end up in the terminal node with the class virginica. Otherwise, records end up in the terminal node with records that are largely belonging to the class versicolor.

The score for a record is the proportion of records for the most populated class in a terminal node. For example, the terminal node that contains all setosa Irises has a score of 100 related to the class setosa. Or put another way, any Iris that is found that has a petal width ≤ 0.6 inches has 100 percent likelihood of belonging to the setosa Iris species based on this data.

There is, therefore, one rule for every terminal node. For the tree in Figure 8-1, these are:

- If pet_width is ≤ 0.6, then the record belongs to class setosa with probability 100 percent.
- If pet_width > 0.6 and petal width > 1.7, then the record belongs to species virginica with probability 97.8.
- If pet_width is > 0.6 and ≤ 1.7, then the record belongs to species versicolor with probability 90.7 percent.

Trees are considered interpretable models because they can be read as rules. The interpretability, however, is severely reduced as the depth of the tree increases.

The depth is the number of conditions between the root split and the deepest terminal node, including the root split. In the tree shown in Figure 8-1, the depth is 2: first splitting on pet_width equal to 0.6 and then splitting on pet_width equal to 1.7.

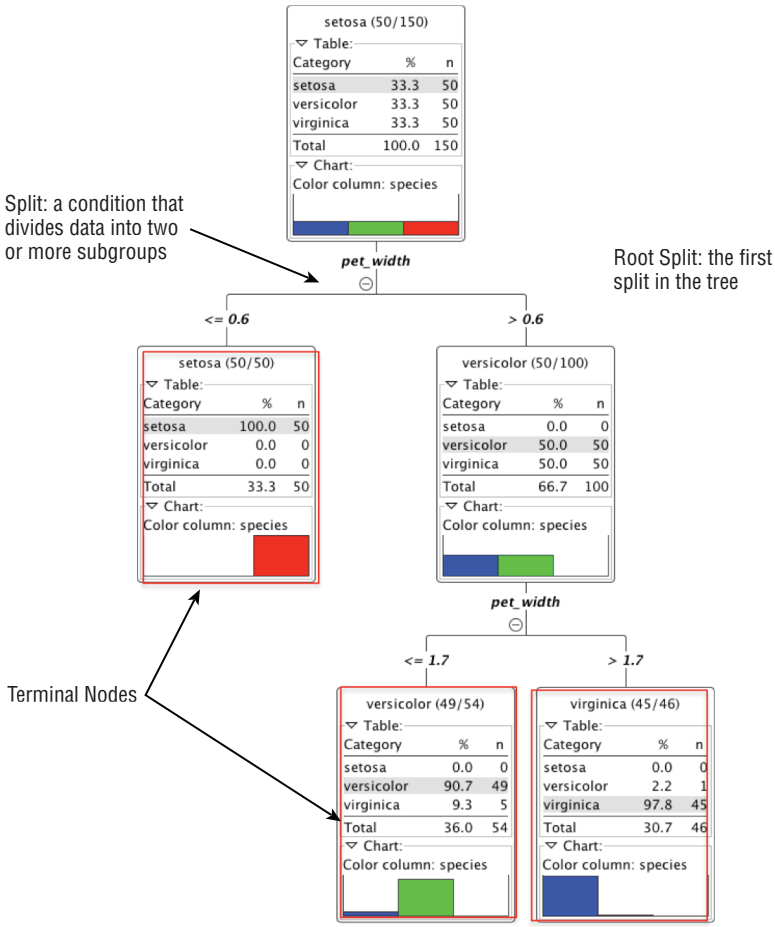

Figure 8-1: Simple decision tree built for the iris data

In a more complex tree (see Figure 8-2), every terminal has five conditions. The tree itself, even with a depth equal to only five, is too complex for the details of each node to be seen on this page. The terminal node at the far-bottom right can be expressed by this rule: If RFA_2F > 1 and CARDGIFT > 3 and LASTGIFT > 7 and AGE > 82 and RFA_2F > 2, then 46 of 100 donors do not respond. This rule that defines the records that fall into a terminal node, one of the 16 terminal nodes in the tree, has already lost considerable transparency and interpretability. What if there were 10 or 15 conditions in the rule? Decision trees are only interpretable if they are relatively simple. Figure 8-1 has a depth equal to two

and therefore is easy to see and understand. I find that trees become more difficult to interpret once the depth exceeds three. If the tree is many levels deep, you can still use the first two or three levels to provide a simpler explanation of the tree to others.

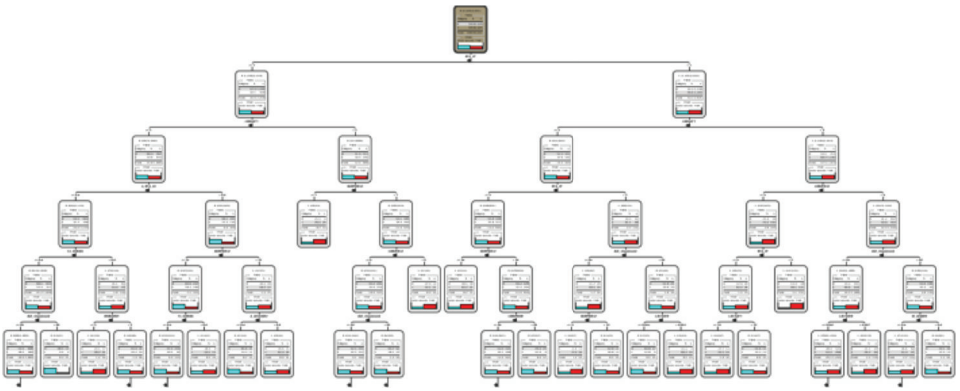

Figure 8-2: Moderately complex decision tree

Building Decision Trees

Decision trees belong to a class of *recursive partitioning algorithms* that is very simple to describe and implement. For each of the decision tree algorithms described previously, the algorithm steps are as follows:

1. For every candidate input variable, assess the *best* way to split the data into two or more subgroups. Select the best split and divide the data into the subgroups defined by the split.
2. Pick one of the subgroups, and repeat Step 1 (this is the recursive part of the algorithm). Repeat for every subgroup.
3. Continue splitting and splitting until all records after a split belong to the same target variable value or until another stop condition is applied. The stop condition may be as sophisticated as a statistical significance test, or as simple as a minimum record count.

The determination of *best* can be made in many ways, but regardless of the metric, they all provide a measure of purity of the class distribution. Every input variable is a candidate for every split, so the same variable could, in theory, be used for every split in a decision tree.

How do decision trees determine "purity"? Let's return to the Iris dataset and consider the variables petal width and petal length with three target variable classes, setosa (circle), versicolor (X-shape), and virginica (triangle), shown in Figure 8-3. It is obvious from the scatterplot that one good split will cleave out the setosa values by splitting on petal length anywhere between 2.25 and 2.75, or by splitting on petal width anywhere between the values 0.6 and 0.8.

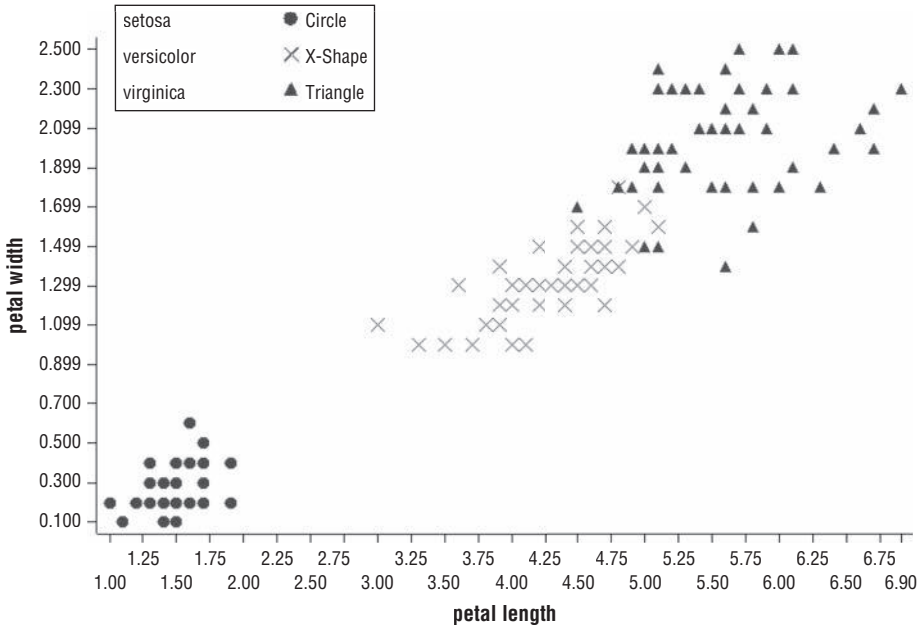

Figure 8-3: Iris data scatterplot of petal width versus petal length

Indeed, the CART algorithm first split could be on either petal length or petal width—both variables can identify perfectly the setosa class. Suppose petal length is selected. The second split selected by CART is at petal width > 1.75. The resulting regions from these two splits are shown in Figure 8-4. The shapes of the regions correspond to the shape of the target variable that is predominant.

However, what if the algorithm selected a different first split (equally as good). The decision regions created by the algorithm when the first split is petal width are shown in Figure 8-5. This tree has the same accuracy as the one visualized in Figure 8-4, but clearly defines different regions: In the first tree, large values of petal length and small values of petal width are estimated to be part of the class versicolor, whereas in the second tree with the same classification accuracy calls the same part of the decision space setosa.

Decision trees are nonlinear predictors, meaning the decision boundary between target variable classes is nonlinear. The extent of the nonlinearities depends on the number of splits in the tree because each split, on its own, is only a piecewise constant separation of the classes. As the tree becomes more complex, or in other words, as the tree depth increases, more piecewise constant separators are built into the decision boundary, providing the nonlinear separation. Figure 8-6 shows scatterplots of decision boundaries from a sequence of three decision trees. At the top, the first split creates a single constant decision boundary. At the middle, a second split, vertical, creates a simple stair step decision boundary. The third tree, at the bottom, the next stair step is formed by a second horizontal constant decision boundary, formed midway up the y axis. The decision boundary is now a nonlinear composition of piecewise constant boundaries.

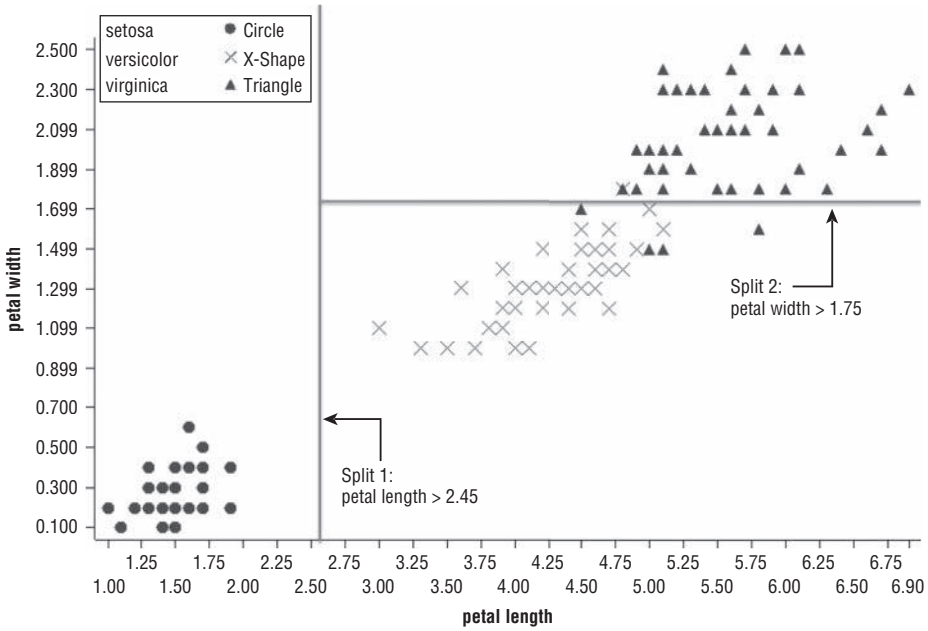

Figure 8-4: First two decision tree splits of iris data

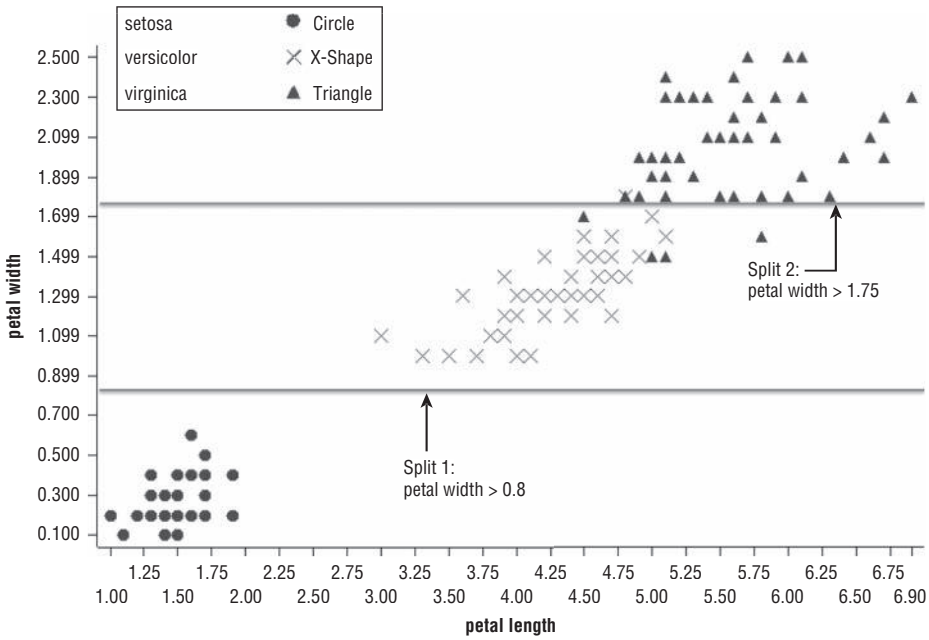

Figure 8-5: Alternative decision tree splits of iris data

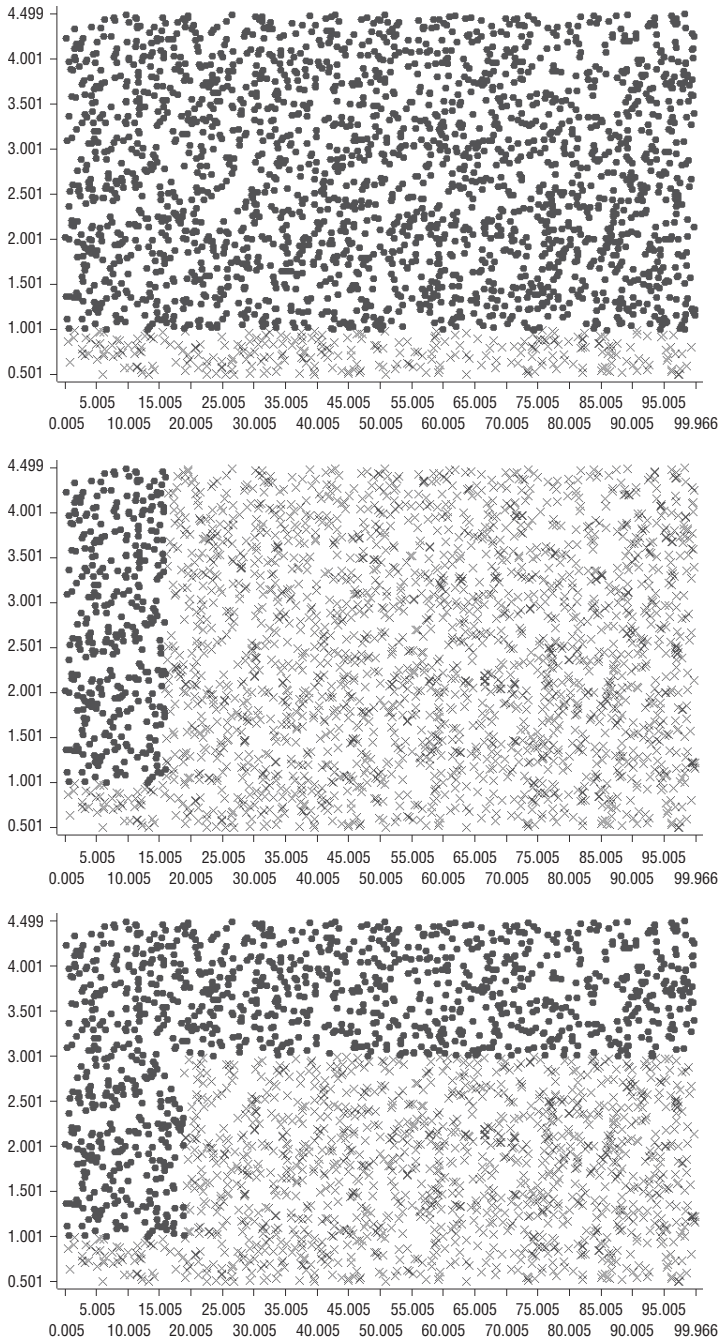

Figure 8-6: Nonlinear decision boundaries for trees

Decision Tree Splitting Metrics

The three most popular decision tree algorithms are CART, C5.0, and CHAID. Table 8-1 provides a brief comparison of these algorithms. Not all implementations

of the algorithms necessarily include all of the options in the table. For example, some implementations of the CART algorithm do not include the ability to change the priors.

The CART and C5.0 decision trees are similar in several regards. They both build trees to full-size, deliberately overfitting the tree and then they prune the tree back to the depth that trades off error with complexity in a suitable manner. Additionally, they both can use continuous and categorical input variables. CHAID, on the other hand, splits only if the split passes a statistically significant test (chi-square). The chi-square test is only applied to categorical variables, so any continuous inputs and output must be binned into categorical variables. Usually, the software bins the continuous variables automatically. You can bin the data yourself as well.

Decision Tree Knobs and Options

Options available in most software for decision trees include:

- **Maximum depth:** The number of levels deep a tree is allowed to go. This option limits the complexity of a tree even if the size of the data would allow for a more complex tree.
- **Minimum number of records in terminal nodes:** This option limits the smallest number of records allowed in a terminal node. If the winning split results in fewer than the minimum number of records in one of the terminal nodes, the split is not made and growth in this branch ceases. This option is a good idea to prevent terminal nodes with so few cases, if you don't have confidence that the split will behave in a stable way on new data. My usual preference is 30, but sometimes make this value as high as several hundred.
- **Minimum number of records in parent node:** Similar to the preceding option, but instead applies the threshold to the splitting node rather than the children of the node. Once there are fewer than the number of records specified by this option, no further split is allowed in this branch. This value is typically made larger than the minimum records in a terminal node, often twice as large. My usual preference is 50.
- **Bonferroni correction:** An option for CHAID only, this correction adjusts the chi-square statistic for multiple comparisons. As the number of levels in the input and target variable increases, and there are more possible combinations of input values to use in predicting the output values, the probability that you may find a good combination increases, and sometimes these combinations may just exist by chance. The Bonferroni correction penalizes the chi-square statistic proportional to the number of possible comparisons that could be made, reducing the likelihood a split will be found by chance. This option is usually turned on by default.

Table 8-1: Characteristics of Three Decision Tree Algorithms

TREE ALGORITHM	SPLITTING CRITERION	INPUT VARIABLES	TARGET VARIABLE	BINARY OR MULTI-WAY SPLITS	COMPLEXITY REGULARIZATION	HANDLING OF CLASS IMBALANCE
CART	Gini index, Two-ing	categorical or continuous	categorical or continuous	Binary	Pruning	Priors or Misclassification costs
C5.0	Gain Ratio, based on entropy	categorical or continuous	categorical	Binary (continuous variables) Multi-way (categorical variables)	Pruning	Misclassification costs
CHAID	chi-square test	categorical	categorical	Binary or Multi-way	Significance test	Misclassification costs

Reweighting Records: Priors

In predictive modeling algorithms, each record has equal weight and therefore contributes the same amount to the building of models and measuring accuracy. Consider, however, a classification problem with 95 percent of the records with the target class labeled as 0 and 5 percent labeled as 1. A modeling algorithm could conclude that the best way to achieve the maximum classification accuracy is to predict every record belongs to class 0, yielding 95 Percent Correct Classification! Decision trees are no exception to this issue, and one can become frustrated when building a decision tree yields no split at all because the great majority of records belong to a single target variable value.

Of course this is rarely what you actually want from a model. When you have an underrepresented target class, it is typical that the value occurring less frequently is far more important than the numbers alone indicate. For example, in fraud detection, the fraud rate may be 0.1 percent, but you can't dismiss fraudulent cases just because they are rare. In fact, it is often the rare events that are of most interest.

One way to communicate to the decision tree algorithm that the underrepresented class is important is to adjust the *prior probabilities*, usually called the *priors*. Most modeling algorithms, by considering every record to have equal weight, implicitly assign prior probabilities based on the training data proportions. In the case with the target class having 5 percent of the target class equal to 1, the prior probability of the target class equaling 1 is 5 percent.

However, what if the training data is either wrong (and you know this to be the case) or misleading? What if you really want the 0s and 1s to be considered *equally* important? The solution to these problems is to change the mathematics that compute impurity so that the records with target variable equal to 1 are weighted equally with those equal to 0. This mathematical up-weighting has the same effect as replicating 1s so that they are equal in number with 0s, but without the extra computation necessary to process these additional records. The tree algorithms that include an option of setting priors include CART and QUEST. For the practitioner, this means that one can build decision trees using the CART and QUEST algorithms when large imbalances exist in the data. In fact, this is a standard, recommended practice when using these algorithms for classification problems.

Reweighting Records: Misclassification Costs

Misclassification costs, as the name implies, provide a way to penalize classification errors asymmetrically. Without misclassification costs, each error made

by the classifier has equal weight and influence in model accuracy statistics. The number of occurrences of a particular value of the target variable has the greatest influence in the relative importance of each target variable value. For example, if 95 percent of the target variable has a value of 0 and only 5 percent has a value of 1, the learning algorithm treats the class value 0 as 19 times more important than the class value 1. This is precisely why a learning algorithm might conclude that the best way to maximize accuracy is to merely label every record as a 0, giving 95 Percent Correct Classification.

Priors provide one way to overcome this bias. Misclassification costs provide a second way to do this and are available as an option in most software implementations of decision trees. For binary classification, if you would like to overcome the 95 percent to 5 percent bias in the data, you could specify a misclassification cost of 19 when the algorithms label a record whose actual value is 1 with a model prediction of 0. With this cost in place, there is no advantage for the algorithm to label all records as 0 or 1; they have equal costs.

Typically, the interface for specifying misclassification costs shows something that appears like a confusion matrix with rows representing actual target variable values and columns the predicted values. Table 8-2 shows an example of a misclassification cost matrix with the cost of falsely dismissing target variable values equal to 1 set to 19. The diagonal doesn't contain any values because these represent correct classification decisions that have no cost associated with them.

Table 8-2: Misclassification Cost Matrix for TARGET_B

COST MATRIX	TARGET = 0	TARGET = 1
Target = 0	—	1
Target = 1	19	—

In the example with 5 percent of the population having the target variable equal to 1 and 95 percent equal to 0, let's assume you have 10,000 records so that 500 examples have 1 for the target, and 9,500 have 0. With misclassification costs set to 19 for false dismissals, if all 500 of these examples were misclassified as 0, the cost would be $19 \times 500 = 9500$, or equal to the cost of classifying all the 0s as 1s. In practice, you don't necessarily have to increase the cost all the way up to 19: Smaller values (even a third of the value, so 6 or 7) may suffice for purposes of building models that are not highly biased toward classifying 0s correctly.

The C5.0 algorithm will not create any splits at all for data with a large imbalance of the target variable, like the example already described with only 5 percent of the target equal to 1; there is not enough Information Gain with any split to justify building a tree. However, setting misclassification costs like the ones in

Table 8-2 communicate to the algorithm that the underrepresented class is more important than the sheer number of examples, and a tree will be built even without resampling the data.

In the case of binary classification, there is no mathematical difference between using priors and misclassification costs to make the relative influence of records with target variable value equal to 1 and those with target variable value equal to 0 the same. Misclassification costs can, however, have a very different effect when the target variable has many levels.

Consider the nasadata dataset, with seven target variable values (alfalfa, clover, corn, soy, rye, oats, and wheat). Priors adjust the relative occurrences of each, as if you replicated records to the point where each class value has the same number of records. Misclassification costs, on the other hand, provide *pairwise* weightings of misclassifications, and therefore there are 21 combinations of costs to set above the diagonal and 21 below, for a total of 42. These costs are not symmetric: If the example has an actual target value equal to alfalfa and the model predicts clover, one may want to set this cost differently than the converse. Table 8-3 shows a full misclassification cost matrix for the nasadata dataset. The actual target variable value is in the rows, and the predicted values in the columns. For example, misclassification costs for the target variable equal to alfalfa but a model predicting the target variable equal to clover are set to 10, whereas the converse is set to the default value of 1. Additionally, wheat misclassified as soy is penalized with a cost of 8.

Table 8-3: Misclassification Cost Matrix for Nasadata Crop Types

COST MATRIX	ALFALFA	CLOVER	CORN	SOY	RYE	OATS	WHEAT
Alfalfa	-	10	1	1	1	1	1
Clover	1	-	1	1	1	1	1
Corn	1	1	-	1	1	1	1
Soy	1	1	1	-	1	1	1
Rye	1	1	1	1	-	1	1
Oats	1	1	1	1	1	-	1
Wheat	1	1	1	8	1	1	-

Misclassification costs, therefore, are tremendously flexible. Figure 8-7 shows a simple decision tree using only the default misclassifications costs: All are equal

to 1. Band11 splits alfalfa and clover into Node 1 to the left, with approximately 50 percent of the cases belonging to each (55 alfalfa and 62 clover records).

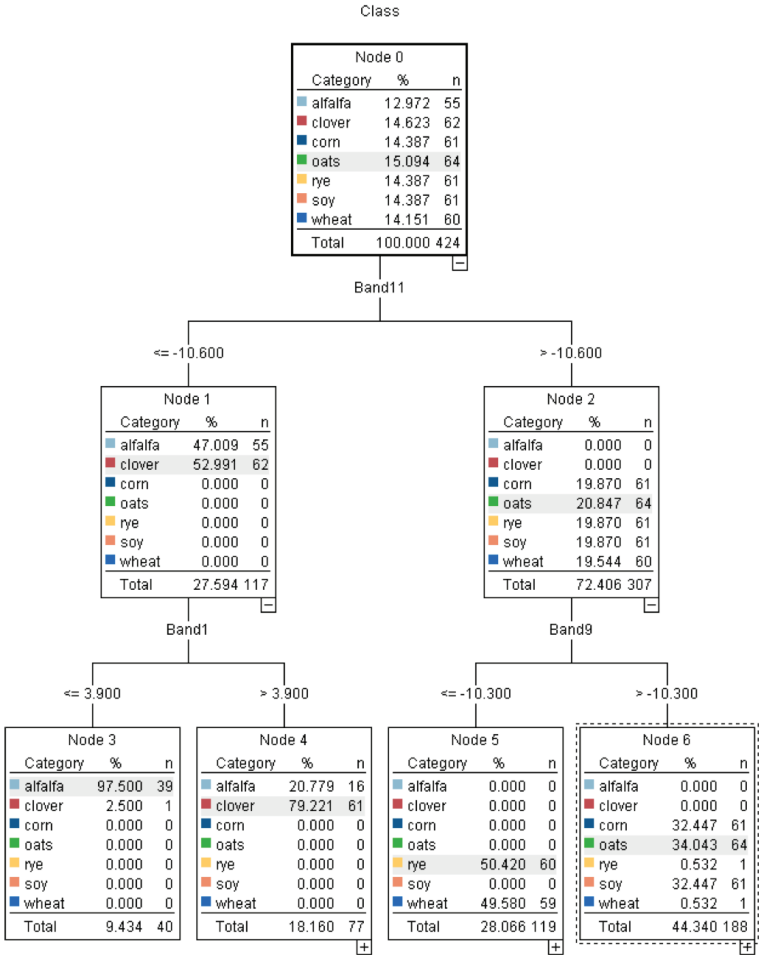

Figure 8-7: Nasadata decision tree with no misclassification costs

Misclassification costs help the classifier change the kinds of errors that are acceptable. A decision tree built with misclassification costs from Table 8-3 is shown in Figure 8-8. The first split no longer groups alfalfa and clover together—the misclassification costs made this split on Band11 too expensive—and splits on Band9 as the root split instead.

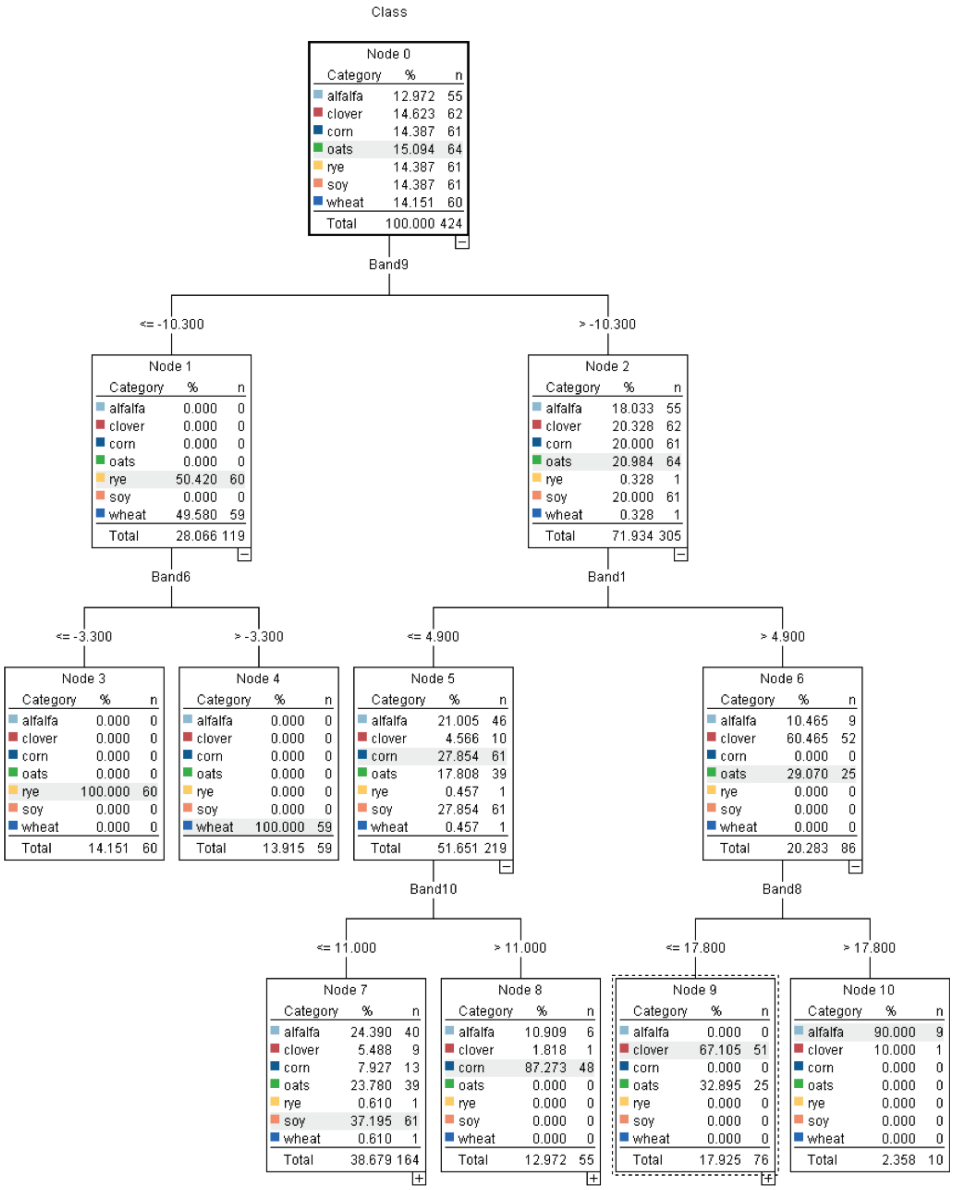

Figure 8-8: Nasadata decision tree with misclassification costs

However, with this flexibility comes difficulty. There are often no known or theoretical values to use for the costs, leaving the modeler with nothing more than a best guess to decide the values to use for misclassification costs.

Some software provides mechanisms to search over ranges of misclassification costs to help determine the optimum values. Nevertheless, misclassification costs can still be valuable for fine-tuning classifiers.

Other Practical Considerations for Decision Trees

Following are some practical considerations to make when using decision trees.

Decision trees are created through a greedy search (forward selection). Therefore, they can never "go back" to revisit a split in a tree based on information learned subsequent to that split. They can be fooled in the short run only to be suboptimal when the tree is completed. Removing the variable in the root split sometimes helps by forcing the tree to find another way to grow. Some software even provides a way to automate this kind of exploratory analysis.

Decision trees incorporate only one variable for each split. If no single variable does a good job of splitting on its own, the tree won't start off well and may never find good multivariate rules. Trees need attributes that provide some lift right away or they may be fooled. If the modeler knows multivariate features that could be useful, they should be included as candidates for the model. As an alternative, other techniques, such as association rules, could be used to find good interaction terms that decision trees may miss. You can then build new derived variables from these interactions and use them as inputs to the decision tree.

Trees are considered weak learners, or unstable models: Small changes in data can produce significant changes in how the tree looks and behaves. Sometimes the differences between the winning split and the first runner-up is very small, such that if you rebuild the tree on a different data partition, the winning/runner-up splits can be reversed. Examining competitors to the winning split and surrogate splits can be very helpful in understanding how valuable the winning splits are, and if other variables may do nearly as well. Trees are biased toward selecting categorical variables with large numbers of levels (high cardinality data). If you find a large number of levels in categorical variables, turn on cardinality penalties or consider binning these variables to have fewer levels. Trees can "run out of data" before finding good models. Because each split reduces how many records remain, subsequent splits are based on fewer and fewer records and therefore have less statistical power.

Single trees are often not as accurate as other algorithms in predictive accuracy on average; in particular, because of greedy, forward variable selection; the piecewise constant splits; and the problem of running out of data. Neural networks and support vector machines will often outperform trees. Consider building ensembles of trees to increase accuracy if model interpretation is not important.

Logistic Regression

Logistic regression is a linear classification technique for binary classification. The history of logistic regression is as follows:

Consider the KDD Cup 1998 data and the question, "Which lapsed donors to a charity can be recovered?" Dozens of giving patterns for each donor were collected and can be used to identify the patterns related to recovering and unrecovered lapsed donors. Two of the historic attributes that were collected were how many gifts a lapsed donor has given during his or her relationship with the charity (NGFITALL) and how much the lapsed donor gave for his or her last gift (LASTGIFT). Figure 8-9 shows how a logistic regression model predicts the likelihood that the donor can be recovered, 1 for recovered and 0 for not recovered. The TARGET_B outcome values predicted by the model (0 or 1) are separated by a line found by the logistic regression model: a *linear decision boundary*. If three inputs were included in the model, the decision boundary changes from a line to a plane that separates the predicted values equal to 0 from the predicted values equal to 1. With more than three inputs, the decision boundary becomes a hyper-plane.

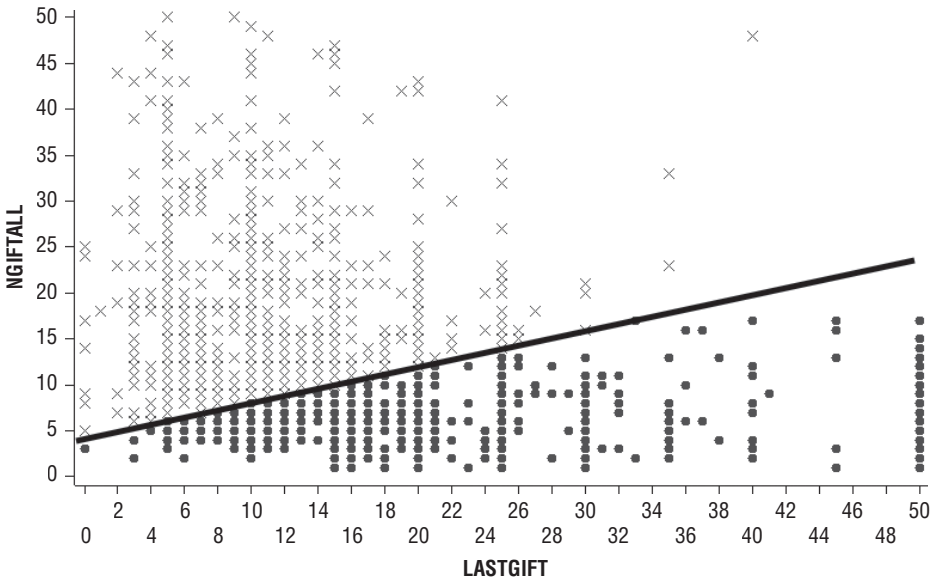

Figure 8-9: Linear decision boundary based on NGIFTALL and LASTGIFT

The core of a logistic regression model is the *odds ratio*: the ratio of the outcome probabilities.

$$odds\,ratio = \frac{P(1)}{1-P(1)} = \frac{P(1)}{P(0)}$$

Bear in mind that the odds ratio is not the same as the likelihood an event will occur. For example, if an event occurs in every 1 of 5 events (20 percent likelihood), the odds ratio is 0.2/(1 – 0.2) = 0.2/0.8 = 0.25. The odds ratio of the event *not* occurring is 0.8/0.2 = 4.0, or in other words, the odds of the event not occurring is 4 times that of it occurring.

Logistic regression does not build linear models of the odds ratio, but rather of the *log* of the odds ratio. Consider the data in Table 8-4. P(0) is the probability that the target variable has value equal to 0 given the input value specified, and P(1) is the comparable measure for a target variable equal to 1.

Table 8-4: Odds Ratio Values for a Logistic Regression Model

INPUT VALUE	P(0)	P(1)	ODDS RATIO	LOG ODDS RATIO
0	0.927	0.073	0.079	–1.103
5	0.935	0.065	0.07	–1.156
10	0.942	0.058	0.062	–1.208
15	0.948	0.052	0.055	–1.26
20	0.954	0.046	0.049	–1.313
25	0.959	0.041	0.043	–1.365
30	0.963	0.037	0.038	–1.417
35	0.967	0.033	0.034	–1.47

If you plot the input value versus the odds ratio (see Figure 8-10), you can see that the relationship between the input and the odds ratio is nonlinear (exponential). The same is true if you plot P(1) versus the input. This flattens out into a linear relationship when you compute the log of the odds ratio (see Figure 8-11).

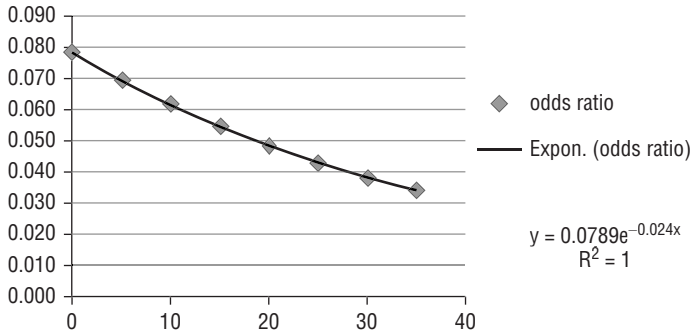

Figure 8-10: Odds ratio versus model input

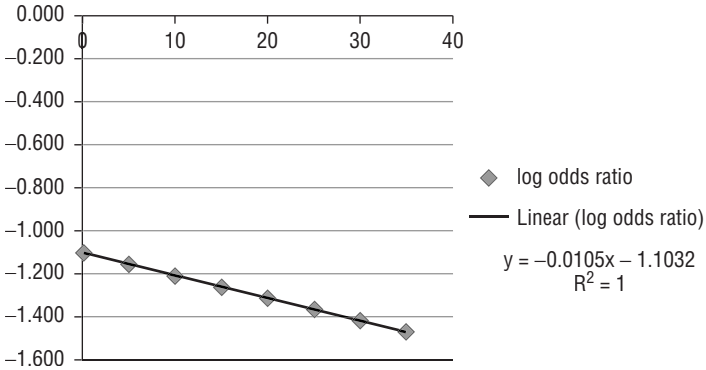

Figure 8-11: Log odds ratio versus model input

This linear relationship *is* the statistical model logistic regression in computing:

$$odds\,ratio = \frac{P(1)}{1-P(1)} = w_0 + w_1 \times x_1 + \ldots + w_n \times x_n$$

In the odds ratio equation, the values w_0, w_1, and so forth are the model coefficients or weights. The coefficient w_0 is the constant term in the model, sometimes called the bias term. The variables x_0, x_1, and so forth are the inputs to the model. For example, x_1 may be LASTGIFT and x_2 may be NGFITALL.

Without providing any derivation, the calculation of the probability that the outcome is equal to 1 is:

$$\Pr(target = 1) = \frac{1}{1 + e^{-(w_0 + w_1 \times x_1 + w_2 \times x_2 + \ldots + w_n \times x_n)}}$$

The probability formula is the function known as the *logistic curve* and takes on the shape shown in Figure 8-12. In contrast to the linear regression model whose predicted values are unbounded, the logistic curve is bounded by the range 0 to 1. The middle of the distribution, approximately in the x axis range –2 to 2 in the figure, is approximately linear. However, the edges have severe nonlinearities to smoothly scale the probabilities to the bounded range. When you change the input value from 0 to +2, the probability goes from 0.5 to 0.88, whereas when you change the input from +4 to +6, the probability changes only from 0.98 to 0.998.

How is the probability calculation, with values scaled nonlinearly so that it has a range between 0 and 1, and the linear decision boundary? Assume that the prior probability of the target variable, TARGET_B, is 50 percent. All of the data points that lie on the line that separates the predicted TARGET_B values equal to 0 from those equal to 1 have predicted probability values, P(1), equal to 0.5.

The further from the line a data point resides, the larger the value of P(1) or P(0). For a data point in the upper left of Figure 8-9, LASTGIFT = 5 and NGIFTALL = 70, the P(1) is 0.82: a high likelihood the donor can be recovered, and a data point far from the decision boundary. The further from the decision boundary, the smaller the relative increase in P(1) until it approaches 1.0.

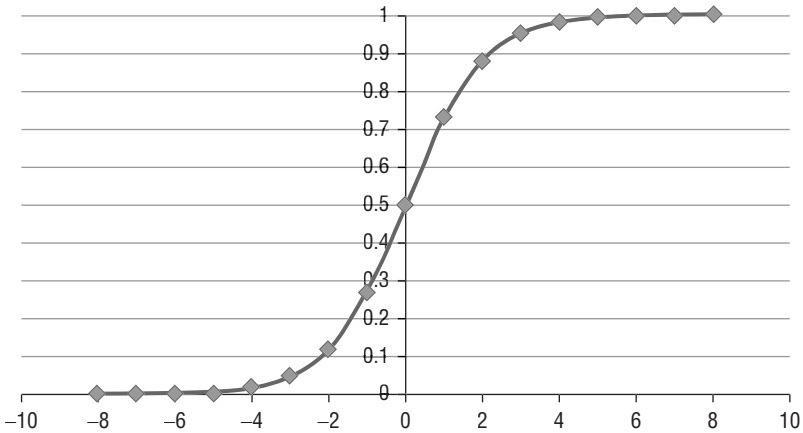

Figure 8-12: Logistic curve

Interpreting Logistic Regression Models

To interpret logistic regression models, you'll examine several numbers most software packages provide in their summary reports:

- **Coefficient:** Sometimes shown as the Greek symbol beta (ß). This is the parameter found by the logistic regression algorithm; one coefficient per input variable plus one constant (bias term).
- **Standard error of the coefficient (*SE*):** A measure of certainty of the coefficient found by logistic regression. Smaller values of standard error imply a smaller level of uncertainty of the coefficient value.
- **Confidence Interval (CI):** The range of values the coefficient is expected to fall between. The CI is computed as:

$$CI = \beta \pm SE$$

This is sometimes helpful when evaluating models. You can compute the Pr(1) not just for the coefficient with value ß, but also for the coefficient plus the *SE* and minus the *SE*, thereby computing the sensitivity of the logistic regression probabilities.

- **z statistic or Wald test:** z is calculated by the equation:

$$z = \frac{\beta}{SE}$$

  The larger the value of z, the more likely the term associated with the coefficient is significant in the model.

- **Pr(>|z|):** The p value associated with the z statistic. Often, analysts consider values less than 0.05 to indicate a significant predictor, although there is no theoretical reason for making this inference; it is historical precedence. The p value does not indicate how much accuracy the variable provides, *per se*, but rather how likely it is that the coefficient is different from 0. The more records one has in the training data, the smaller SE and the p value will be. Therefore, the more records you have, the more stable the coefficients become.

Some implementations of logistic regression include variable selection options, usually forward selection or backward elimination of variables. The p value is often the metric used to decide to include or exclude a variable, by including new variables only if the p value is less than some value (typically 0.05), or excluding a variable if its p value is greater than some value (often 0.1). These thresholds are usually configurable.

If variable selection options are not available, the practitioner can use the p values to decide which variables should be excluded. For example, if you use logistic regression to build a model to predict lapsed donors with target variable TARGET_B, and include inputs NGIFTALL, LASTGIFT, FISTDATE, RFA_2F, RFA_2A, and AGE, the following model was created, with coefficients and summary statistics shown in Table 8-5.

Table 8-5: Logistic Regression Model Report

VARIABLE	COEFF.	STD. ERR.	Z-SCORE	P>|Z|
NGIFTALL	0.0031	0.0024	1.2779	0.2013
LASTGIFT	0.0008	0.0014	0.5702	0.5685
FISTDATE	−0.0003	0.0000664	−3.9835	0.0000679
RFA_2F=2	0.2686	0.0405	6.636	3.22E−11
RFA_2F=3	0.3981	0.047	8.4638	0
RFA_2F=4	0.5814	0.0538	10.7987	0
RFA_2A=E	−0.1924	0.0535	−3.5937	0.0003
RFA_2A=F	−0.3558	0.0613	−5.8085	6.31E−09

VARIABLE	COEFF.	STD. ERR.	Z-SCORE	P>|Z|
RFA_2A=G	–0.5969	0.075	–7.9642	1.89E–15
AGE	–0.0014	0.0011	–1.3129	0.1892
Constant	–0.381	0.6351	–0.5999	0.5486

The worst predictor according to the *p* value is LASTGIFT, and therefore this variable is a good candidate for removal. This doesn't mean that LASTGIFT isn't a good predictor by itself, only that in combination with the other inputs, it is either relatively worse, or the information in LASTGIFT also exists in other variables rendering LASTGIFT unnecessary to include in the model. After removing LASTGIFT, the model is rebuilt, resulting in the model coefficient values and summary statistics shown in Table 8-6.

Table 8-6: Logistic Regression Model after Removing LASTGIFT

VARIABLE	COEFF.	STD. ERR.	Z-SCORE	P>|Z|
NGIFTALL	0.0031	0.0024	1.2772	0.2015
FISTDATE	–0.0003	0.0000664	–3.9834	0.0000679
RFA_2F=2	0.2671	0.0404	6.6136	3.75E–11
RFA_2F=3	0.396	0.0469	8.4465	0
RFA_2F=4	0.5787	0.0536	10.7924	0
RFA_2A=E	–0.1898	0.0534	–3.558	0.0004
RFA_2A=F	–0.349	0.0601	–5.8092	6.28E–09
RFA_2A=G	–0.5783	0.0673	–8.5932	0
AGE	–0.0014	0.0011	–1.3128	0.1893
Constant	–0.3743	0.635	–0.5894	0.5556

Now NGIFTALL and AGE are the only remaining variables with *p* values greater than 0.05. You can continue the process until all the variables have *p* values greater than 0.05 if the goal is to identify variables that are all statistically significant at the 0.05 level.

However, it is possible that variables with *p* values above the 0.05 significance level still provide predictive accuracy. If accuracy is the objective, you should use accuracy to decide which variables to include or exclude.

Other Practical Considerations for Logistic Regression

In addition to the principles of building logistic regression models described so far, successful modelers often include other data preparation steps.

The following four considerations are among the most common I apply during modeling projects.

Interactions

Logistic regression is a "main effect" model: The form of the model described so far has been a linearly weighted sum of inputs with no interaction terms. Sometimes, however, it is the interaction of two variables that is necessary for accurate classification.

Consider a simple example in Figure 8-13: The cross data set created from two inputs, x and y, and a binary target variable z represented by X-shapes (target variable = +1) and circles (target variable = –1). This example is just an extension of the famous "XOR" problem cited by Minsky and Pappert in their book *Perceptrons* (MIT, 1969). Clearly there is no linear separator that classifies this dataset.

A logistic regression model using only a linear combination of inputs x and y—a main effect model—only classifies approximately 50 percent of the cases correctly. The predicted values from the logistic regression model are shown in Figure 8-14—clearly not an effective classifier.

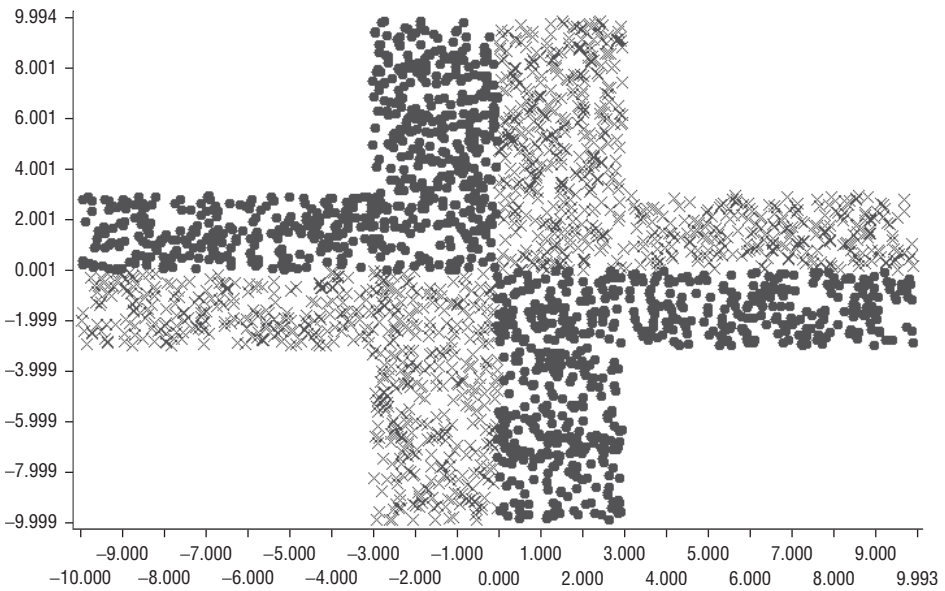

Figure 8-13: Cross data

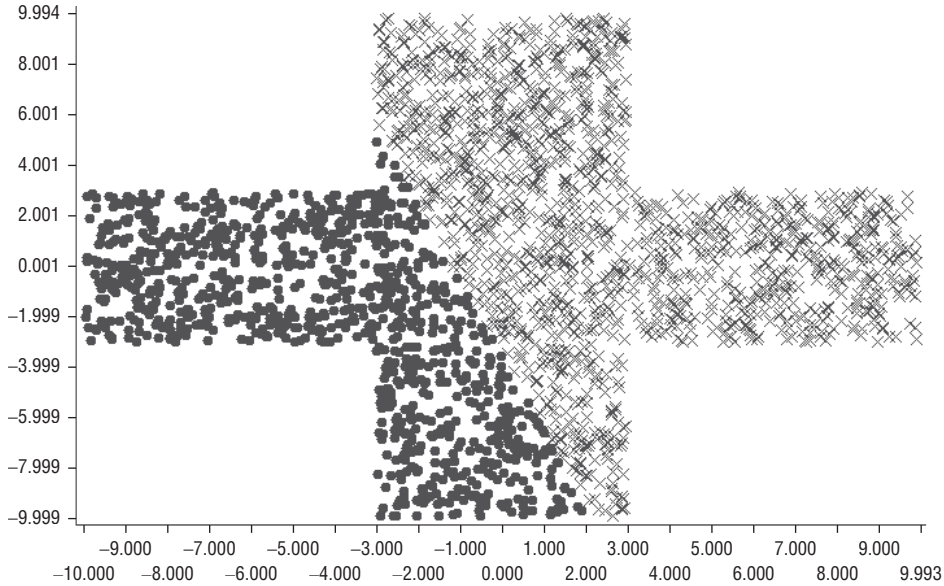

Figure 8-14: Linear separation of cross data

Consider adding an interaction term to the model: $x \times y$. A histogram of this new feature, shown in Figure 8-15, shows the value of this interaction: The model that was able to do no better than a "random" guess using x and y by themselves has become, with an interaction term, an excellent model.

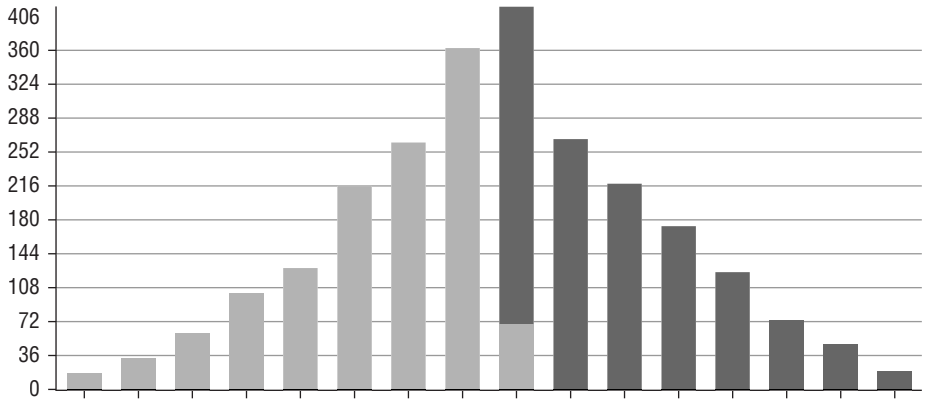

Figure 8-15: Histogram of interaction variable

Missing Values

Logistic regression cannot have any missing values. How missing values are handled is implementation dependent, although often the course of action in software is listwise deletion. All of the typical missing value imputation methods apply.

Dummy Variables

Logistic regression is a numeric algorithm: All inputs must be numeric. Categorical variables must therefore be transformed into a numeric format, the most common of which is 1/0 dummy variables: one column for each value in the variable minus one. If a categorical variable has eight levels, create seven dummy variables only; the contribution for the eighth will be taken up by the bias (constant) term. If all values are represented in the dummy variable, the multicollinearity in the data will cause the terms combing from the same target variable to have *z* values that fail significance tests.

Consider a simple logistic regression model with one categorical input having four levels: A, B, C, and D. Dummy variables are created for each level so that there are now four candidate inputs to the logistic regression model. If all four are included in the model, you may see bizarre standard error values like those in Table 8-7. The *p* values equal to 0.9999 make these four variables seem unrelated to the target. Note that even the constant term is problematic.

Table 8-7: Logistic Regression Model with *n* Dummy Variables

VARIABLE	BETA	SE	WALD TEST, Z	P>|Z|
X1=D	2.38	28,630.90	8.31E–05	0.9999
X1=E	2.80	28,630.90	9.78E–05	0.9999
X1=F	3.24	28,630.90	0.0001	0.9999
X1=G	3.49	28,630.90	0.0001	0.9999
Constant	–0.1471	28,630.90	–5.14E–06	1

The clue, however, is that the *SE* values are so large. This indicates a numerical degeneracy in their calculation. The general principle is this: For categorical variables with *n* levels, only include *n* –1 dummy variables as inputs to a logistic regression model. If you remove one of the dummy variables, such as

X1=G, the problem goes away and the model looks fine (Table 8-8). It doesn't matter which dummy variable is removed; you can choose to remove the dummy that is the least interesting to report.

Table 8-8: Logistic Regression Model with *n*–1 Dummy Variable

VARIABLE	BETA	SE	WALD TEST, Z	P>|Z|
X1=D	–1.1062	0.08	–14.0	0
X1=E	–0.6865	0.07	–10.01	0
X1=F	–0.2438	0.06	–3.7863	0.0002
Constant	3.3387	0.0558	59.7835	0

Multi-Class Classification

If the target variable has more than two levels, logistic regression cannot be used in the form described here. However, there are two primary ways practitioners extend the logistic regression classifier.

The most common way is to explode the multi-class variable with N levels into N –1 dummy variables, and build a single logistic regression model for each of these dummy variables. Each variable, as a binary variable, represents a variable for building a "one vs. all" classifier, meaning that a model built from each of these dummy variables will predict the likelihood a record belongs to a particular level of the target variable in contrast to belonging to *any* of the other levels. While you could build N classifiers, modelers usually build N–1 classifiers and the prediction for the Nth level is merely one minus the sum of all the other probabilities:

$$P(Target = N) = \sum_{i=1}^{N-1} P(Target = i)$$

When deploying models using this approach, N–1 model scores are generated and a seventh score is computed from the other six. For the nasadata dataset, six models are created which generate six probabilities. The seventh probability is one minus the sum of the other six. Typically, the maximum probability from the seven classes is considered the winner, although other post-processing steps can be incorporated as well to calculate the winning class.

The benefit of this approach is that the number of classifiers scales linearly with the number of levels in the target variable, and the interpretation of the resulting models is straightforward. However, this approach groups all the remaining target levels into a single value, 0, regardless of whether they are related to one another or not, and therefore the model may suffer in accuracy as a result.

An alternative that is used sometimes is to build a model for every pair of target variable levels: the *pairwise classifier* approach. This approach requires sampling records to find just those that apply to each pair of levels a classifier is built for. For that levels, this requires building $N \times (N–1)/2$ classifiers. For example, the nasadata contains 7 classes, and therefore building a classifier for each pair of classes requires building 21 classifiers ($(7 \times 6)/2$).

It is certainly true that the pairwise approach requires building many more models than the one-vs.-all approach; it isn't necessarily the case that computationally it is more expensive. The nasadata has 424 records, but only approximately 60 per class. Every classifier therefore is built on only 120 records rather than the full dataset, so the computation burden depends also on the efficiency of the algorithm.

Deploying the pairwise set of classifiers requires that a new score (a column) be created for every classifier. Each level will have $N–1$ classifiers with its level involved. For the nasadata example, you have 21 classifiers and each target variable level is involved in 6 of them. You can decide on the winner through a number of approaches, including averaging the probabilities for each level (6 each in the nasadata example) or counting votes where each level has the larger probability.

Beware of implementations that automatically handle multilevel target variables by creating dummies. Some implementations use the same model structure for each one-vs-all classifier even if some variables are not good predictors for a particular one-vs-all model.

Neural Networks

Artificial Neural Networks (ANN) began as linear models in 1943 with the introduction of a model that emulated the behavior of a neuron with McCulloch and Pitts. These were linear models with a threshold, although they were not introduced primarily for their biological analogy but rather for their computational flexibility. In the 1950s and 60s, there was a flurry of activity surrounding these ideas, primarily using single nodes, although some began experimenting

with multiple layers of nodes, including Frank Rosenblatt, Bernie Widrow, Roger Barron, and others. However, ANNs were primarily known by the single neuron in the early days, either the perceptron as designed by Rosenblatt or an ADALINE (Adaptive Linear Neuron) by Widrow.

The momentum gained in neural network research during the 1950s and 60s hit a wall in 1969 with the publication of the book *Perceptrons* (The MIT Press) by Marvin Minsky and Seymour Pappert, which proved that the perceptron could not predict even elementary logic functions such as the exclusive or (XOR) function. While this is true, a second hidden layer in a neural network can easily solve the problem, but the flexibility of 2-layer neural networks wasn't able to break through the impression gained from the Minsky and Pappert book about the deficiencies of perceptrons. Neural networks remained in the background until the 1980s.

The resurgence of neural networks in the 1980s was due to the research in Parallel Distributed Processing (PDP) systems by David Rumelhart, James Mclelland, and others. It was in 1986 that the backpropagation algorithm was published. Backpropagation enabled the learning of the neural network weights of a multilayer network of elements with continuous (initially sigmoidal) activation functions. It was designed to be simple from the start, or in the words of Rumelhart, "We'll just pretend like it [the neural network] is linear, and figure out how to train it as if it were linear, and then we'll put in these sigmoids." Another significant development was the increase in speed of computers so they could handle the computational load of neural networks. This allowed researchers to begin experimenting with neural networks on a wide range of applications. The first international neural network conference of the Institute of Electrical and Electronics Engineers (IEEE) was held in 1987, giving further credibility to neural networks as an accepted approach to data analysis.

In 1989, George Cybenko proved that a sigmoidal neural network with one hidden layer can approximate any function under some mild constraints. This flexibility of the neural network to adapt to any reasonable function became a powerful rallying cry for practitioners, and by the 1990s, neural networks were being widely implemented in software both in standalone, neural network–only software packages as well as options in data mining software that contained a suite of algorithms.

ANNs are a broad set of algorithms with great variety in how they are trained, and include networks that are only feedforward as well as those with feedbacks. Most often, when the phrase "neural network" is used in predictive analytics, the intent is a specific neural network called the mutli-layer perceptron (MLP). The neural networks described in this section therefore are limited to MLPs.

Building Blocks: The Neuron

The perceptron is an ANN comprised of a single neuron. In an MLP, neurons are organized in layers in a fully connected, feedforward network. Each neuron is simply a linear equation just like linear regression as shown in the following equation. Figure 8-16 shows a representation of a single neuron. Just as with the logistic regression model, the values *w0*, *w1*, and so forth are the weights, and the values *x0*, *x1*, and so forth are the inputs to the neural network.

$$y_i = w_0 + w_1 \times x_1 + w_2 \times x_2 + \ldots + w_n \times x_n$$

This equation is often called the *transfer function* in an ANN. In the early days of neural networks, in the 1960s, this linearly weighted sum would be thresholded at some value so that the output of the neuron would be either 1 or 0. The innovation in the 1980s was the introduction of a soft-thresholding approach by means of an *activation function*.

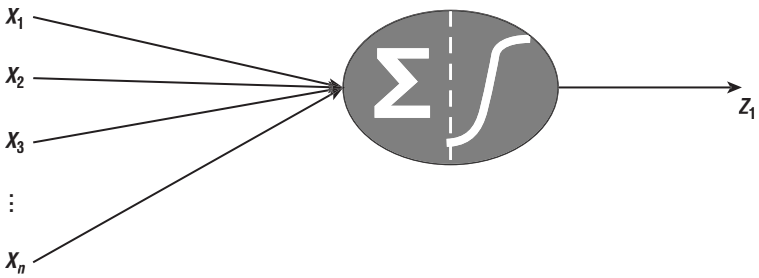

Figure 8-16: Single neuron

The neuron contains one other element after the linear function. The output of the linear function is then transformed by the activation function, often called a *squashing function*, that transforms the linear output to a number between 0 and 1. The most common squashing function is the sigmoid, which is the same function as the logistic regression logistic curve:

$$z_1 = logstic(y) = \frac{1}{1 - e^{-y}}$$

Other functions used for the squashing function include the hyperbolic tangent (tanh) and arctangent (arctan). Determining which activation function to use is not usually very important: Any of them suffice for the purpose of transforming the transfer function. One difference between these activation functions is that the logistic curve ranges from 0 to 1, whereas tanh and arctan range from –1 to +1.

$$z_1 = \tanh(y) = \frac{e^{y} - e^{-y}}{e^{y} + e^{-y}}$$

The key is that the activation functions are continuous and nonlinear. Because they are continuous, you can compute derivatives, an essential property of the neurons so that the learning algorithms used to train neural networks can be applied. Neurons, except for output layer neurons for reasons to be described later, must be nonlinear for the neural network to be able to estimate nonlinear functions and nonlinear relationships in the data. It is precisely the nonlinear aspect of the neuron that gives the neural network predictive power and flexibility.

A single neuron builds a linear model: a linear decision boundary for classification or a linear estimator for continuous-valued prediction. Such models are not particularly interesting because there are many linear modeling algorithms that are as accurate as the single neuron but are algorithmically far more efficient. However, when layers are added to the ANN, they become far more flexible and powerful as predictors.

A layer is a set of neurons with common inputs, as shown in Figure 8-17. The neurons, in general, will have different coefficients, all learned by the training algorithms used by the ANN.

Some layers in the ANN are called *hidden layers* because their outputs are hidden to the user. In the network shown in Figure 8-18, there are two hidden layers in the ANN. This network also has an output layer: one for every target variable in the data. If you build classification models for the nasadata, each of the seven target variable classes will have its own output layer neuron. Note that this is in contrast to logistic regression models where completely separate models for each target variable value are built.

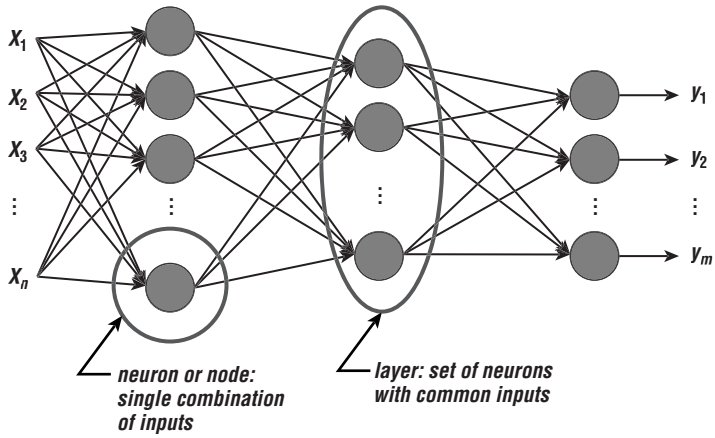

Figure 8-17: Neural network terminology, part 1

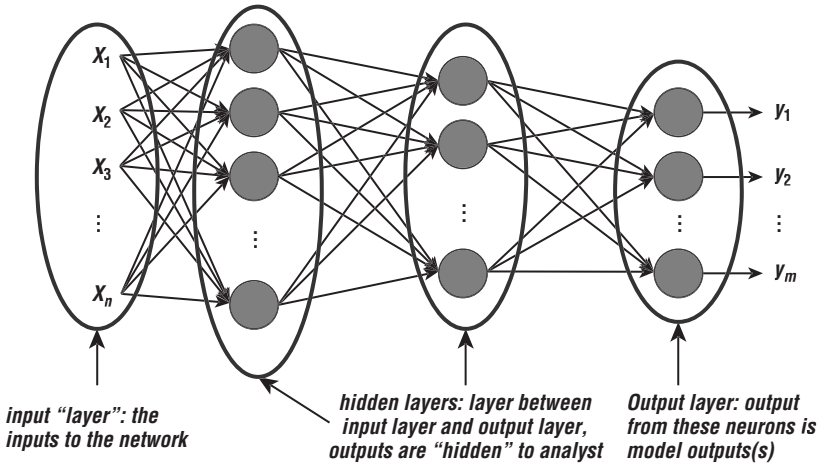

Figure 8-18: Neural network terminology, part 2

For classification modeling, the output layer neurons usually have the same sigmoid activation function already described, one that transforms the neuron to a value between 0 and 1. For continuous-valued prediction, however, software implementations of ANNs most often use what is called a "linear" activation function, which means that no transformation is applied to the linearly weighted sum transfer function. For target variables that are unbounded, you can also use an exponential activation function in the output layer neurons.

The cost function for a neural network predicting a continuous-valued output minimizes squared error just like linear regression. In fact, without the squashing functions for each neuron in a neural network, the network would just be a large, inefficient, linear regression model. The nonlinear squashing functions enable the network to find very complex relationships between the inputs and output without any intervention by the user.

For classification, many predictive analytics software packages use cross-entropy as the cost function rather than minimizing squared error because it seems to be a more appropriate measure of error for a dichotomous output classification problem. It has been shown, however, that even a squared-error cost can find solutions every bit as good as cross-entropy for classification problems.

A neural network for classification with no hidden layer and a single output neuron is essentially a logistic regression model when the neuron has a logistic activation function, especially if the cost function is cross-entropy. With the squared error cost function, there is no guarantee the model will be the same.

Neural Network Training

ANNs are iterative learners, in contrast to algorithms like linear regression, which learn the coefficients in a single processing step. The learning process is similar to how I learned to catch fly balls as a boy. First, imagine my father

hitting a fly ball to me in the outfield. In the beginning, I had absolutely no idea where the ball was going to land, so I just watched it until it landed, far to my left. Since my goal was to catch the ball, the distance between where the ball landed and where I stood was the error. Then my father hit a second fly ball, and, because of the prior example I had seen, I moved (hesitantly) toward where the ball landed last time, but this time the ball landed to my right. Wrong again.

But then something began to happen. The more fly balls my father hit, the more I was able to associate the speed the ball was hit, the steepness of the hit, and the left/right angle—the initial conditions of the hit—and predict where the ball was going to land. Major league outfielders are so good at this that for many fly balls, they arrive at the spot well before the ball arrives and just wait.

This is exactly what neural networks do. First, all the weights in the ANN are initialized to small random values to "kick start" the training. Then, a single record is passed through the network, with all the multiplies, adds, squashing computed all the way through neurons in hidden layers and the output layer, yielding a prediction from each output neuron in the neural network. The error is computed between the actual target value and the prediction for that record. Weights are adjusted in the ANN proportional to the error. The process for computing the weights for one output neuron is described here, but the same principles apply for more than one output neuron.

The process is repeated next for the second record, including the forward calculations ending in predictions from the output layer neuron(s) and error calculations, and then weights are updated by backpropagation of the errors. This continues until the entire set of records has passed through the ANN. Figure 8-19 is a representation of the process where each jump is the result of a single record passing through the network and weights being updated. The representation here shows a network converging to the minimum error solution if the error surface is a parabola, as is the case when minimizing squared errors for a linear model such as linear regression. The actual error surface is much more complex than this.

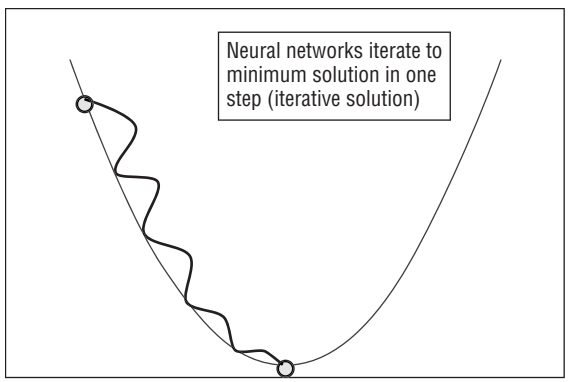

Figure 8-19: Iterative convergence to the bottom of a quadratic error curve

An *epoch* or *pass* occurs when every record in the training data has passed through the neural network and weights have been updated. Training a neural network can take dozens, hundreds, or even thousands of training epochs; you can understand why ANNs have a reputation for taking a long time to train. The training time on commonly available computer hardware is acceptable for most problems.

The forward and backward process continues until a stop condition applies to end training. The difficulty in converging to a solution is non-trivial because of the non-linearity of neural networks. Rather than the solution following the smooth quadratic error surface shown in Figure 8-19, it is more like the conceptual error curve shown in Figure 8-20 with many dips and hills. These dips are local minima. The challenge for training a neural network is to traverse the error surface at just the right pace.

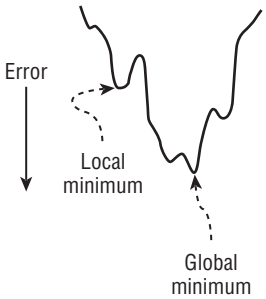

Figure 8-20: Local and global minima

If the algorithm takes too small of a step, it can easily end up in a local minimum, as shown on the left in Figure 8-21. A larger learning rate is sufficient to jump over the local minimum on the right, enabling the algorithm to find the global minimum.

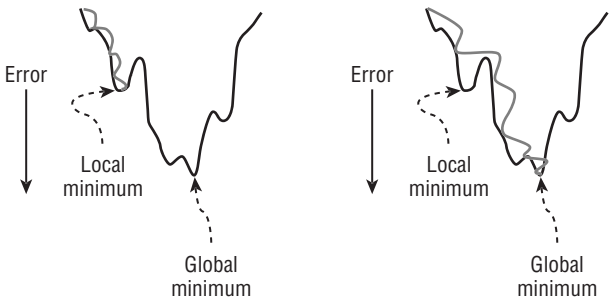

Figure 8-21: Iterating to local and global minima

You never know if the global minimum has been found because you never know if all of the points of the error surface have been found. In the left sequence of errors of Figure 8-22, you have found a minimum. You only know it is a local minimum if you build a second neural network and find a better solution, such as the sequence of errors on the right of Figure 8-22. Is this minimum a global minimum? You can never know for sure.

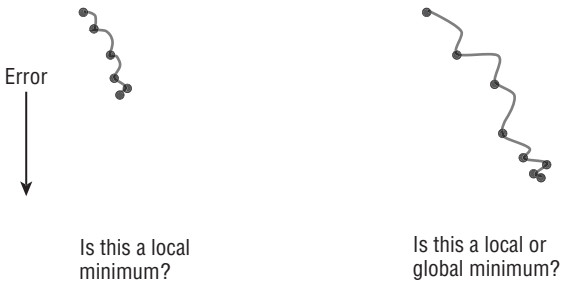

Figure 8-22: Local or global minimum?

Experimentation therefore is key with neural networks. Modelers typically build several networks with varying parameters, which include varying how many neurons to include in hidden layers, learning rates, even changing the random number seed that generates the initial weights for the network.

The Flexibility of Neural Networks

As universal approximators, neural networks are very flexible predictors. One example is the ability of the ANN to predict the sombrero function without having to create derived variables to unravel the nonlinearities in the data. The sombrero function is defined by the equation:

$$z = \frac{\sin(sqrt(x^2 + y^2))}{sqrt(x^2 + y^2)}$$

and has two classes, represented by 1 (X-shapes) and 0 (circles). The (X-shapes) class is characterized by those data points with $z > 0$ and the (circles) class with $z <= 0$. Figure 8-23 shows the function.

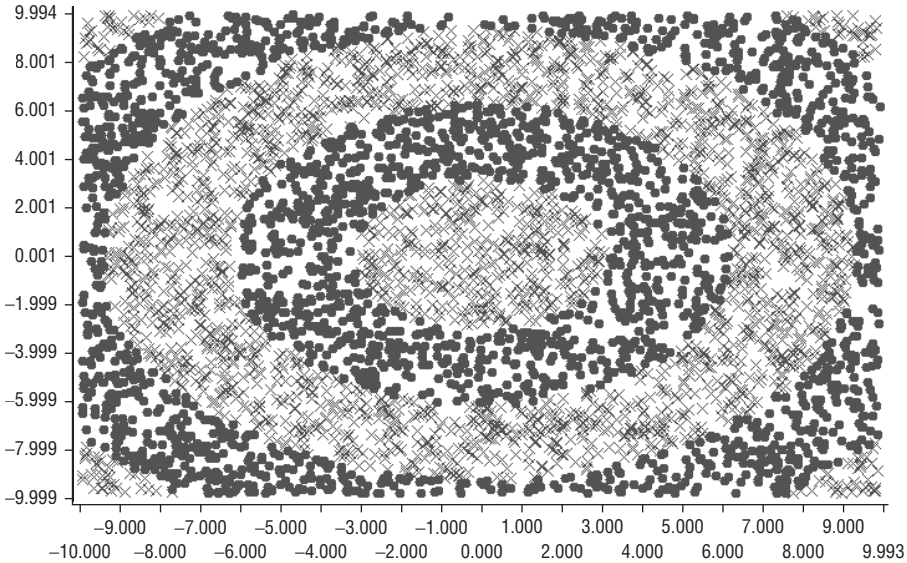

Figure 8-23: Sombrero function

This function is clearly not linearly separable; any line drawn will necessarily contain both classes on either side of the line (see Figure 8-24). This function cannot, therefore, be estimated by logistic regression models without transforming the inputs to "linearize" the relationship between inputs and the output. Likewise, decision trees will have great difficulty with this function because of the smooth curves throughout the decision boundaries between the values.

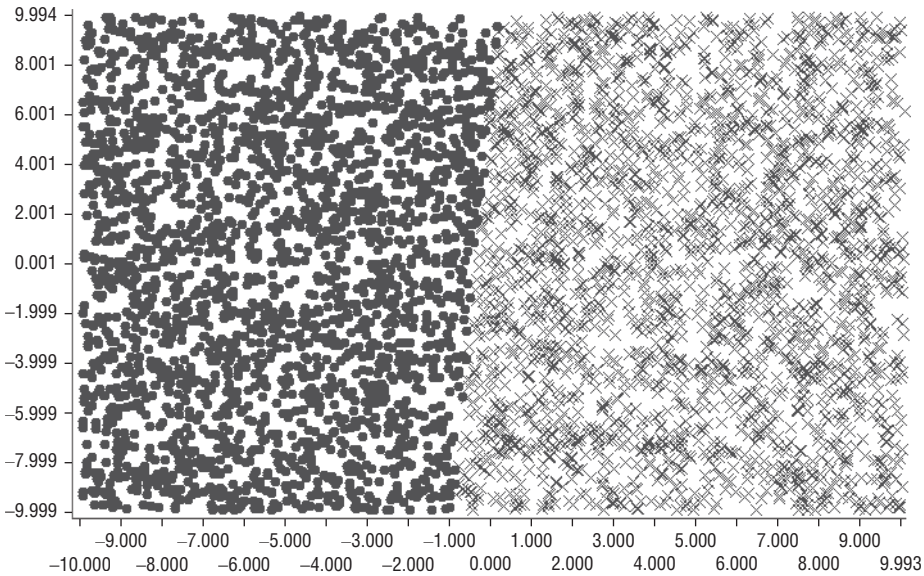

Figure 8-24: Linear classifier for sombrero function

An MLP with two hidden layers containing twelve hidden units per hidden layer was trained on this data. As training of the neural network progresses through 10, 100, 200, 500, 1000 and 5000 epochs, you see in Figure 8-25 the true shape of the sombrero function forming. With only 10 or 100 epochs, the sombrero pattern has not yet been learned. The gist of the sombrero function is seen after 500 epochs, and after 5000 epochs, the sombrero shape is clear.

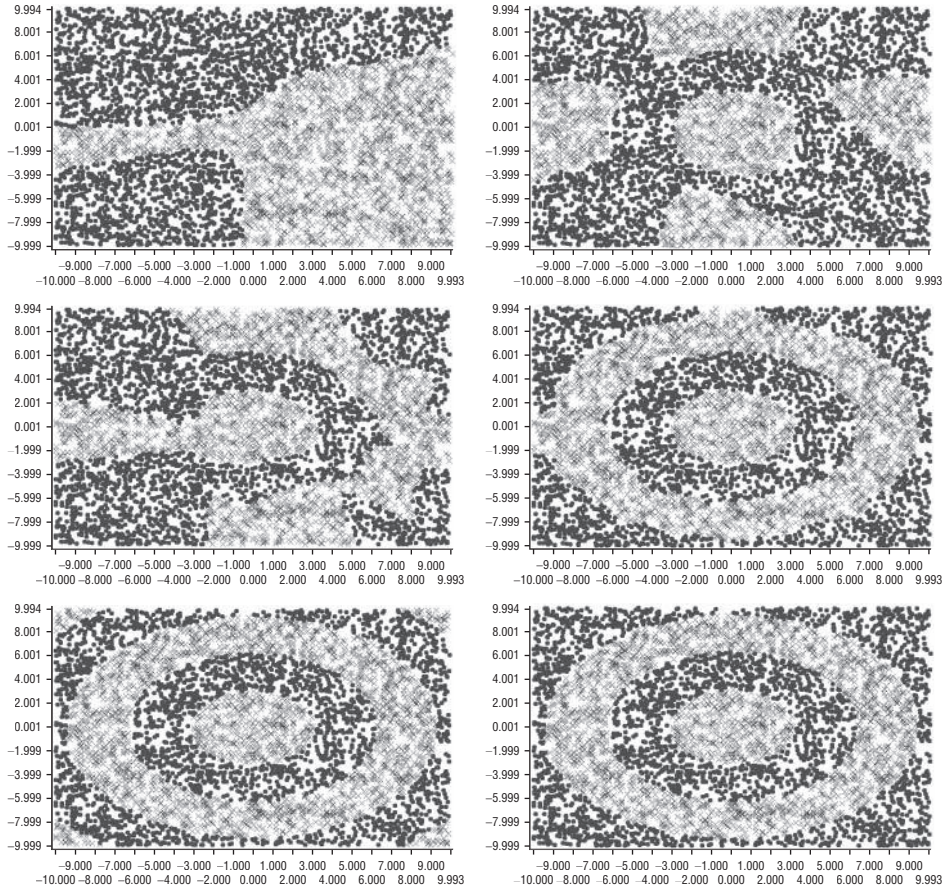

Figure 8-25: Decision boundaries as neural network learns after 10, 100, 200, 500, 1000, and 5000 epochs

Neural Network Settings

ANNs have several options available in software for the building of the neural networks:

- **Learning rate (for MLPs only):** Adjust the amount of change in weights per iteration. Larger learning rates help the MLP to train faster by making larger jumps. In addition, larger learning rates help the MLP guard against

converging to a local minimum. However, a learning rate that is too large could allow the MLP to jump over a global minimum. The solution some software uses to the dilemma of learning rate size is to schedule the learning rate so it decreases as the number of epochs increases. Good values for learning rates range from 0.01 to 0.5. The value 0.1 is a good starting value.

- **Learning rate (for MLPs only):** Adjust the amount of change in weights per iteration. Larger learning rates help the MLP to train faster by making larger jumps. In addition, larger learning rates help the MLP guard against converging to a local minimum. However, a learning rate that is too large could allow the MLP to jump over a global minimum. The solution some software uses to the dilemma of learning rate size is to schedule the learning rate so it decreases as the number of epochs increases. Good values for learning rates range from 0.01 to 0.5. The value 0.1 is a good starting value.
- **Momentum (for MLPs only):** Helps speed up convergence with backprop by smoothing weight changes by adding a fraction of the prior weight change to the current rate. It also has the effect, if the prior weight change and current weight changes have the same sign, of amplifying the magnitude of the weight adjustment. Good values of momentum range from 0.3 to 0.9, with the larger value.
- **Number iterations/epochs:** The maximum number of passes through the data for training the network. This is included for time savings and in some cases to guard against overfitting the network.
- **Number of hidden layers and number of neurons in each hidden layer:** This specifies the neural network architecture. While theoretically only one hidden layer is necessary, in practice more complex problems can benefit from adding a second hidden layer to the network. No theory exists to specify exactly how many neurons are included in a neural network, although more neurons are needed for noisier data and more complex data. Many software packages now provide the ability to search through a wide range of architectures so the modeler doesn't have to iterate manually.
- **Stopping criteria:** Some implementations of neural networks allow stopping once the training accuracy reaches an acceptable level so that overfitting the data is avoided. Some implementations allow training for a pre-specified number of minutes. Another option available in some software is to stop training when the error on testing data increases for many consecutive epochs, indicating that overfitting has begun in the network.
- **Weight updates:** Some implementations of neural networks also provide a setting for how often the weights are updated. The default is that weights are updated after every single record passes through the network. However, 100 or more records can be accumulated first, and the average error on these 100 records is computed to update the weights. This approach will speed up training considerably but at the expense of reducing the influence of some records that might produce large errors on their own.

The most popular learning algorithm for training neural networks is still the backpropagation algorithm, which is a first order gradient descent method. Dozens of alternatives to backpropagation exist and that was a great topic for doctoral dissertations in the 1990s.

- **Quick propagation:** Assumes the error surface is quadratic. If the slope of the error change with respect to the weight change (Δerror/Δweight, the error gradient) switches sign, an approximation to the bottom of the error parabola is computed, and the weights are adjusted to the value that achieves that minimum-error estimate. Do not let the change in weights exceed a maximum value. It is usually several times faster convergence than backprop.
- **Resilient propagation (Rprop):** Recognizes that the magnitude of the error gradient is often noisy, so it considers only the sign of the slope. If the error gradient is the same sign, increase the learning rate. If the value is not the same, reduce the learning rate. The amount of the increase or decrease is not always changeable.
- **Second-order methods:** These methods converge in far fewer epochs than backpropagation but require significantly more computation and memory. Algorithmically they are significantly more complex as well, which is a primary reason they are included in only a few tools. Conjugate Gradient, Gauss-Newton, and Levenberg Marquardt algorithms are three methods that are found in software tools.

Neural Network Pruning

Thus far, the assumption in building the neural network has been that an architecture is specified and a neural network is trained. But which architecture generates the smallest testing dataset error? A modeler can build multiple neural networks with different architectures and determine empirically which is best. A few software implementations build in the ability to build many networks with different architectures. However, what about the inputs? Are all of them necessary, or are some irrelevant at best or even harmful?

Pruning algorithms for neural networks exist but are rarely implemented in mainstream predictive analytics software. These algorithms progressively remove inputs and neurons from the network in an attempt to improve error on the test data. The simplest approach to removing inputs and neurons is to identify those that have the least influence on the network predictions; if the weights throughout a neural network associated with an input are small, the input is essentially being ignored by the network and can be safely removed and the network retrained without the input.

The advantage of pruning is that the final neural network will be faster to deploy because of the fewer inputs, neurons, and weights that create additional multiplies and adds in the model. Moreover, the network could become more stable by removing terms.

Building neural networks with pruning can take considerable time, however, and therefore it is often left as a final step in the modeling-building process, after a good set of inputs for the model are found and a good baseline architecture is found.

Interpreting Neural Networks

Neural networks have the reputation of being "black boxes," meaning that the interpretation of why predicted values are small or large cannot be determined. This reputation is unfair to a large degree; while neural networks are not transparent, the input variables that most influence the predictions can be determined in a variety of ways.

Most software implementations of neural networks generate a set of variable influence scores for each variable in the neural network. These scores are sometimes called *variable importance* or *influence*. The methods used to determine the influence of each variable fall into two general categories. The first category examines weights associated with each input (attached to the first hidden layer) and subsequent weights between the first hidden layer and the output layer. These approaches may use partial derivatives or just multiply the weights to derive the relative influence of the variables.

The second category determines the influence indirectly by changing values of each input and measuring the amount of change in the output. The larger the relative change, the more influence the variable has. Some techniques hold all variables but one constant at the mean and randomly vary a single variable from its mean value plus or minus one standard deviation. The variables are then scored by the relative change in the predictions as each of the inputs are wiggled. A more extreme version of this approach is to remove a single variable completely, retrain the neural network, and examine how much the error increases. This last technique has the danger, however, that the neural network might converge in such a way that the two networks with and without the input are not comparable; it is best to start the training of the network without the input with the same weights found in the network being assessed, if possible.

However, there are also other methods to determine which variables are influential. In one approach, decision trees are trained from the same inputs as were used in training the neural network, but the target variable for the tree is the neural network output rather than the actual target variable. The tree therefore finds the key patterns the neural network is finding with the advantage that the decision tree generates rules that are more easily interpreted than neural network weights.

Neural Network Decision Boundaries

Since neural networks are universal approximators and can fit very complex functions, many modelers believed they would never need to use logistic regression again. However, the reality is that sometimes the more complex neural networks either overfit the data or fail to converge to an optimum solution. In other words, just because a neural network *in theory* is a universal approximately doesn't mean that *in practice* it will find that solution. Therefore, it is usually a good idea to build models using several algorithms.

Nevertheless, neural networks are usually better predictors than k-NN, logistic regression, or decision trees. As you can see in Figure 8-26, decision boundaries for the nasadata dataset are largely linear, although there are some nonlinear regions, such as between rye, corn, and oats. Moreover, the nonlinear decision boundaries are very smooth.

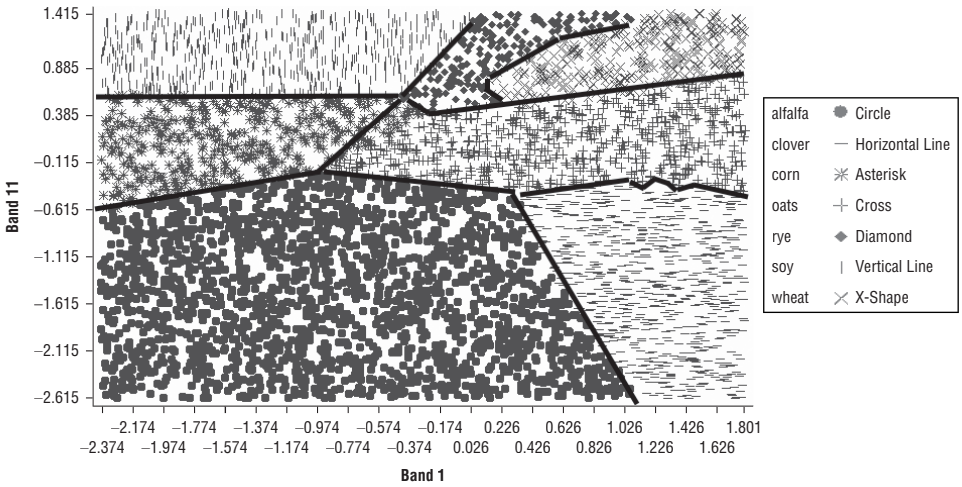

Figure 8-26: Neural network decision regions on nasadata

Other Practical Considerations for Neural Networks

ANNs, like other numeric algorithms including linear regression, logistic regression, k-nearest neighbor, and support vector machines requires numeric input data, and cannot have missing data. Typically, categorical data is represented numerically through the creation of dummy variables, as described in the discussion of logistic regression. Missing values can be imputed using typical imputation methods.

Variable selection for neural networks is done sometimes using the variable importance measure. For example, you may decide to keep only the top 20 variables in the network, and then retrain the neural network with only these variables. Some modelers use other algorithms that select variables automatically, like decision trees, to do variable selection for neural networks. This kind of variable selection is not optimal, but sometimes is practical.

K-Nearest Neighbor

The nearest neighbor algorithm is a non-parametric algorithm that is among the simplest of algorithms available for classification. The technique was first described by E. Fix and J. L. Hodges in 1951 in their paper "Discriminatory Analysis: Nonparametric Discrimination: Consistency Properties" (a work that was later republished in 1989) and by 1967 had been studied sufficiently that its theoretical qualities were known, namely that k-NN is a universal approximator with worst-case error that is bounded. The precise nature of its theoretical properties is not a concern here; nearest neighbor has proven to be an effective modeling algorithm. Because of these properties and the ease of training the models, nearest neighbor is included in most predictive analytics software packages.

The nearest neighbor algorithm is a so-called "lazy learner," meaning that there is little done in the training stage. In fact, nothing is done: The training data is the model. In essence, the nearest neighbor model is a lookup table.

The k-NN Learning Algorithm

The k-NN learning algorithm is quite simple: It begins with the data itself. In other words, k-NN is merely a lookup table that you use to predict the target value of new cases unseen during training. The character "k" refers to how many neighbors surrounding the data point you need to make the prediction." The mathematics behind the k-NN algorithm resides in how you compute the distance from a data point to its neighbors.

Consider the simple example in Figure 8-27. There are two classes of data in the training set—one class colored black and the other colored gray—and a total of 13 data points in two dimensions (the x-axis direction and y-axis direction). The 13 data points therefore are the model. To classify a new data point, x, using the 1-NN algorithm, you compare each and every one of the data points in the training data to x, computing the distance calculated from each of these comparisons. In Figure 8-27, the closest data point is gray, so x will be labeled as gray.

However, suppose you instead use three nearest neighbors to classify x. Figure 8-28 shows the ring within which the three nearest neighbors reside. If you decide which class to predict for x by majority vote, the prediction this time is the black class, different from the prediction using the 1-NN model.

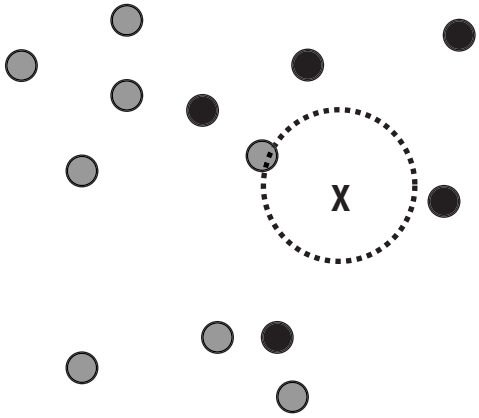

Figure 8-27: 1-nearest neighbor solution

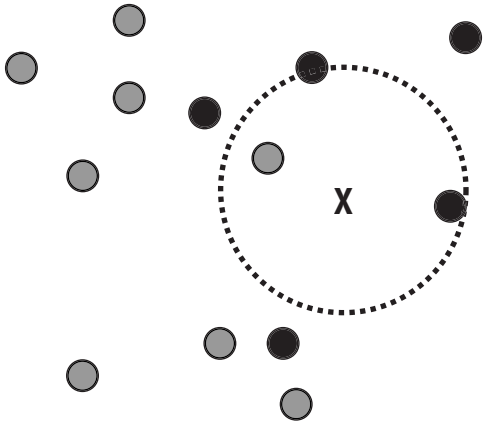

Figure 8-28: 3-NN solution

The more neighbors you include in the prediction, the smoother the predictions. For binary classification, it is common to use an odd number of nearest neighbors to avoid ties in voting.

There is no theory to tell you exactly how many neighbors to use. A small number of nearest neighbors can be very effective in finding small, subtle shifts in the data space, but can also be very susceptible to noise in the data. This is what appears to be the case in Figure 8-28: The gray dot inside the ring is an isolated example in that region, and a 3-NN solution is more robust. However, as the number of nearest neighbors increases, the classifier becomes less localized, smoother, less susceptible to noise, but also less able to find pockets of homogeneous behavior in the data, similar to a decision tree with only one split.

Therefore, the number of nearest neighbors you choose is often discovered through an iterative search, beginning with a small number of nearest neighbors, like 1, and continuing to increase the number of nearest neighbors until the testing data error rate begins to increase. Figure 8-29, for example, shows the decision regions for a 1-NN model built on the nasadata dataset with its seven crop types. The x axis is Band 1 and the y axis is Band 11. The shapes of the data points refer to the testing data classification predictions. Notice that while there are general, homogeneous regions for each crop type, there are still some local pockets of behavior with these regions. The *corn* crop type has some veins within the soy region in the upper left, and also in the *rye* region in the top-middle of the data space. The *wheat* predictions have a local pocket within the *rye* region, and *alfalfa* has a small group within the *clover* region in the lower right of the plot.

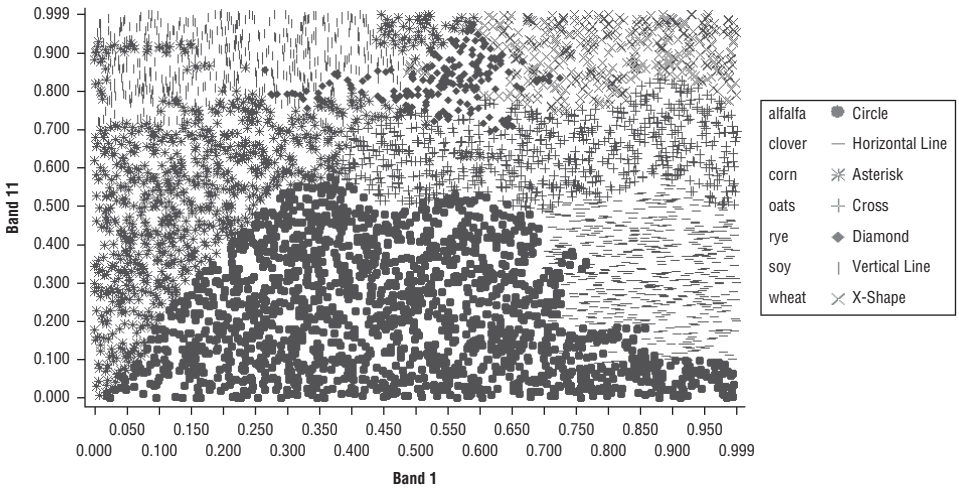

Figure 8-29: 1-NN decision regions for nasadata

Increasing the number of nearest neighbors to 3 reduces these localized pockets of behavior significantly, as shown in Figure 8-30. There are now only a few data points in the chart that do not correspond to the dominant crop type in the regions.

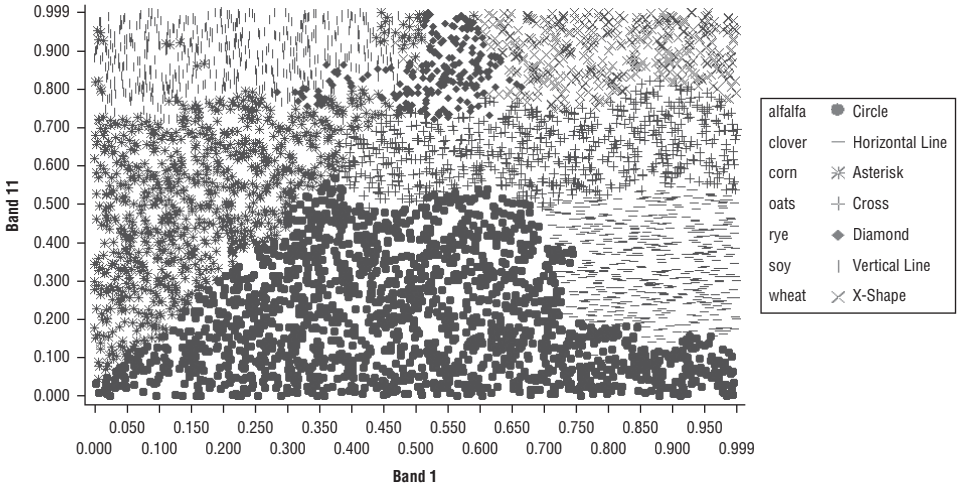

Figure 8-30: 3-NN decision regions for nasadata

When you increase k to 7, all seven regions become homogeneous, although not always smooth (see Figure 8-31).

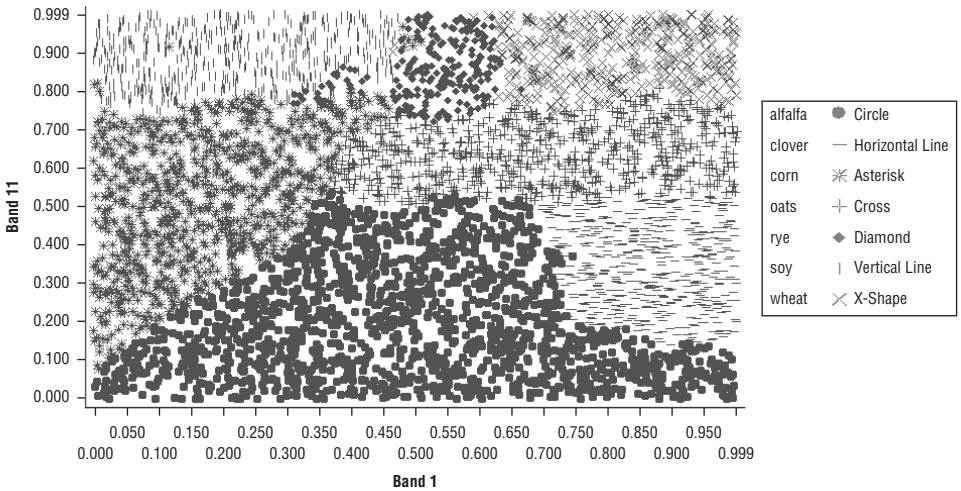

Figure 8-31: 7-NN decision regions for nasadata

In general, what value of k is correct? There is no theoretical answer to this question, but you can find an empirical solution by assessing model performance on testing data for each of the values of k you would like to test. As the value of k increases, the errors on testing data usually reach a minimum asymptotically.

For example, Table 8-9 shows the number of nearest neighbors, k, scored by AUC. The reduction in AUC from k equal to 101 to 121 is only 0.18%, showing convergence.

Table 8-9: The Number of Nearest Neighbors, K, Scored by AUC

NUMBER OF NN	AUC	% AUC REDUCTION
1	0.521	
3	0.536	2.83%
11	0.555	3.65%
21	0.567	2.14%
41	0.580	2.23%
61	0.587	1.23%
81	0.591	0.57%
101	0.594	0.66%
121	0.596	0.18%

Distance Metrics for k-NN

The primary distance metric used with the k-NN algorithm is Euclidean distance. Assume there are m records in the training data, and n inputs to the k-NN model. Collect the inputs into a vector called "x" of length n. The vector of inputs we are intending to predict based on the k-NN model is called y and is also of length n. The Euclidean distance can be expressed by the formula:

$$(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

where D is the Euclidean distance between the training vector x and the vector to classify, y. The summation merely adds the squares of the differences between training vector x and the vector to score called y. The distance will be computed m times, once for every record in the training data. The 1-NN algorithm finds the smallest of these distances and assigns to new data point y the target variable label of the nearest record.

Euclidean distance is the most commonly used distance metric, although others are sometimes found in software packages, including the Manhattan

distance, the Hamming distance, and the Mahalanobis distance. The Manhattan distance, rather than computing the squares of the differences between the training vector and the vector to score the new value, y, computes the absolute value of the differences.

$$D(x,y) = \sum_{i=1}^{n} |x_i - y_i|$$

The Mahalanobis distance is similar to the Euclidean distance except that the inputs are normalized by the covariance matrix. Covariance is analogous to variance (the square of the standard deviation) but measures the variance on two variables rather than a single one. To compute covariance, you multiply the standard deviation of variable 1, the standard deviation of variable 2, and the correlation between variables 1 and 2. The effect of normalizing by the covariance matrix is that the data is scaled to the unit sphere. The scaling takes into account the magnitudes of the variables if some variables have a larger spread than others.

Other Practical Considerations for k-NN

The k-NN algorithm usually has few options you need to set besides the value of k. Other common considerations you should consider include which distance metric to use, how to handle categorical variables, and how many inputs to include in the models.

Distance Metrics

Euclidean distance is known to be sensitive to magnitudes of the input data; larger magnitudes can dominate the distance measures. In addition, skew in distributions of the inputs can also effect the distance calculations. For example, consider a k-NN model built from two inputs from the KDD Cup 1998 data: RAMNTALL and CARDGIFT. The first 13 of 4,844 records in a sample of that data are shown in Table 8-10 and represent the training data. The mean value on the entire dataset (not just the 13 records shown here) for RAMNTALL is 112 and for CARDGIFT is 6.12, so RAMNTALL is more than 18 times larger on average than CARDGIFT.

Table 8-10: Two Inputs from KDD Cup 1998 Data

RECORD, I	RAMNTALL	CARDGIFT
1	98	6
2	119	9
3	61	10
4	123	12
5	68	6
6	102	14
7	132	5
8	94	8
9	38	3
10	30	5
11	44	2
12	25	1
13	35	2
Mean	112	6.12

If you assume that you are evaluating a new datapoint with the value of RAMNTALL equal to 98 and CARDGIFT equal to 6, Table 8-11 shows the distances from these 13 records in the training data. The distance in the first record represents the distance from the record to score to the first record in the training data. Notice that most of the distance values for RAMNTALL are much larger than they are for CARDGIFT (they are more than 10 times larger), precisely because RAMNTALL itself is larger. RAMNTALL therefore is the primary contributor to Total Distance.

Table 8-11: Euclidean Distance between New Data Point and Training Records

RECORD NUMBER IN TRAINING DATA	TRAINING RAMNTALL	TRAINING CARDGIFT	NEW RAMNTALL TO SCORE	NEW CARDGIFT TO SCORE	DISTANCE
1	0	5	98	6	5.00
2	21	3	98	6	21.21
3	37	4	98	6	37.22
4	25	6	98	6	25.71

RECORD NUMBER IN TRAINING DATA	TRAINING RAMNTALL	TRAINING CARDGIFT	NEW RAMNTALL TO SCORE	NEW CARDGIFT TO SCORE	DISTANCE
5	30	0	98	6	30.00
6	4	8	98	6	8.94
7	34	1	98	6	34.01
8	4	2	98	6	4.47
9	60	3	98	6	60.07
10	68	1	98	6	68.01
11	54	4	98	6	54.15
12	73	5	98	6	73.17

For this reason, practitioners usually transform the input variables so their magnitudes are comparable, a process often called *normalization*. Let's assume you are transforming RAMNTALL: The most common transformations applied to continuous data are shown in Table 8-12. The functions Mean, Min, Max, and StdDev are not presented in mathematical terms but rather in functional terms and refer to an operation on the entire column in the training data.

Table 8-12: Two Transformations for Scaling Inputs

TRANSFORMATION NAME	TRANSFORMATION FORMULA
Z-score	$RAMNTALL_z = \frac{(RAMNTALL - Mean(RAMNTALL))}{StdDev(RAMNTALL)}$
Min-max	$RAMNTALL_{minmax} = \frac{(RAMNTALL - Min(RAMNTALL))}{Max(RAMNTALL) - Min(RAMNTALL)}$

The z-score is very appealing for k-NN and other algorithms whose errors are based on Euclidean distance or squared error because the z-score has zero mean and unit standard deviation. A typical range for the z-scored variables will be between –3 and +3, and all variables will therefore have similar magnitudes, an important consideration for k-NN.

However, computing the z-score assumes the distribution is normal. If, for example, the distribution is severely skewed, the mean and standard deviation will be biased and the z-score units won't accurately reflect the distribution.

Moreover, and more importantly for k-NN, the values of the variable won't be in the range –3 to +3. RAMNTALL actually has z-score values as high as 18.

An alternative is to use the min-max transformation. The formula in Table 8-11 converts the variable from its original units to a range from 0 to 1. You could scale this result to the range –1 to +1 if you prefer to have a center value 0 in the transformed variable; you merely multiply the min-max transformation result by 2 and then subtract 1.

One advantage of the min-max transformation is that it is guaranteed to have bounded minimum and maximum values, and therefore the transformation will put all the transformed variables on the same scale. Also, if one is including dummy variables in the nearest neighbor model, they, too, will be on the scale of 0 to 1.

However, just as skewed distributions cause problems for z-scores, they also cause problems for the min-max transformation. Suppose RAMNTALL has one value that is 10,000, the second largest value is 1,000, and the minimum value is 0. The 10,000 value will map to the transformed value 1, but the second highest value will map to 0.1, leaving 90 percent of the range (from 0.1 to 1) completely empty except for the one value at 1. This variable would then be at a severe disadvantage for use in a k-NN model because its values are almost always less than 0.1, whereas other variables may have values distributed throughout the 0 to 1 range.

Therefore, you must take great care *before* applying any normalization that the distribution of the variable to be transformed is not heavily skewed and does not have outliers. Methods to treat these kinds of data were described in Chapter 4. The modeler must choose the course of action based on the pros and cons of each possible correction to the data.

Handling Categorical Variables

k-NN is a numerical algorithm requiring all inputs to be numeric. Categorical variables must therefore be transformed into a numeric format, the most common of which is 1/0 dummy variables: one column for each value in the variable.

If many key variables are categorical and dummies have been created to make them numeric, consider matching the continuous variables with the categorical variables by using min-max normalization for the continuous variables. However, k-NN models with many categorical variables can easily be dominated by the categorical variables because the distances generated by the dummy variables are either the minimum possible distance, 0 (if the new data point value matches a neighbor) or the maximum possible distance 1 (if the new

data point doesn't match the neighbor). The maximum distance will influence the overall distance more than continuous variables whose Euclidean distance will have the full range 0 to 1.

One alternative to min-max scaling is to rescale the dummy variables of the new data points from 1 to 0.7 so the maximum distance is reduced. A second alternative is to rescale the dummy variables so that the 0 values are coded as –2 and the 1 values are coded as +2, values that are smaller than the maximum and larger than the minimum.

The Curse of Dimensionality

One of the challenges with k-NN and other distance-based algorithms is the number of inputs used in building a model. As the number of inputs increases, the number of records needed to populate the data space sufficiently increases exponentially. If the size of the space increases without increasing the number of records to populate the space, records could be closer to the edge of the data space than they are to each other, rendering many, if not all, records as outliers. This is the curse of dimensionality. There is very little help from theory for us here; descriptions of how much space is too much, and how much additional variables can negatively impact k-NN models (if they do at all), have not been forthcoming.

One solution to the curse of dimensionality is to keep dimensionality low; include only a dozen to a few dozen inputs in k-NN models. Additionally, the quality of the inputs is important. If too many irrelevant features are included in the model, the distance will be dominated by noise, thus reducing the contrast in distances between the target variable values. Third, you should exclude inputs that are correlated with other inputs in the modeling data (exclude one of the two correlated variables, not both). Redundant variables give the false impression of smaller distances. However, reducing dimensionality too much will result in poorer predictive accuracy in models.

Weighted Votes

Some k-NN software includes an option to weight the votes of the nearest neighbors by the distance the neighbor is from the data point. The reason for this is to compensate for situations where some of the nearest neighbors are very far from the data point and intuitively would not be expected to contribute to the voting. The weighting diminishes the influence.

Naïve Bayes

The Naïve Bayes algorithm is a simplification of the Bayes classifier, an algorithm with a long and successful history. It's named after English mathematician and Presbyterian minister Thomas Bayes, developer of Bayes' theorem, or Bayes' rule, in the 1740s. But it wasn't until the 1930s that it became a contender in data analysis. It was proposed by Harold Jeffreys as an alternative to the approaches developed and advocated by English statistician Sir Ronald Aylmer Fisher, including the use of statistical tests and p-values. However, it wasn't until the 1980s that Bayesian methods become more mainstream in data analysis, and in the 1990s, the Naïve Bayes algorithm began appearing in technical writings and as an algorithm used in data mining competitions.

Bayes' Theorem

Before describing the Bayes' theorem and Naïve Bayes algorithm, first consider two variables, and a probability that each is "True" labeled A and B.

- P(A) is the probability that a variable has value A (or that A is the "true" value).
- P(~A) is the probability that a variable doesn't have the value A.
- P(B) is the probability that a second variable has the value B (or that B is the "true" value).
- P(~B) is the probability that the second variable doesn't have the value B.
- P(A and B) is the probability that both A and B are true.
- P(A or B) is the probability that either A or B is true.

The values P(A) and P(B) are called the *prior probabilities* that A is true or that B is true respectively, or just *priors*. These values are either set by a domain expert who knows the likelihood an event has occurred, calculated from historical data, or can be supposed in an experiment. Another way some describe these probabilities is as the initial degree of belief that A is true (or B is true).

Usually, in predictive modeling problems, the values are calculated from the data. However, as described in the Chapter 8 it is sometimes advantageous to override the historical values in the data if the modeler would like to alter how the algorithms interpret the variable.

A conditional probability is the probability that a condition is true, given that a second condition is true. For example,

P(A | B) is the probability that A is true given that B is true, and

P(B | A) is the probability that B is true given that A is true.

Some describe P(A|B) as a *posterior* probability: the degree of belief having accounted for B. Note that P(A|B) is not, in general, equal to P(B|A) because the two conditional probabilities presuppose different subsets of data (the former when B is true and the latter when A is true).

The conditional probabilities are formulated as follows:

$$P(A \mid B) = P(A \text{ and } B) \div P(B), \text{ and}$$

$$P(B \mid A) = P(A \text{ and } B) \div P(A)$$

If you rewrite these equations, you have:

$$P(A \text{ and } B) = P(A \mid B) \times P(B) \text{ and}$$

$$P(A \text{ and } B) = P(B \mid A) \times P(A)$$

This is sometimes called the "chain rule" of conditional probabilities.

Now consider Figure 8-32 with its abstract representation of these two conditions. The overlap between the two conditions is the "A and B" region in the figure. This region relates to the P(A and B) in the equations already described: P(A and B) is the product of the conditional probability P(A|B) and the prior of B, P(B). Or, stated another way, the intersection of the two sets is the conditional probability times the prior.

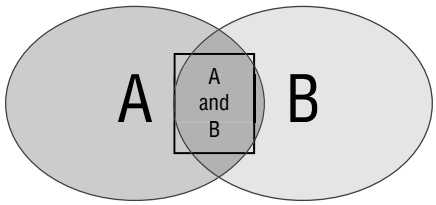

Figure 8-32: Conditional probability

Bayes' theorem tells you how to compute the probability that A is true given B is true, P(A|B), with the formula:

$$P(A \mid B) = P(B \mid A) \times P(A) \div P(B).$$

This is the form that you usually see in the literature, but the A and B values are just placeholders for actual conditions you find in their data.

For example, consider a doctor's office with a patient coming in who is sneezing (S) that wants to be able to diagnose if the patient has a cold (C) or pneumonia (P). Let's assume that there are probabilities that have already been computed or assumed, including:

$$P(S) = 0.3 \text{ (30 percent of patients come in sneezing)}$$

$$P(C) = 0.25 \text{ (25 percent of patients come in with a cold)}$$

$$P(P) = 1 \times 10^{-6} \text{ (1 in a million patients come in with pneumonia)}$$

These three are prior probabilities. The role of "A" in Bayes' theorem will be the probability the diagnosis is a cold, and then in a second computation, the probability that the diagnosis is pneumonia. In the language of predictive analytics, the target variables or the outcomes of interest are "cold" or "pneumonia."

You also need conditional probabilities for Bayes' theorem to work, namely P(S|C) and P(S|P), or in words, what is the probability that the patient sneezes because he has a cold, and what is the probability the patient sneezes because he has pneumonia. Let's assume both of these are 100 percent.

The solution for the cold is:

$$P(C \mid S) = P(S \mid C) \times P(C) \div P(S)$$

$$P(C \mid S) = 1.0 \times 0.25 / 0.3 = 0.83$$

$$P(P \mid S) = P(P \mid C) \times P(P) \div P(S)$$

$$P(P \mid S) = 1.0 \times 10^{-6} \div 0.3 = 3 \times 10^{-6}$$

In summary, the probability that the patient has a cold is 83 percent, whereas the probability the patient has pneumonia is only 3 chances in a million. Clearly, the disparity of the two probabilities is driven by the priors associated with the outcomes.

One key difference between the Bayes approach and other machine learning approaches described so far is the Bayes approach considers only one class at a time. The patient example had only two outcomes, so two calculations were made. For the nasadata, with seven target value classes, seven probabilities will be computed, one each for alfalfa, clover, corn, soy, oats, rye, and wheat. To predict a single outcome, you take the largest probability as the most likely outcome.

So far, the description of the Bayes classifier has been in probabilistic terms. Mathematically, for normal distributions, the Bayes classifier algorithm computes a mean value for every input (a mean vector) and the covariance matrix (the variance in one dimension, covariance in more than one). The maximum probability solution is the one with the smallest normalized distance to the mean of each population.

Consider the same simple data shown in Figure 8-27 for k-NN. With k-NN, the distance from the new value "x" and each of the data points was computed. The Bayes classifier, on the other hand, only needs to compute the distance from "x" to the mean of each class: gray and black in this case, or two distances. The rings around the mean values are normalized units of covariance. As you can see in Figure 8-33, "x" is closer to the gray class mean, less than one unit of covariance, than to the black class, so it is assigned the label "gray" by the classifier. In addition, the distance from the mean for each class can be appended to the data much like probabilities in logistic regression and confidences with neural networks.

This distance is not the Euclidean distance used in k-NN but a normalized distance, where the covariance matrix is the normalizer. The effect of this normalization is that magnitudes of the inputs are handled automatically (unlike k-NN), and rather than building piecewise linear decision boundaries as is done with k-NN, the Bayes classifier constructs quadratic decision boundaries, so it is more flexible and matches the normal distributions assumed in the data perfectly.

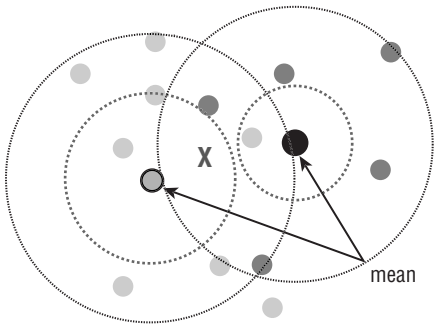

Figure 8-33: Bayes classifier distances

The Bayes classifier is an excellent algorithm, but what happens when there are many inputs to be considered in the probability calculations? Consider the patient example again, and instead of having just one symptom (sneezing), assume there are two symptoms: sneezing and fever (F). The Bayes' theorem solution for colds is now much more complex.

For two inputs, you must compute additional probabilities and conditional probabilities, including the following:

P(F) = probability of fever in patients

P(F and S | C)

P(F and ~S | C)

P(~F and S | C)

P(~F and ~S | C)

P(F and S | P)

P(F and ~S | P)

P(~F and S | P)

P(~F and ~S | P)

As the number of inputs increases, the number of conditional probabilities to compute also increases, especially those that include combinations of inputs. The problem becomes one of data size: Is there enough data to compute all of the

conditional probabilities? Have they all occurred often enough for these measures to be reliable? Mathematically, the Bayes classifier requires the computation of a covariance matrix. Is there enough data for these measures to be stable?

The Naïve Bayes Classifier

One solution to the problem of computing large numbers of conditional probabilities is to assume independence of the input variables, meaning you assume that input variables are unrelated to one another. With the assumption of independence, the conditional probabilities involving combinations of inputs are zero and you are left with only the conditional probabilities relating the output variable to the input variables, an enormous simplification. This is the assumption behind the Naïve Bayes classifier: We naively assume independence of inputs.

For the patient example, the calculation of the probability of having a cold from the measured inputs becomes:

$$P(C|S \text{ and } F) = P(S|C) \times P(F|C) \times P(C) \div P(S) \text{ and}$$
$$P(P|S \text{ and } F) = P(S|C) \times P(F|P) \times P(P) \div P(S)$$

The complex interaction of all the input variables has now become a simple multiplication of first order conditional probabilities. Not only does this simplify the building of models, but it also simplifies the interpretation.

If one applies the Naïve Bayes algorithm to the nasadata, just including Band1 and Band11 as inputs, the decision regions shown in Figure 8-34 are found. The shapes correspond to the maximum probability regardless of its value. The regions are largely linear and tend to be "boxy" due to the categorical inputs. These regions are very similar to those found by the 7-NN algorithm (refer to Figure 8-31) in the upper third of the scatterplot, though it does not have the nonlinear decision regions; the decision boundaries are all perpendicular or parallel to the Band 1 (x axis) and Band 11 (y axis).

Interpreting Naïve Bayes Classifiers

Naïve Bayes models are interpreted by examining the conditional probabilities generated in the training data. Most often, software reports these as lists of probabilities, like the ones shown in Table 8-13. The training data for the Naïve Bayes model is a stratified sample, with 50 0s and 50 percent 1s, so the baseline rate for comparison in the table is 50 percent. The model report is very easy to understand because each variable has its own list of probabilities. For RFA_2F, it is clear that when RFA_2F is equal to 4, 65 percent of the donors that match this value are responders, which is the group with the highest probability. Only

the RFA_2F equal to 1 group is lower than the average response rate equal to 50 percent.

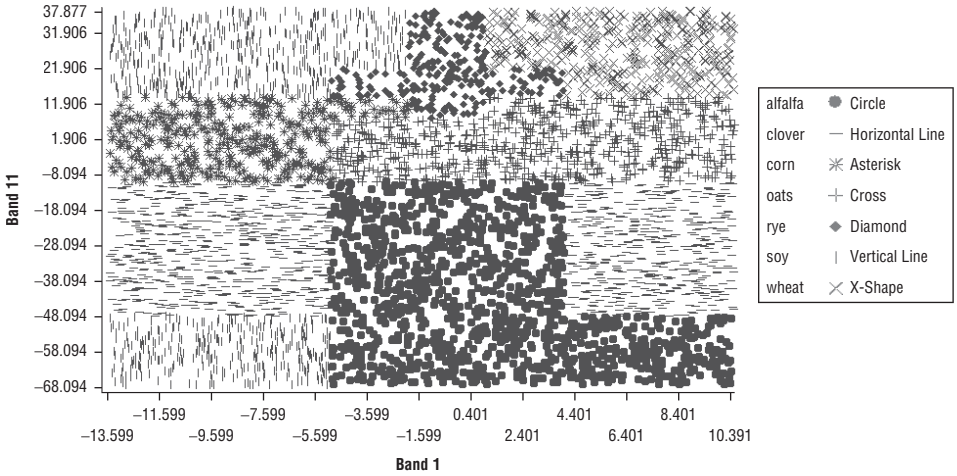

Figure 8-34: Naïve Bayes model for nasadata

Table 8-13: Naïve Bayes Probabilities for RFA_2F

CROSSTAB	RFA_2F = 1	RFA_2F = 2	RFA_2F = 3	RFA_2F = 4
Counts with TARGET_B = 0	1268	510	382	266
Counts with TARGET_B = 1, counts	930	530	471	494
Percent Target_B = 1	42.3%	51.0%	55.2%	65.0%

Each variable has its own table of results, and since Naïve Bayes assumes each input variable is independent, there are no tables of interactions between inputs. Some software packages also provide a table with a list of the most predictive input variables in the model, a report that is very useful to any analyst.

Other Practical Considerations for Naïve Bayes

Additional data preparation steps to consider when you build Naïve Bayes models include:

Naïve Bayes requires categorical inputs. Some implementations will bin the data for you, and others require you to bin the data prior to building the Naïve Bayes models. Supervised binning algorithms can help generate better bins than simple equal width or equal count bins.

Naïve Bayes is susceptible to correlated variables. Naïve Bayes can suffer from poor classification accuracy if input variables are highly correlated because of the assumption of independence and therefore the replication of probabilities from multiple variables in the model. It is best if the variables of only one of the correlated variables is included in the model.

Naïve Bayes does not find interactions. If you know that there are interactions in the data, create them explicitly for the Naïve Bayes model. This is similar to the strategy you use with logistic regression models to ensure the interactions are considered by the algorithm, and can improve accuracy significantly.

Regression Models

Regression models predict a continuous target variable rather than a categorical target variable. In some ways, this is a more difficult problem to solve than classification. Classification has two or a few values to predict: two for the KDD Cup 1998 dataset, three for the Iris data set, and seven for the nasadata dataset, to name three examples. The models have to predict the outcome correctly for these groups. Regression models must predict every value contained in the target variable well to have high accuracy.

Regression belongs to the supervised learning category of algorithms along with classification. I am using the term "regression" for this kind of model, but several other terms are used to convey the same idea. Some modelers call these models "continuous valued" prediction models; others call them "estimation" or "function estimation" models.

The most common algorithm predictive modelers use for regression problems is linear regression, an algorithm with a rich history in statistics and linear algebra. Another popular algorithm for building regression models is the neural network, and most predictive analytics software has both linear regression and neural networks. Many other algorithms are also used for regression, including regression trees, k-NN, and support vector machines. In fact, most classification algorithms have a regression form.

Throughout this section, the KDD Cup 1998 data will be used for illustrations, though instead of using the binary, categorical target variable TARGET_B that was used to illustrate classification modeling, a different target variable is predicted for regression problems, TARGET_D. This target variable is the amount a donor gave when he or she responded to the mail campaign to recover lapsed donors.

Linear Regression

Most analysts are familiar with linear regression, easily the most well-known of all algorithms used in regression modeling. Even analysts who don't know about linear regression are using it when they are adding a "trend line" in Excel. Figure 8-35 shows a scatterplot with a trend line added to show the best fit of the data points using linear regression. The linear regression algorithm finds the slope of the output with respect to the input. In the model in Figure 8-35, every time the variable on the x axis increases 10 units, the target variable on the y axis increases approximately 2.5 units.

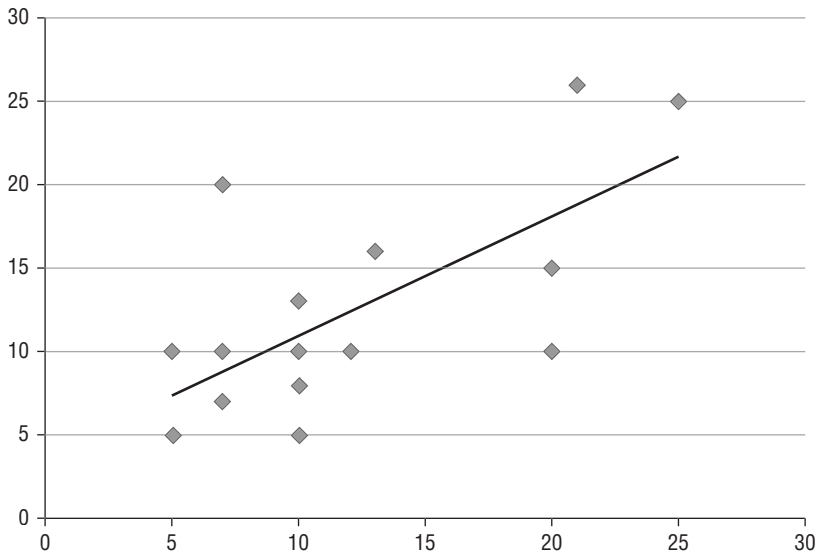

Figure 8-35: Regression line

This discussion presents linear regression from a predictive modeler's perspective, not a statistician's perspective. Predictive modelers usually use linear regression in the same way other algorithms are used, primarily to predict a target variable accurately.

The form of a linear regression model is identical to the equations already described for the neuron within a neural network:

$$y_i = w_0 + w_1 \times x_1 + w_2 \times x_2 + \ldots + w_n \times x_n$$

As before, the *y* variable is the target variable—the output. The inputs to the model are *x1*, *x2*, and so forth. The weights or coefficients are *w0*, *w1*, *w2*, and so forth. To be more precise, however, linear regression only includes the output variable, *y*, and a single input variable, *x*; we regress the target variable on the input. The full equation is multiple regression where we regress the target variable on all the input variables.

Often, the variables *y* and *x* have subscripts, like the letter *i* in the regression equation just shown, which refer to the record number; each record will have its own predicted value based on the input values in that record. In addition, the predicted target value is often represented with a carat above it, usually pronounced as "hat," indicating it is a predicted value in contrast to the actual value of the target variable, *y*.

The difference between the target variable value in the data and the value predicted by the regression model, *y* minus *y* hat, is called a *residual* and is often annotated with the letter *e*, as in the equation:

$$e = y_i - \hat{y}_i$$

Strictly speaking, residuals are different than the regression error term, which is defined as the difference between the target variable value and what statisticians call the *true regression line*, which is the perfect regression model that could be built if the data satisfied all of the regression assumptions perfectly, and the training data contained all of the patterns that could possibly exist. Predictive modelers don't usually include any discussions or descriptions of this kind of idealized model, and therefore often treat the terms residuals and errors synonymously.

Residuals are visualized by dropping a perpendicular line between the data value and the regression line. Figure 8-36 shows the target variable, TARGET_D, fit with a regression line created from input variable LASTGIFT. The residuals are shown as vertical lines, the distance between the actual value of TARGET_D and the predicted value of the TARGET_D (*y* hat in the equation) as predicted by the linear model. You can see there are large errors above and below the regression line. If one or more of the data points above the regression line were removed, the slope of the regression line would certainly change.

A plot of the residuals versus LASTGIFT is shown in Figure 8-37. Note that magnitudes of residuals are approximately the same regardless of the value of LASTGIFT, the residuals are both positive and negative, and there is no shape to the residuals. If, however, the shape of the residuals versus LASTGIFT is quadratic, there is a quadratic relationship between LASTGIFT and TARGET_D that is missing in the model and should be included by adding the square of LASTGIFT as an input. Any detectable pattern in the residual plot indicates that one or more derived variables can be included to help improve the fit of the model.

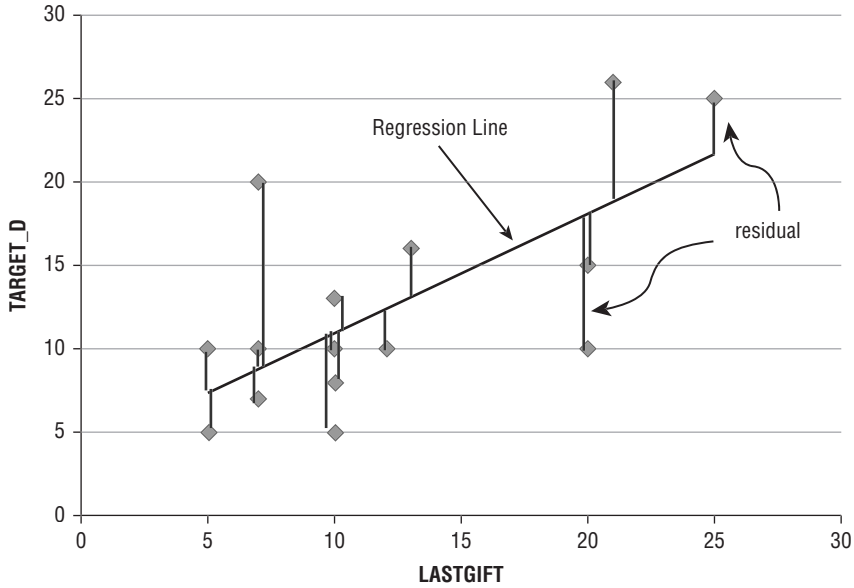

Figure 8-36: Residuals in linear regression

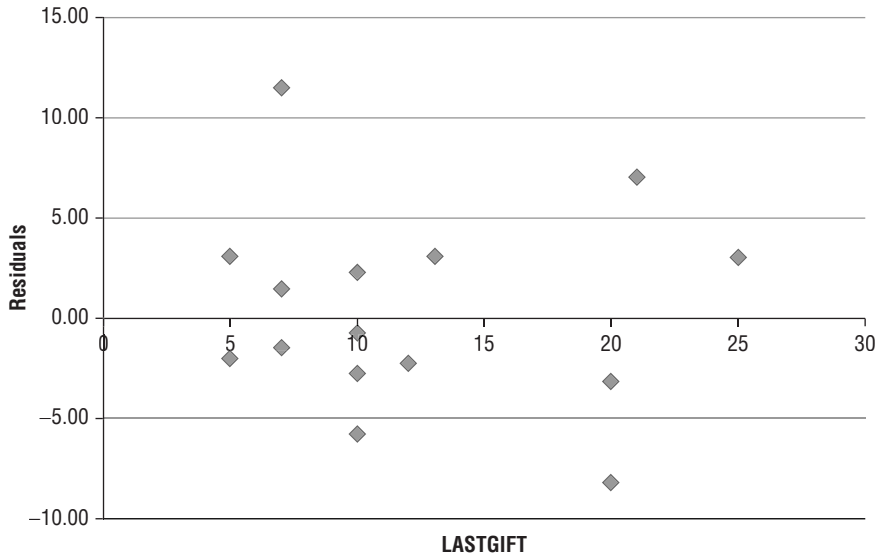

Figure 8-37: Residuals vs. LASTGIFT

The influence of large residuals is magnified because the linear regression minimizes the *square* of the residuals. The reason that outliers, especially outliers that have large magnitudes compared to the rest of the input values, create

problems for regression models is because of the squared error metric. This is also the reason that positive skew has such influence on regression models; models will be biased toward trying to predict the positive tail of a distribution because of the squared error metric.

However, this is not always bad. Consider the KDD Cup 1998 data again. TARGET_D is positively skewed, as most monetary variables are. Therefore, linear regression models trying to predict TARGET_D will be biased toward predicting the positive tail of TARGET_D over predicting the smaller values of TARGET_D. But is this bad? Don't we want to identify the large donors well, perhaps even at the expense of worse predictive accuracy for smaller donors? The question of transforming TARGET_D is not just a question of linear regression assumptions; it is also a question of which cases we want the linear regression model to predict well.

Linear Regression Assumptions

The linear regression algorithm makes many assumptions about the data; I've seen lists of assumptions numbering anywhere from four to ten. Four of the assumptions most commonly listed include the following. First, the relationship between the input variables and the output variable is assumed to be linear. If the relationship is not linear, you should create derived variables that transform the inputs so that the relationship to the target becomes linear. Figure 8-38 shows what appears to be a quadratic relationship between the input on the x axis and the output on the y axis. The linear fit from a regression model is shown by the line, which obviously doesn't fully capture the curvature of the relationship in the data: The data violates the assumption that the relationship is linear.

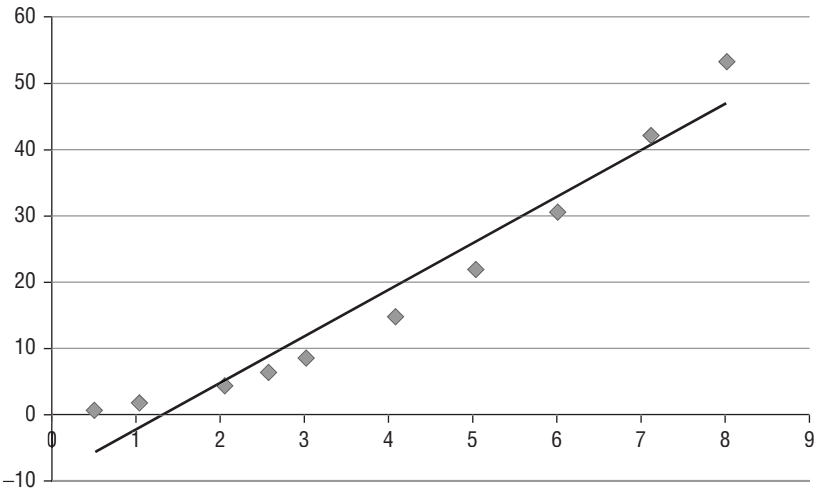

Figure 8-38: Linear model fitting nonlinear data

A residual plot, like Figure 8-37, would reveal the quadratic relationship between the residuals and the input presented by the x axis. The corrective action is to create a derived variable to linearize the relationship. For example, rather than building a model using the input directly, if the square of the input is used instead, the scatterplot is now linear, as shown in Figure 8-39.

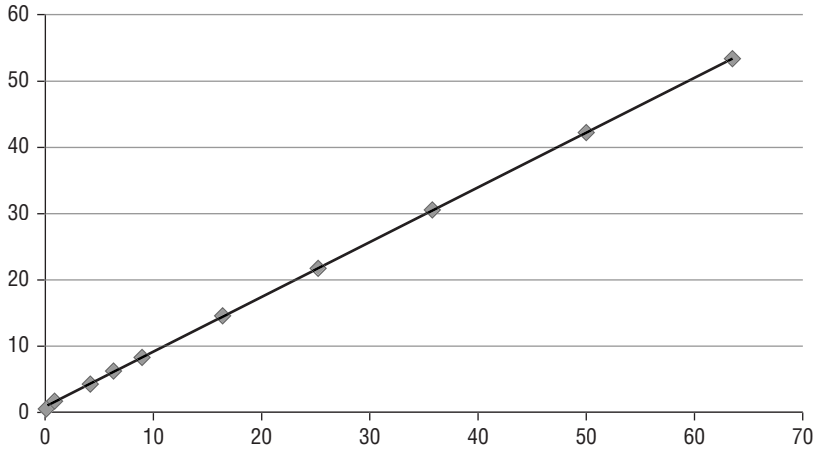

Figure 8-39: Linear model after transforming nonlinear data

The second assumption is that the inputs are uncorrelated with each other, or, in other words, the correlations between input variables are all zero. If this isn't the case, then the regression coefficient for one variable may carry some of the influence from another variable, blurring the influence of a variable on the outcome. Consider the following model for TARGET_D:

$$\text{TARGET\_D} = 2.3261 + 0.5406 \times \text{LASTGIFT} + 0.4319 \times \text{AVGGIFT}$$

This equation says that every increase in LASTGIFT of $10 will produce an increase in the amount the lapsed donor gives to the recovery mailing (TARGET_D) $5.41 (5.406 rounded to the nearest penny). However, LASTGIFT and AVGGIFT are correlated with a correlation coefficient equal to 0.817. If AVGGIFT is removed from the model, the new regression equation is:

$$\text{TARGET\_D} = 3.5383 + 0.7975 \times \text{LASTGIFT}$$

Now the interpretation of the relationship between LASTGIFT and TARGET_D is different: Every increase in LASTGIFT of $10 will produce an increase of $7.975 in the amount the lapsed donor gives. The difference is the correlation between the two variables, meaning they share variance, and therefore, they share influence on the target variable.

High levels of correlation, therefore, should be avoided if possible in regression models, though they cause more problems for interpretation of the models

rather than the model accuracy. However, if variables are correlated extremely high, 0.95 or above as a rule of thumb, the correlation can cause instability that is sometimes called *destructive collinearity*. One symptom of this is seen when the linear regression coefficients of two highly correlated terms (positively) are both very high and opposite in sign. In these cases, the two terms cancel each other out and even though the terms appear to be influential, they are not, at least not on the training data. However, if new data contains values of these two variables that deviate from the high correlation, the large coefficients produce wild fluctuations in the predicted values.

Most statistics software contains collinearity diagnostics that alert you when there is dangerous collinearity with the inputs to the model. Even if you have already removed highly correlated variables during Data Preparation, the collinearity could exist in three or more variables. Because of this, some practitioners routinely apply Principal Component Analysis to find these high levels of correlation before selecting inputs for modeling. If you used the principal components (PC) themselves as inputs to a linear regression model, you are guaranteed to eliminate collinearity because each PC is independent of all others, so the correlations are zero. If you instead use the single variable that loads highest on each PC, you will reduce the possibility for collinearity.

The remaining assumptions you find in the literature all relate to the residual, which are assumed to be normally distributed with a mean of zero and equal. In other words, residuals should contain only the noise in the data, also referred to as the unexplained variance.

This last set of assumptions, especially the normality assumption, is the reason many analysts transform inputs so that they are normally distributed. For example, with positively skewed inputs, many will transform the inputs with a log transform, and this is exactly the approach that was described in Chapter 4. This is not necessary, strictly speaking, to comply with regression assumptions, but beginning with inputs and outputs that are normally distributed increases the likelihood that the assumptions will be met.

Variable Selection in Linear Regression

Linear regression fits all of the inputs to the target variable. However, you will usually do some variable selection, especially during the exploratory phase of building models. This step is very important to modelers: Identifying the best variables to include in the model can make the difference between deploying an excellent model or a mediocre model.

Perhaps the simplest method of variable selection is called *forward selection*, which operates in the same way variable selection occurs in decision trees. In forward selection, a model is built using all of the input variables, one at a time. The variable with the smallest error is kept in the model. Then, all of the remaining variables are added to this variable one at a time, and a model is built

for each of these combinations. The best two-term model is kept. This process continues until a condition applies to stop the adding of more variables, most often to prevent the model from overfitting the training data.

Some software provides the option to do *backward selection*, where the first model is built using all of the inputs, each variable is removed one at a time from the set of inputs, a model is built using each of these subsets, and the best model is retained, now with one variable removed. The process of removing the variable that contributes the least to reducing fitting error continues until no input variables remain or a stop condition applies.

A third approach, called *stepwise* variable selection, is a combination of these two. The algorithm begins by finding the best single variable (forward selection), and alternatively adds a new variable or removes a variable until a stop condition applies. Stepwise regression is preferred because it usually finds better variable subsets to include in the regression model.

How does forward, backward, or stepwise regression decide which variables to include or remove? Chapter 4 described several methods that are frequently used, including the Akaike information criterion (AIC) and minimum description length (MDL). Other metrics that are sometimes used include the Bayesian information criterion (BIC) and Mallow's *Cp*. All of these metrics function the same way: They compute a complexity penalty for the regression model from the number of inputs in the model and the number of records the model is built from. The more records you have, the more terms that can be added to the model without incurring a higher penalty.

Figure 8-40 shows a representation of what these metrics do. As the number of inputs in a regression model increases, the error of the regression model (fitting error in the figure) becomes smaller. Each term added to the model, however, incurs a complexity penalty that grows linearly with the number of terms in the model. These two numbers added together provide a number that trades off fitting with complexity. In the figure, five inputs provide the best tradeoff between fitting error and complexity.

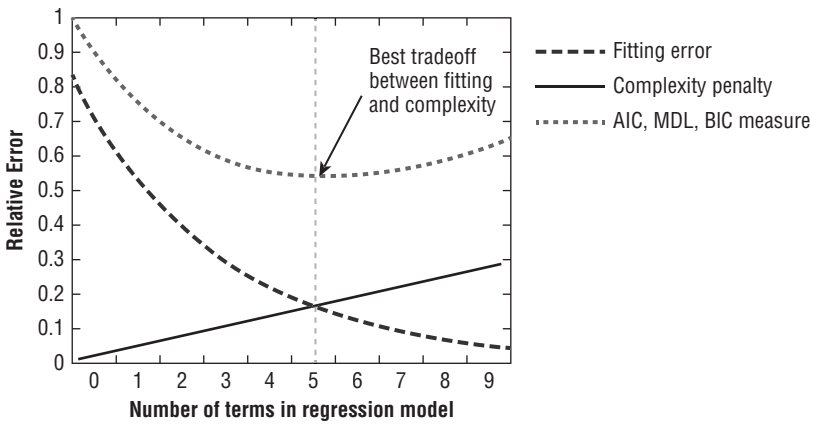

Figure 8-40: Trading fitting error and complexity

Interpreting Linear Regression Models

Linear regression equations are interpreted in two ways: through the coefficients and the *p* values. The coefficients describe the amount of change to expect in the target variable prediction for every unit change in the input variable. This kind of interpretation has already been discussed. But which input variables are most influential in predicting the target variable? If all the inputs were scaled to the same range, such as with min-max normalization or z-score normalization, the regression coefficients themselves show the influence of the inputs, and the input associated with the largest coefficient magnitude is the most important variable.

Consider the simple linear regression model summarized in Table 8-14, predicting TARGET_D. The largest coefficient in the model (excluding the constant) is RFA_2F. The values of RFA_2F are 1, 2, 3, and 4. However, the smallest *p* values are for LASTGIFT and AVGGIFT, whose *p* values don't show any value at all to four significant digits. These two are the strongest predictors in the model. An average value of LASTGIFT, 14, is nearly seven times larger than the average value for RFA_2F. If RFA_2F had the same influence on TARGET_D as LASTGIFT, its coefficient would be seven times larger than LASTGIFT.

Table 8-14: Regression Model Coefficients for TARGET_D Model

VARIABLE	COEFFICIENT	*P*
Constant	13.4704	0.0473
LASTGIFT	0.4547	0.0000
AVGGIFT	0.4478	0.0000
RFA_2F	−0.5847	0.0002
NGIFTALL	−0.0545	0.0349
FISTDATE	−0.0009	0.2237

Some modelers use a *p* value of 0.05 as the rule of thumb to indicate which variables are significant predictors and which should be removed from the model. Using this principle, FISTDATE is not a significant predictor and therefore should be removed from the model.

However, this approach has several problems. First, there is no theoretical reason why 0.05 is the right threshold. As the number of records increases, the *p* values for inputs will decrease even if the model is no more predictive just because of the way *p* is calculated. Second, there are other more direct ways to assess which inputs should be included, such as AIC, BIC, and MDL. Third, just because a variable is not a significant predictor doesn't necessarily mean

that it is harmful to the model. In this model, removing FISTDATE from the model actually reduces model accuracy on testing data according to one common measure, R^2, from 0.589 to 0.590, a tiny change. Clearly, FISTDATE isn't contributing significantly to model accuracy, nor is it contributing to overfitting the target variable.

Using Linear Regression for Classification

Linear regression is usually considered as an algorithm that applies only to problems with continuous target variables. But what would happen if linear regression tried to predict a categorical target variable coded as 0 and 1? Figure 8-41 shows what this kind of problem would look like. Linear regression finds a line that tries to minimize the squares of the errors. With only two values of the target variable, the regression line obviously cannot fit the data well at all. Notice that what it tries to do in the figure is put the line in the center of the density of data points with values 0 and 1.

A regression fit of the dummy variable, while creating a poor estimate of the values 0 and 1, can create a score that rank-orders the data. In the figure, it is clear that the smaller the values of the input, the larger the value of the predicted value. Intuitively this makes sense: Smaller values of the input are more likely to have the value 1 because they are further from the place where the target switches from the value 1 to the value 0.

In practice, I have found this phenomenon true for problems with a business objective that calls for rank-ordering the population and acting on a subset of the scored records, such as fraud detection, customer acquisition, and customer attrition. In fact, for some problems, the linear regression model sometimes is the best model for selecting records at the top end of the rank-ordered list.

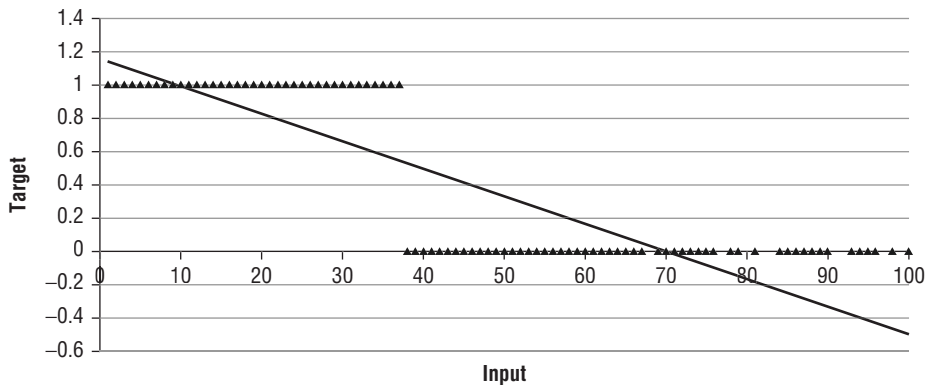

Figure 8-41: Using linear regression for classification

Other Regression Algorithms

Most classifications also have a regression form, including neural networks, decision trees, k-NN, and support vector machines. Regression Neural Networks are identical to classification neural networks except that output layer nodes usually have a linear activation function rather than the usual sigmoidal activation function. The linear activation function actually does nothing at all to the linearly weighted sum. They therefore behave very similarly to a linear regression function. Figure 8-42 shows how the linear activation function is represented in neural network pictures.

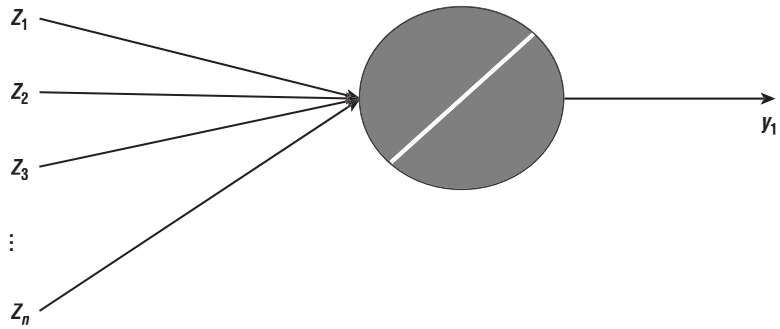

Figure 8-42: Linear activation function for regression output nodes

The remaining nodes in the neural network, including all the neurons in the hidden layer, still have the sigmoid activation functions. The neural network is still a very flexible nonlinear model. The advantage over linear regression is clear; neural networks can find nonlinear relationships between inputs and a continuous-valued target variable without the need to create derived variables.

Regression trees use the same decision tree form, but terminal nodes compute the mean or median value of the continuous target value. The ideal regression trees will create terminal nodes with small standard deviations of the target variable, and the mean or median values in the terminal nodes will be distinct from one another. Regression trees have the same advantages over numeric algorithms that classification trees have. They are insensitive to outliers, missing values, and strange distributions of inputs and the target variable.

Modifying the k-NN algorithm for regression is simple. Rather than voting to determine the value of a new data point, an average of the target value for nearest neighbors is computed. All other aspects of the k-NN algorithm are the same, including the principles for data preparation.

Summary

This chapter described the predictive modeling algorithms most commonly found in software, and therefore the algorithms most commonly used in practice. All of the predictive modeling algorithms described in this chapter fall under the category of supervised learning, which predicts one or more target variables in the data from a set of candidate input variables. Classification algorithms predict categorical target variables, and regression algorithms predict continuous target variables.

Each algorithm comes with its own set of assumptions about the data, and has its own strengths and weaknesses. I recommend that predictive modelers learn at least three different algorithms well enough to understand how to modify the parameters that change the way the models are built. Additionally, understanding the assumptions algorithms make about data helps you diagnose problems when they arise. This mindset develops as you build more models and experience models performing well and poorly.

CHAPTER

9

Assessing Predictive Models

This chapter explains how to assess model accuracy so that you can select the best model to deploy and have a good estimate of how well the model will perform operationally. In some ways, this should be a very short chapter. If you assess model accuracy by using percent correct classification for classification problems or average squared error for regression problems, your choices would be simple. However, because the choice of the model assessment metric should be tied to operational considerations rather than algorithmic expedience, you should keep several methods of model assessment in your toolbox.

Every algorithm has its own way of optimizing model parameters. Linear regression minimizes squared error. Neural networks minimize squared error or sometimes cross-entropy; k-NN minimizes the distance between data points. However, businesses often don't care about the root mean squared error, R-squared, the Gini Index, or entropy. Rather, the business may care about return on investment, expected profit, or minimizing false alarm rates for the next best 1,000 cases to be investigated. This chapter explores the most common ways models are assessed from a business perspective.

Model assessment should be done first on testing data to obtain an estimate of model accuracy for every model that has been built. Two decisions are made based on test data assessment: an assessment of which model has the best accuracy and therefore should be selected for deployment, and a conclusion if model accuracy is high enough for the best model to be used for deployment. The validation data can also be used for model assessment but usually only to

estimate the expected model accuracy and performance once it is deployed. Throughout discussions in the chapter, the assumption is that model assessment for selection purposes will be done using test data.

Batch Approach to Model Assessment

The first approach to assessing model accuracy is a *batch* approach, which means that all the records in the test or validation data are used to compute the accuracy without regard to the order of predictions in the data. A second approach based on a rank-ordered sorting of the predictions will be considered next. Throughout this chapter, the target variable for binary classification results will be shown as having 1s and 0s, although any two values can be used without loss of generality, such as "Y" and "N," or "good" and "bad."

Percent Correct Classification

The most straightforward assessment of classification models is percent correct classification (PCC). The PCC metric matches the predicted class value from a model with actual class value. When the predicted value matches the actual value, the record is counted as a correct classification, and when they don't match, the record is counted as an incorrect classification.

Consider a binary classification problem with 50 percent of the records labeled as 1 and the remaining 50 percent labeled as 0. The worst you can do with a predictive model built properly is random guessing, which can be simulated by flipping a fair coin and using heads as a guess for 1 and tails as a guess for 0; a random guess would be correct 50 percent of the time for binary classification. (It is possible for a model to predict worse than a random guess on testing data if it has been overfit on the training data, but we assume care has already been taken to avoid overfit.) A well-built classifier therefore will always have PCC on testing or validation data greater than or equal to 50 percent, and less than or equal to 100 percent.

The *lift* of a model is the ratio of model accuracy divided by the accuracy of a baseline measure, usually the expected performance of a random guess. If the proportion of 1s is 50 percent, the minimum lift of a model, if it does no better than a random guess, is 1 (50 percent ÷ 50 percent), and the maximum lift of a model, if it is a perfect predictor, is 2 (100 percent ÷ 50 percent).

What if the proportion of 1s is not 50 percent, but some smaller percentage of the data? The baseline PCC from a random guess will always be the proportion of 1s in the data, and lift is the ratio of PCC to this original proportion. If the proportion of 1s in the testing data is 5 percent, the baseline (random) expected performance of a classifier will be 5 percent correct classification, and the maximum lift of a model is now 20: 100 percent PCC ÷ 5 percent at random.

Therefore, the lower the baseline percentage of 1s in the target variable, the higher the potential lift that is achievable by a model. This method of calculating lift works for classification problems regardless of the number of levels in the target variable.

If there are more than two levels for the target variable, PCC is still computed the same way: the proportion of records classified correctly. However, the more levels in the target variable, the more difficult the classification problem is and the lower the random baseline accuracy is. For the three-class problem, if the proportion of each value is 33.3 percent, a random guess is also 33.3 percent and therefore the maximum lift is 3 instead of 2 for binary classification. In general, the maximum lift can differ by class if the proportion of each class value is different from the others. If, for example, class 1 is represented in 10 percent of the records, class 2 in 20 percent of the records, and class 3 in 70 percent of the records, you would expect that a random guess would estimate 10 percent of the records belong to class 1, 20 percent belong to class 2, and 70 percent belong to class 3. The maximum lift values for estimates of class 1, class 2, and class 3 are therefore 10 (100 percent ÷ 10 percent), 5 (100 percent ÷ 20 percent), and 1.4 (100 percent ÷ 70 percent).

Table 9-1 summarizes the differences between the possible lift compared to the baseline rate. It is immediately obvious that the smaller the baseline rate, the larger the possible lift. When comparing models by the lift metric, you must always be aware of the base rate to ensure models with a large baseline rate are not perceived as poor models because their lift is low, and conversely, models with very small baseline rates aren't considered good models even if the lift is high.

Table 9-1: Maximum Lift for Baseline Class Rates

BASELINE RATE OF CLASS VALUE	MAXIMUM LIFT
50 percent	2
33 percent	3
25 percent	4
10 percent	10
5 percent	20
1 percent	100
0.1 percent	1000

Table 9-2 shows 20 records for a binary classification problem. Each record contains the target variable, the model's predicted probability that the record belongs to class 1, and the model's predicted class label based on a probability threshold of 0.5. (If the probability is greater than 0.5, the predicted class label is assigned 1;

otherwise, it is assigned the value 0.) In this sample data, 13 of 20 records are classified correctly, resulting in a PCC of 65 percent.

Table 9-2: Sample Records with Actual and Predicted Class Values

ACTUAL TARGET VALUE	PROBABILITY TARGET = 1	PREDICTED TARGET VALUE	CONFUSION MATRIX QUADRANT
0	0.641	1	false alarm
1	0.601	1	true positive
0	0.587	1	false alarm
1	0.585	1	true positive
1	0.575	1	true positive
0	0.562	1	false alarm
0	0.531	1	false alarm
1	0.504	1	true positive
0	0.489	0	true negative
1	0.488	0	false dismissal
0	0.483	0	true negative
0	0.471	0	true negative
0	0.457	0	true negative
1	0.418	0	false dismissal
0	0.394	0	true negative
0	0.384	0	true negative
0	0.372	0	true negative
0	0.371	0	true negative
0	0.341	0	true negative
1	0.317	0	false dismissal

Confusion Matrices

For binary classification, the classifier can make an error in two ways: The model can predict a 1 when the actual value is a 0 (a *false* alarm), or the model can predict a 0 when the actual value is a 1 (a *false* dismissal). PCC provides a measure of classification accuracy regardless of the kind of errors: Either kind of error counts as an error.

A confusion matrix provides a more detailed breakdown of classification errors through computing a cross-tab of actual class values and predicted class values, thus revealing the two types of errors the model is making (false alarms and false dismissals) as well as the two ways the model is correct (true positives and true negatives), a total of four possible outcomes. In Figure 9-1, these four outcomes are listed in the column Confusion Matrix Quadrant.

The diagonal of the confusion matrix represents correct classification counts: true negatives, when the actual and predicted values are both 0, and true positives when both the actual and predicted values are both 1. The errors are on the off-diagonal, with false negatives in the lower-left quadrant when the actual value is 1 and predicted value is 0, and false positives in the upper-right quadrant when the actual value is 0 and the predicted value is 1. A perfect confusion matrix, therefore, has values equal to 0 in the off-diagonal quadrants, and non-zero values in the diagonal quadrants.

Once you divide the predictions into these four quadrants, several metrics can be computed, all of which provide additional insight into the kinds of errors the models are making. These metrics are used in pairs: sensitivity and specificity, type I and type II errors, precision and recall, and false alarms and false dismissals. Their definitions are shown in Table 9-3. Engineers often use the terms *false alarms* and *false dismissals*; statisticians often use *type I* and *type II* errors or *sensitivity* and *specificity*, particularly in medical applications; and computer scientists who do machine learning often use *precision* and *recall*, particularly in information retrieval. The table shows that the definitions of several of these measures are identical.

Some classifiers will excel at one type of accuracy at the expense of another, perhaps having a high sensitivity but at the expense of incurring false alarms (low specificity), or *vice versa*. If this is not a desirable outcome, the practitioner could rebuild the model, changing settings such as misclassification costs or case weights to reduce false alarms.

Confusion Matrix		Predicted Class		
		0 (predicted value is negative)	**1** (predicted value is positive)	Total Actual (down)
Actual Class	**0** (actual value is negative)	t_n (true negative)	f_p (false positive, false alarm)	Total actual negatives tn + fp
	1 (actual value is positive)	f_n (false negative, false dismissal)	t_p (true positive)	Total actual positives tp + fn
	Total Predicted (across)	Total negative predictions tn + fn	Total positive predictions tp + fp	Total Examples tp + tn + fp + fn

Figure 9-1: Confusion Matrix Components

Table 9-3: Confusion Matrix Measures

CONFUSION MATRIX MEASURES	WHAT IT MEASURES	QUADRANTS USED IN COMPUTATION
$PCC = \frac{t_p + t_n}{t_p + t_n + f_p + f_n}$	Overall accuracy	t_n f_p / f_n t_p
$False\ Alarm\ Rate\ (FA) = \frac{f_p}{t_n + f_p}$	Actual negative cases misclassified as positive	t_n f_p
$False\ Dismissal\ Rate\ (FD) = \frac{f_n}{t_p + f_n}$	Actual positive cases misclassified as negative	f_n t_p
$Precision = \frac{t_p}{t_p + f_p}$	Predicted positive cases classified correctly	f_p / t_p
$Recall = 1 - FD = \frac{t_p}{t_p + f_n}$	Actual positive cases classified correctly	f_n t_p
$Sensitivity = Recall = \frac{t_p}{t_p + f_n}$	Actual positive cases classified correctly	f_n t_p
$Specificity = True\ Negative\ Rate = t\_n / (t\_n + f\_p)$	Actual negative cases classified correctly	t_n f_p

$Type\ I\ Error\ Rate = FA = \frac{f_p}{t_n + f_p}$	Actual negative cases misclassified as positive — Incorrect rejection of a true null hypothesis	
$Type\ II\ Error\ Rate = FD = \frac{f_n}{t_p + f_n}$	Actual positive cases misclassified as negative — Failure to reject a false null hypothesis	

As an example of a confusion matrix, consider a logistic regression model built from the KDD Cup 1998 dataset and assessed on 47,706 records in a test set, with responders, coded as the value 1, comprising 5.1 percent of the population (2,418 of 47,706 records). The model was built from a stratified sample of 50 percent 0s and 50 percent 1s, and the confusion matrix represents the counts in each quadrant based on a predicted probability threshold of the model of 0.5. From these counts, any of the confusion matrix metrics listed in Table 9-3 could be computed, although you would rarely report all of them for any single project. Table 9-4 shows every one of these computed for a sample of 47,706 records. This model has similar false alarm and false dismissal rates (40 percent and 43.3 percent, respectively) because the predicted probability is centered approximately around the value 0.5, typical of models built from stratified samples.

The confusion matrix for this model is shown in Table 9-5. Rates from Table 9-4 are computed from the counts in the confusion matrix. For example, the false alarm rate, 40.0 percent is computed from true negatives and false positives, 18,110 ÷ (27,178 + 18,110).

A second model, built from the natural proportion (not stratified), has a confusion matrix for test data in Table 9-6. Once again, the confusion matrix is created by using a predicted probability threshold of 0.5, although because the model was trained on a population with only 5.06 percent responders, only two of the predictions exceeded 0.5. The second model appears to have higher PCC, but only because the model prediction labels are nearly always 0. Be careful of the prior probability of the target variable values and the threshold that is used in building the confusion matrix. Every software package defaults to equal probabilities as the threshold for building a confusion matrix (0.5 for binary classification), regardless of the prior probability of the target variable.

If you threshold the model probabilities by 0.0506 rather than 0.5, the resulting confusion matrix is nearly identical to the one shown in Table 9-5.

Table 9-4: Comparison Confusion Matrix Metrics for Two Models

METRIC	RATES, MODEL 1	RATES, MODEL 2
PCC	59.8 percent	94.9 percent
FA	40.0 percent	0.0 percent
FD	43.3 percent	100.0 percent
Precision	7.0 percent	0.0 percent
Recall	56.7 percent	0.0 percent
Sensitivity	56.7 percent	0.0 percent
Specificity	60.0 percent	100.0 percent
Type I Rate	40.0 percent	0.0 percent
Type II Rate	43.3 percent	100.0 percent

Table 9-5: Confusion Matrix for Model 1

CONFUSION MATRIX, MODEL 1	0	1	TOTAL
0	27,178	18,110	45,288
1	1,047	1,371	2,418
Total	28,225	19,481	47,706

Table 9-6: Confusion Matrix for Model 2

CONFUSION MATRIX, MODEL 2	0	1	TOTAL
0	45,286	2	45,288
1	2,418	0	2,418
Total	47,704	2	47,706

Beware of comparing individual metrics by themselves. One model may appear to have a low precision in comparison to a second model, but it may only be because the second model has very few records classified as 1, and therefore,

while the false alarm rate is lower, the true positive rate is also much smaller. For example, precision from Table 9-6 is 0 percent because no records were predicted to be responders and were actually value responders. In Table 9-5, precision was equal to 7.0 percent. However, the overall classification accuracy for the model in Table 9-6 is 94.9 percent (45,286 + 0) ÷ 47,706, much larger than the 59.8 percent PCC for Table 9-5.

Confusion Matrices for Multi-Class Classification

If there are more than two levels in the target variable, you can still build a confusion matrix, although the confusion matrix measures listed in Table 9-3 don't apply. Table 9-7 shows a confusion matrix for the seven-level target variable in the nasadata dataset, with the classes re-ordered to show the misclassifications more clearly. The actual values are shown in rows, and the predictive values in the columns. Note that for this classifier there are three groupings of classification decisions: alfalfa and clover form one group; corn, oats, and soy form a second group; and rye and wheat form a third group.

Table 9-7: Multi-Class Classification Confusion Matrix

MULTI-CLASS CONFUSION MATRIX	ALFALFA	CLOVER	CORN	OATS	SOY	RYE	WHEAT
Alfalfa	28	27	0	0	0	0	0
Clover	5	57	0	0	0	0	0
Corn	0	0	50	7	4	0	0
Oats	0	0	4	59	1	0	0
Soy	0	0	3	2	56	0	0
Rye	0	0	0	0	1	53	7
Wheat	0	0	0	0	1	15	44

ROC Curves

Receiver Operating Characteristic (ROC) curves provide a visual trade-off between false positives on the x-axis and true positives on the y-axis. They are essentially, therefore, a visualization of confusion matrices, except that rather than plotting a single confusion matrix, *all* confusion matrices are plotted. Each true positive/false positive pair is found by applying a different threshold on the predicted probability, ranging from 0 to 1. A sample ROC curve is shown in Figure 9-2.

At the extremes, a threshold of 1 means that you only predict 1 for the model if the predicted probability exceeds 1. Because the maximum predicted probability

is 1, the threshold is never exceeded so you never have any false alarms (the false alarm rate is 0). Likewise, you also don't have any actual target values of 1 classified correctly (the sensitivity is also 0). This is the bottom left of the ROC curve. At the other extreme, if you threshold the predictions at 0, every record is predicted to have the value 1, so the sensitivity and false alarm rates are both 1, the upper right of the ROC curve.

The interesting part of the ROC curve is what happens in between these extremes. The higher the sensitivity for low false alarm rates, the steeper the vertical rise you see at the left end of the ROC curve; a perfect ROC curve has a vertical line for sensitivity up to value 1 at the false alarm rate equal to 0, and then is horizontal at the sensitivity value equal to 1 for all values of the false alarm rate. In the sample ROC curve, thresholds between 0.2 and 0.8 at uniform intervals of 0.1 are shown. The distance between the thresholds is not uniform in this ROC curve; the distance depends on the distribution of probabilities.

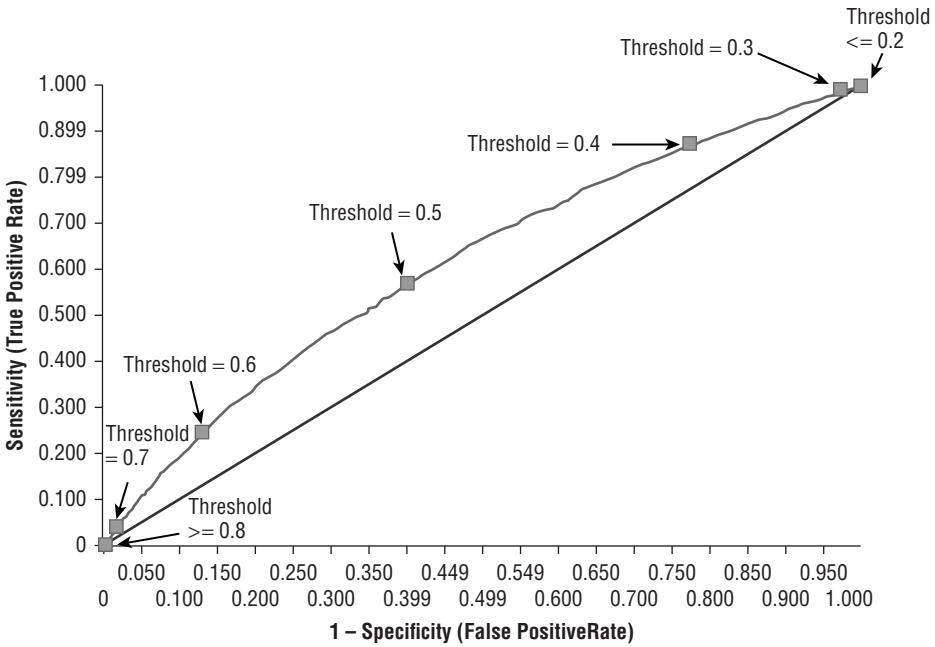

Figure 9-2: Sample ROC curve

One of the most commonly used single-number metrics to compare classification models is Area under the Curve (AUC), usually referring to the area under the ROC curve. For a ROC curve with x- and y-axes containing rates rather than record counts, a perfect model will have AUC equal to 1. A random model will have AUC equal to 0.5 and is represented by the diagonal line between the extremes: coordinates (0,0) and (1,1). A larger AUC value can be achieved by

the ROC curve stretching to the upper left of the ROC curve, meaning that the false alarm rate does not increase as quickly as the sensitivity rate compared to a random selection of records. The AUC for Figure 9-2 is 0.614.

Often, practitioners will display the ROC curves for several models in a single plot, showing visually the differences in how the models trade off false positives and true positives. In the sample shown in Figure 9-3, Model 2 is the best choice if you require false alarm rates to be less than 0.4. However, the sensitivity is still below 0.6. If you can tolerate higher false alarm rates, Model 3 is the better choice because its sensitivity is highest.

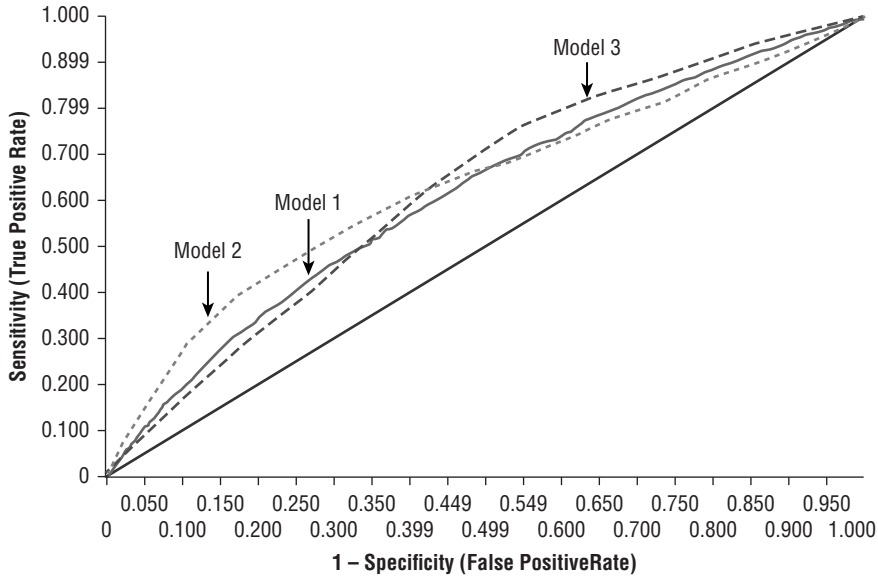

Figure 9-3: Comparison of three models

Rank-Ordered Approach to Model Assessment

In contrast to batch approaches to computing model accuracy, rank-ordered metrics begin by sorting the numeric output of the predictive model, either the probability or confidence of a classification model or the actual predicted output of a regression model. The rank-ordered predictions are binned into segments, and summary statistics related to the accuracy of the model are computed either individually for each segment, or cumulatively as you traverse the sorted file list. The most common segment is the *decile*, with10 percent of the dataset's records in each segment. Other segment sizes used commonly include 5 percent groups, called vinigtiles (also called *vingtiles*, *twentiles*, or *demi-deciles* in software) and 1 percent groups called *percentiles*. Model results in this section are shown with vinigtiles, although you could use any of the segment sizes without changing the principles.

Rank-ordered approaches work particularly well; the model will identify a subset of the scored records to act on. In marketing applications, you treat those who are most likely to respond to the treatment, whether the treatment is a piece of mail, an e-mail, a display ad, or a phone call. For fraud detection, you may want to select the cases that are the most suspicious or non-compliant because there are funds to treat only a small subset of the cases. In these kinds of applications, so long as the model performs well on the selected population, you don't care how well it rank orders the remaining population because you will never do anything with it.

The three most common rank-ordered error metrics are *gains charts*, *lift charts*, and *ROI charts*. In each of these charts, the x-axis is the percent depth of the rank-ordered list of probabilities, and the y-axis is the gain, lift, or ROI produced by the model at that depth.

Gains and Lift Charts

Gain refers to the percentage of the class value of interest found cumulatively in the rank-ordered list at each file depth. Without loss of generality, assume the value of interest is a 1 for a binary classification problem. Figure 9-4 shows a typical gains chart, where the upper curve represents the gain due to the model. This particular dataset has the target variable value of interest represented in 5.1 percent of the data, although this rate is not known from the gains chart itself. The contrasting straight line is the expected gain due to a random draw from the data. For a random draw, you would expect to find 10 percent of the 1s in the first 10 percent of the data, 20 percent in the first 20 percent of the records, and so on: a linear gain. The area between the random line and the model gain is the incremental improvement the model provides. The figure shows through the 10 percent depth, the model selects nearly 20 percent of the 1s, nearly twice as many as a random draw.

The depth you use to measure a model's gain depends on the business objective: It could be gain at the 10 percent depth, 30 percent depth, 70 percent depth, or even the area between the model gain and the random gain over the entire dataset, much like the AUC metric computed from the ROC curve. With a specific value of gain determined by the business objective, the predictive modeler can build several models and select the model with the best gain value.

If a model is perfect, with 5.1 percent of the data with a target value equal to 1, the first model vinigtile will be completely populated with 1s, the remaining 1s falling in the second vinigtile. Because the remainder of the data does not have any more 1s, the remaining values for the gains chart are flat at 100 percent. A perfect gains chart is shown in Figure 9-5. If you ever see a perfect gains chart, you've almost certainly allowed an input variable into the model that is

incorporating target variable information. Find the variable that is most important to the model and correct it or remove it from the list of candidate inputs.

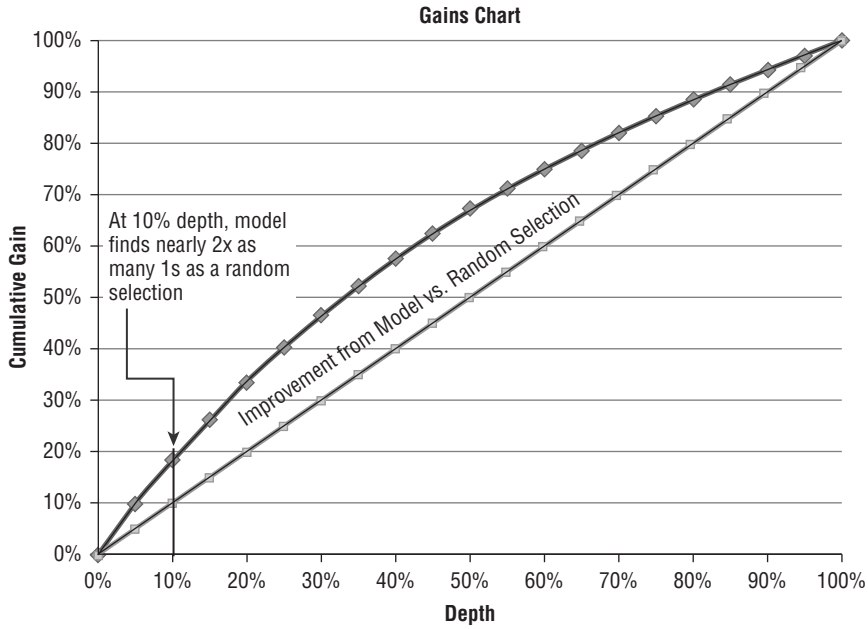

Figure 9-4: Sample gains chart

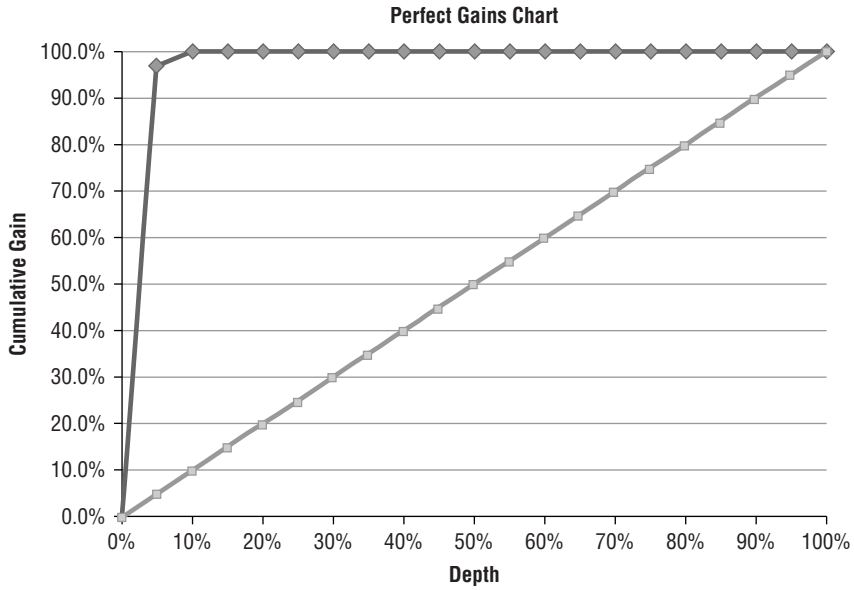

Figure 9-5: Perfect gains chart

Lift is closely related to gain, but rather than computing the percentage of 1s found in the rank ordered list, we compute the *ratio* between the 1s found by the model and the 1s that would have been found with a random selection at the same depth. Figure 9-6 shows a cumulative lift chart for the same data used for the gains chart in Figure 9-4. In a lift chart, the random selection line has a lift of 1.0 through the population. The cumulative lift chart will always converge to 1.0 at the 100 percent depth. Figure 9-7 shows the segment lift chart, the lift for each segment (vinigtiles in this chart). When computing lift per segment, the segment lift must eventually descend below the lift of 1.0, typically halfway through the file depth. One advantage of seeing the segment lift is that you can also see how few 1s are still selected by the model in the bottom segments. For Figure 9-7, the lift is just above 0.5.

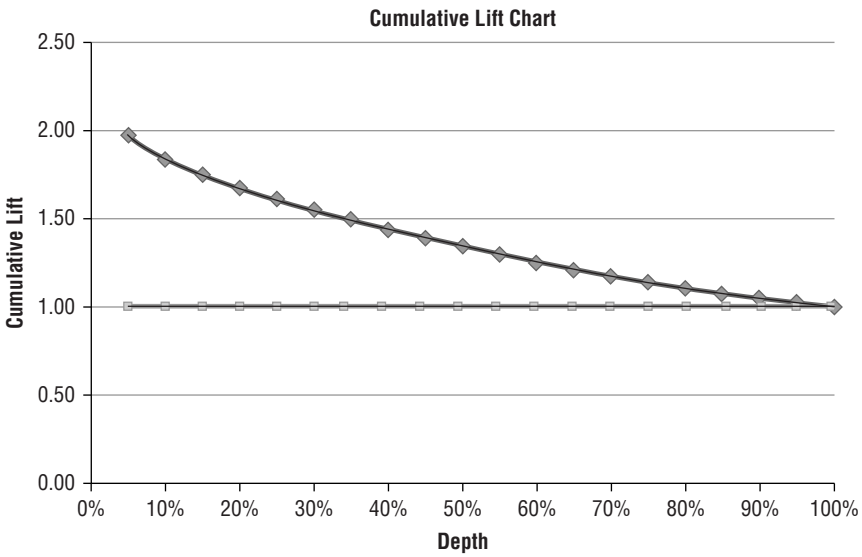

Figure 9-6: Sample cumulative lift chart

Sometimes it is useful to measure model lift from segment to segment rather than cumulatively. Cumulative lift charts aggregate records as the depth increases and can mask problems in the models, particularly at the higher model depths. Consider Figures 9-8 and 9-9, which show the cumulative lift and segment lift for a different model than was used for the lift charts in Figures 9-6 and 9-7. The cumulative lift chart is worse than the prior model but has a smooth appearance from vinigtile to vinigtile. However, in Figure 9-9, the segment lift chart is erratic, with positive and negative slopes from vinigtile to vinigtile. Erratic segment lift

charts created from testing or validation data are indicative of models that are overfit: A stable model will have monotonic lift values from segment to segment.

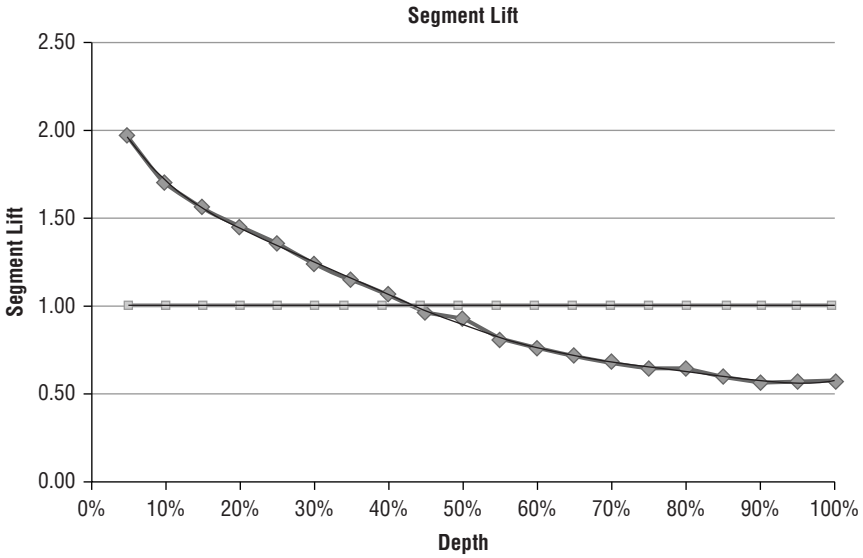

Figure 9-7: Sample segment lift chart

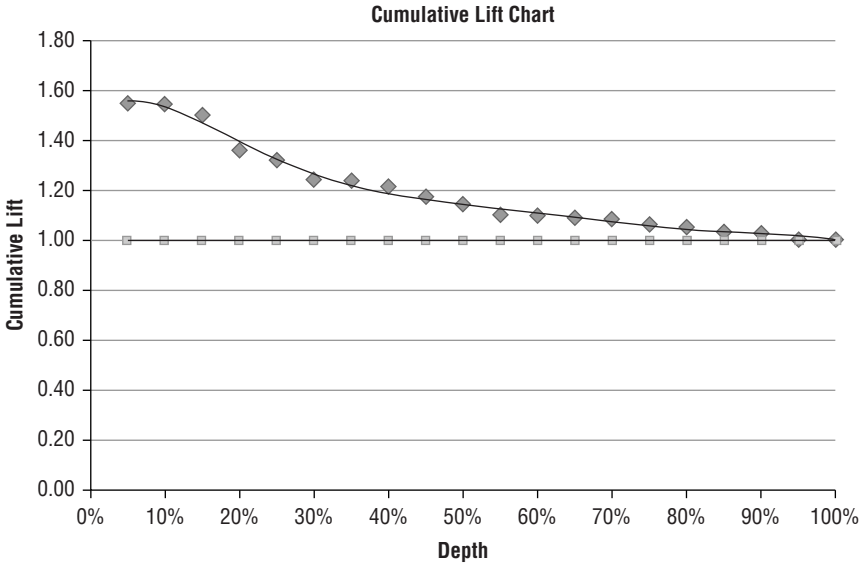

Figure 9-8: Cumulative lift chart for overfit model

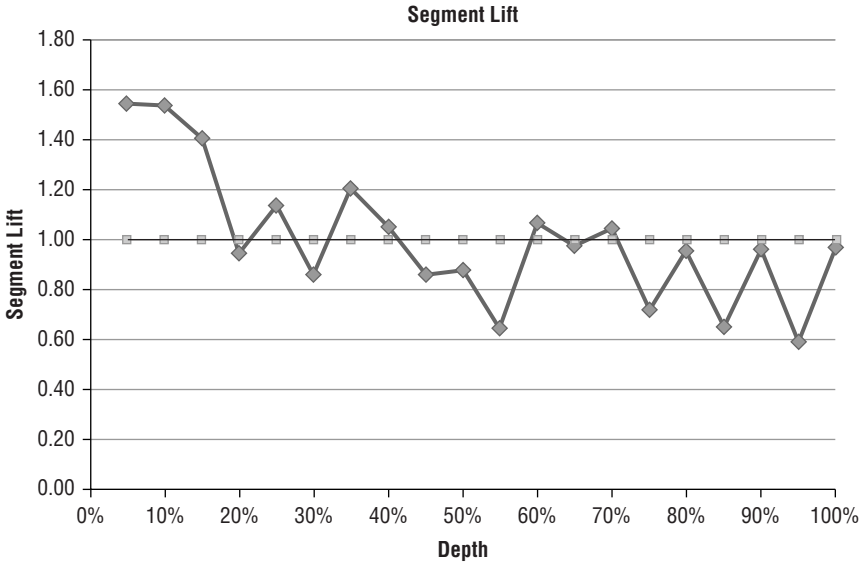

Figure 9-9: Segment lift chart for overfit model

When there are multiple levels in the target variable—multi-class classification problems—gains and lift charts can still be computed, but for only one value of the target variable at a time. For example, with the nasadata, you would build a separate lift chart for each of the seven classes.

Custom Model Assessment

When you use a rank-ordered metric to assess model performance, the actual value predicted by the model no longer matters: Only the metric matters. For gains, the metric is the percentage of 1s found by the model. For lift, the metric is the ratio of the percentage of 1s found to the average rate. For ROC, it is the comparison of true alerts to false alarms. In each of these cases, each record has equal weight. However, some projects can benefit from weighting records according to their costs or benefits, creating a custom assessment formula.

For example, one popular variation on rank-ordered methods is computing the expected cumulative profit: a fixed or variable gain minus a fixed or variable cost. Fixed cost and gain is a straightforward refinement of a gains chart, and many software tools include profit charts already. For example, the variable gain in a marketing campaign can be the purchase amount of a new customer's first visit, and the variable cost the search keyword ad cost. Or for a fraud detection

problem, the variable gain can be the amount of fraud perpetrated by the offender and the variable cost the investigation costs.

Consider an application like the KDD Cup 1998 data that has fixed cost (the cost of the mailing) and variable gain (the donation amount). Figure 9-10 shows the profit chart for this scenario. This particular model generates a maximum profit of nearly $2,500 at the depth of 25,000 donors. This maximum profit is nearly two times as much compared to contacting all the donors (depth of 43,000).

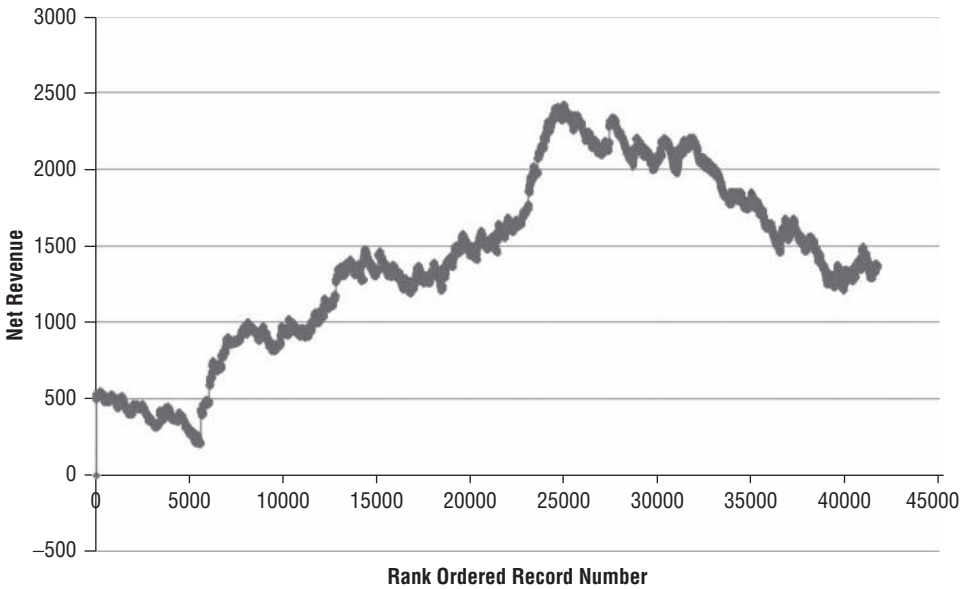

Figure 9-10: Profit chart

Custom model assessments can be applied to regression models just as easily as classification models precisely because it is only the rank-ordering that matters, not the actual prediction value. By extension, this also means that completely different models, built for different target variables, can be assessed and compared directly, on a common scale.

In addition, custom model assessments can also be applied to batch metrics, adding to the list of confusion matrix metrics already listed in Table 9-3. For example, consider a business objective that stipulates false alarms have four times the cost of false dismissals. Table 9-8 shows the same data already shown in Table 9-2 with the additional column showing the cost of an error. Correct classifications receive a cost of 0, false dismissals a cost of 1, and false alarms a cost of 4. The sum of these costs is the model score and the best model is the lowest-scoring model for the cost function.

Table 9-8: Custom Cost Function for False Alarms and False Dismissals

ACTUAL TARGET VALUE	PROBABILITY TARGET = 1	PREDICTED TARGET VALUE	CONFUSION MATRIX QUADRANT	COST OF ERROR, FALSE ALARM 4X
0	0.641	1	false alarm	4
1	0.601	1	true positive	0
0	0.587	1	false alarm	4
1	0.585	1	true positive	0
1	0.575	1	true positive	0
0	0.562	1	false alarm	4
0	0.531	1	false alarm	4
1	0.504	1	true positive	0
0	0.489	0	true negative	0
1	0.488	0	false dismissal	1
0	0.483	0	true negative	0
0	0.471	0	true negative	0
0	0.457	0	true negative	0
1	0.418	0	false dismissal	1
0	0.394	0	true negative	0
0	0.384	0	true negative	0
0	0.372	0	true negative	0
0	0.371	0	true negative	0
0	0.341	0	true negative	0
1	0.317	0	false dismissal	1

Which Assessment Should Be Used?

In general, the assessment used should be the one that most closely matches the business objectives defined at the beginning of the project during Business Understanding. If that objective indicates that the model will be used to select one-third of the population for treatment, then model gain or lift at the 33 percent depth is appropriate. If all customers will be treated, then computing AUC for a batch metric may be appropriate. If the objective is to maximize the records selected by the model subject to a maximum false alarm rate, a ROC is appropriate.

The metric used for model selection is of critical importance because the model selected based on one metric may not be a good model for a different metric.

Consider Figure 9-11: a scatterplot of 200 models and their rank based on AUC at the 70 percent depth and the root mean squared (RMS) error. The correlation between these two rankings is 0.1—almost no relationship between the two rankings; the ranking based on AUC cannot be determined at all from the ranking of RMS error. Therefore, if you want to use a model operationally in an environment where minimizing false positives and maximizing true positives is important, choosing the model with the best RMS error model would be sub-optimal.

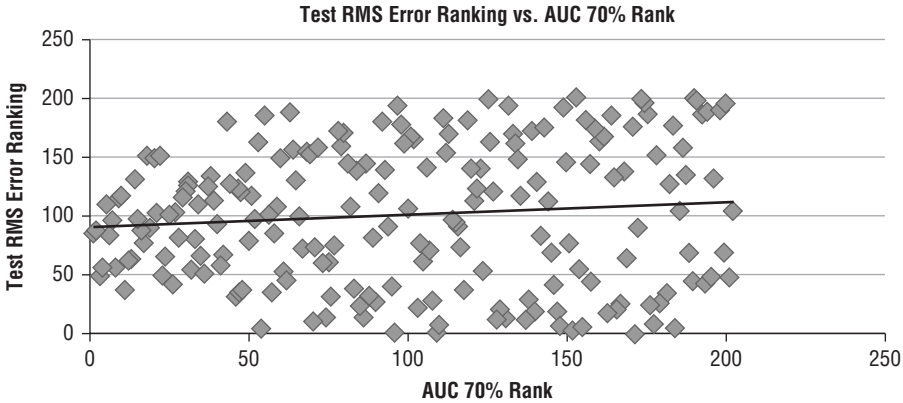

Figure 9-11: Scatterplot of AUC vs. RMS Error

Assessing Regression Models

The metrics most predictive analytics software packages provide to assess regression models are batch methods computed for the data partition you are using to assess the models (training, testing, validation, and so on). The most commonly used metric is the coefficient of determination known as R^2, pronounced "r squared." R^2 measures the percentage of the variance of the target variable that is explained by the models. You can compute it by subtracting the ratio of the variance of the residuals and the variance of the target variable from 1. If the variance of the residuals is 0, meaning the model fit is perfect, R^2 is equal to 1, indicating a perfect fit. If the model explains none of the variance in the model, the variance of the residuals will be just as large as the variance of the target variable, and R^2 will be equal to 0. You see R^2 even in an Excel trend line equation.

What value of R^2 is good? This depends on the application. In social science problems, an R^2 of 0.3 might be excellent, whereas in scientific applications you might need an R^2 value of 0.7 or higher for the model to be considered a good fit. Some software packages also include a modification of R^2 that penalizes

the model as more terms are added to the model, much like what is done with AIC, BIC, and MDL.

Figure 9-12 shows two regression models, the first with a relatively high R^2 value and the second with a nearly zero R^2 value. In the plot at the left, there is a clear trend between LASTGIFT and TARGET_D. At the right, the relationship between the variables is nearly random; there is very little explained variance from the linear model.

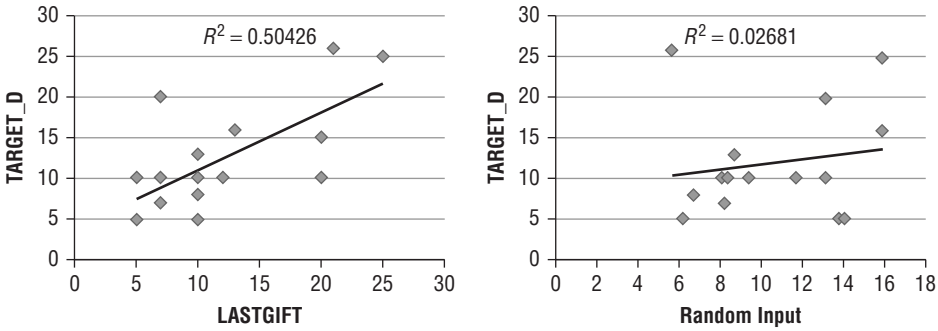

Figure 9-12: R^2 for two linear models

R^2 is only one of the many ways regression models are assessed. Table 9-9 shows a list of other common measures of model accuracy. Even if these are not included in your predictive modeling software, they are easy to compute.

Table 9-9: Batch Metrics for Assessing Regression Models

MEASURE	EXPLANATION OF MEASURE
R^2	Percent variance explained
Mean squared error (MSE)	Average error. If the model predictions are unbiased positively or negatively, this should be equal to 0.
Mean absolute error (MAE)	Compute the absolute value of the error before averaging. This provides an average magnitude of error measure and is usually preferred over MSE for comparing models.
Median error	A more robust measure of errors than MSE because outliers in error don't influence the median
Median absolute error	A more robust measure of error magnitudes
Correlation	The square root of R^2, this is an intuitive way to assess how similar model predictions are to the target values. Correlations are sensitive to outliers.

MEASURE	EXPLANATION OF MEASURE
Average percent error	Compute the percentage of the error rather than the mean. This normalizes by the magnitude of the target variable, showing the relative size of the error compared to the actual target value.
Average absolute percent error	Compute the percentage of the absolute value of the error rather than the mean. As with average percent error, this shows the relative size of the error.

The error metric you use depends on your business objective. If you care about total errors, and larger errors are more important than smaller errors, mean absolute error is a good metric. If the relative size of the error is more important than the magnitude of the error, the percent error metrics are more appropriate.

Table 9-10 shows several of the error metrics for the KDD Cup 1998 data using four regression models, each predicting the target variable TARGET_D. The linear regression model has the largest R^2 in this set of models. Interestingly, even though the linear regression model had the highest R^2 and lowest mean absolute error, the regression trees had lower median absolute error values, meaning that the most typical error magnitude is smaller for the trees.

Table 9-10: Regression Error Metrics for Four Models

MEASURE	LINEAR REGRESSION	NEURAL NETWORK	CART REGRESSION TREE	CHAID REGRESSION TREE
R^2	0.519	0.494	0.503	0.455
Mean Error	–0.072	–0.220	–0.027	0.007
Mean Absolute Error	4.182	4.572	4.266	4.388
Median Absolute Error	2.374	3.115	2.249	2.276
Correlation	0.720	0.703	0.709	0.674

Rank-ordered model assessment methods apply to regression problems in the same way they apply to classification problems, even though they often are not included in predictive analytics software. Rank-ordered methods change the focus of model assessment from a record-level error calculation to the ranking, where the accuracy itself matters less than getting the predictions in the right order.

For example, if you are building customer lifetime value (CLV) models, you may determine that the actual CLV value is not going to be particularly accurate. However, identifying the customers with the most potential, the top 10 percent of the CLV scores, may be strategically valuable for the company. Figure 9-11 showed that batch assessment measures might not be correlated with rank-ordered assessment methods for classification. The same applies to regression.

For example, Table 9-11 shows a decile-by-decile summary of two models built on the KDD Cup 1998 data. The top decile for the linear regression model finds donors who gave on average 32.6 dollars. The average for donors who gave a donation was 15.4, so the top decile of linear regression scores found gift amounts with a lift of 2.1 (32.6 ÷ 15.4).

Table 9-11: Rank-Ordering Regression Models by Decile

DECILE	LINEAR REGRESSION, MEAN TARGET_D	NEURAL NETWORK, MEAN TARGET_D
1	32.6	32.3
2	22.1	21.4
3	18.9	19.1
4	17.1	16.8
5	15.3	15.0
6	13.2	13.1
7	11.3	11.2
8	10.5	10.2
9	7.6	7.8
10	6.0	6.6

The deciles are shown in tabular form, but they can also be shown in graphical form, as in the chart in Figure 9-13. The figure shows that the model predicts the highest donors in the top decile particularly well; the average TARGET_D values for the top decile is larger than you expect based on the decreasing trend of the remaining deciles.

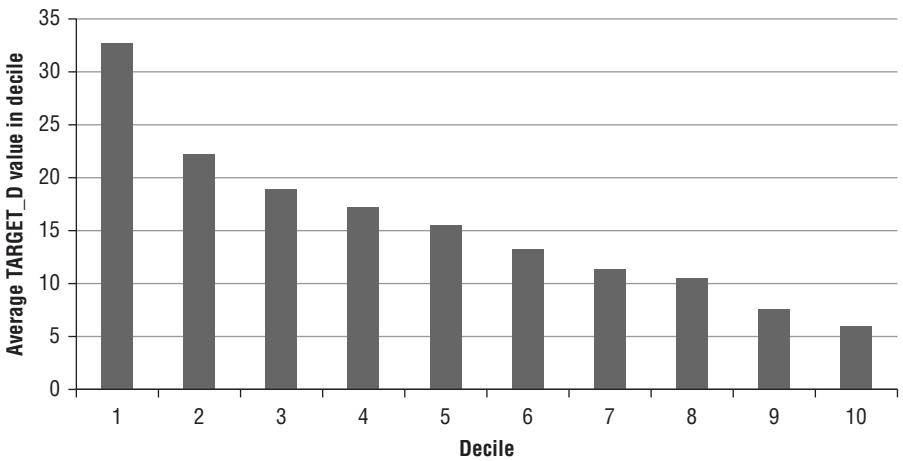

Figure 9-13: Average actual target value by decile