chapter 7

Transformations:

Engineering the input and output

In the previous chapter we examined a vast array of machine learning methods: decision trees, decision rules, linear models, instance-based schemes, numeric prediction techniques, clustering algorithms, and Bayesian networks. All are sound, robust techniques that are eminently applicable to practical data mining problems.

But successful data mining involves far more than selecting a learning algorithm and running it over your data. For one thing, many learning methods have various parameters, and suitable values must be chosen for these. In most cases, results can be improved markedly by suitable choice of parameter values, and the appropriate choice depends on the data at hand. For example, decision trees can be pruned or unpruned, and in the former case a pruning parameter may have to be chosen. In the *k*-nearest-neighbor method of instance-based learning, a value for *k* will have to be chosen. More generally, the learning scheme itself will have to be chosen from the range of schemes that are available. In all cases, the right choices depend on the data itself.

It is tempting to try out several learning schemes, and several parameter values, on your data and see which works best. But be careful! The best choice is not necessarily the one that performs best on the training data. We have repeatedly cautioned about the problem of overfitting, where a learned model is too closely tied to the particular training data from which it was built. It is incorrect to assume that performance on the training data faithfully represents the level of performance that can be expected on the fresh data to which the learned model will be applied in practice.

Fortunately, we have already encountered the solution to this problem in Chapter 5. There are two good methods for estimating the expected true performance of a learning scheme: the use of a large dataset that is quite separate from the training data, in the case of plentiful data, and cross-validation (Section 5.3), if data is scarce. In the latter case, a single 10-fold cross-validation is typically used in practice, although to obtain a more reliable estimate the entire procedure should be repeated 10 times. Once suitable parameters have been chosen for the learning scheme, use the whole training set—all the available training instances—to produce the final learned model that is to be applied to fresh data.

Note that the performance obtained with the chosen parameter value during the tuning process is *not* a reliable estimate of the final model's performance, because the final model potentially overfits the data that was used for tuning. To ascertain how well it will perform, you need yet another large dataset that is quite separate from any data used during learning and tuning. The same is true for cross-validation: you need an "inner" cross-validation for parameter tuning and an "outer" cross-validation for error estimation. With 10-fold cross-validation, this involves running the learning scheme 100 times. To summarize: when assessing the performance of a learning scheme, any parameter tuning that goes on should be treated as though it were an integral part of the training process.

There are other important processes that can materially improve success when applying machine learning techniques to practical data mining problems, and these are the subject of this chapter. They constitute a kind of data engineering: engineering the input data into a form suitable for the learning scheme chosen and engineering the output model to make it more effective. You can look on them as a bag of tricks that you can apply to practical data mining problems to enhance the chance of success. Sometimes they work; other times they don't—and at the present state of the art, it's hard to say in advance whether they will or not. In an area such as this where trial and error is the most reliable guide, it is particularly important to be resourceful and understand what the tricks are.

We begin by examining four different ways in which the input can be massaged to make it more amenable for learning methods: attribute selection, attribute discretization, data transformation, and data cleansing. Consider the first, attribute selection. In many practical situations there are far too many attributes for learning schemes to handle, and some of them—perhaps the overwhelming majority—are clearly irrelevant or redundant. Consequently, the data must be preprocessed to select a subset of the attributes to use in learning. Of course, learning methods themselves try to select attributes appropriately and ignore irrelevant or redundant ones, but in practice their performance can frequently be improved by preselection. For example, experiments show that adding useless attributes causes the performance of learning schemes such as decision trees and rules, linear regression, instance-based learners, and clustering methods to deteriorate.

Discretization of numeric attributes is absolutely essential if the task involves numeric attributes but the chosen learning method can only handle categorical ones. Even methods that can handle numeric attributes often produce better results, or work faster, if the attributes are prediscretized. The converse situation, in which categorical attributes must be represented numerically, also occurs (although less often); and we describe techniques for this case, too.

Data transformation covers a variety of techniques. One transformation, which we have encountered before when looking at relational data in Chapter 2 and support vector machines in Chapter 6, is to add new, synthetic attributes whose purpose is to present existing information in a form that is suitable for the machine learning scheme to pick up on. More general techniques that do not depend so intimately on the semantics of the particular data mining problem at hand include principal components analysis and random projections.

Unclean data plagues data mining. We emphasized in Chapter 2 the necessity of getting to know your data: understanding the meaning of all the different attributes, the conventions used in coding them, the significance of missing values and duplicate data, measurement noise, typographical errors, and the presence of systematic errors—even deliberate ones. Various simple visualizations often help with this task. There are also automatic methods of cleansing data, of detecting outliers, and of spotting anomalies, which we describe.

Having studied how to massage the input, we turn to the question of engineering the output from machine learning schemes. In particular, we examine techniques for combining different models learned from the data. There are some surprises in store. For example, it is often advantageous to take the training data and derive several different training sets from it, learn a model from each, and combine the resulting models! Indeed, techniques for doing this can be very powerful. It is, for example, possible to transform a relatively weak learning method into an extremely strong one (in a precise sense that we will explain). Moreover, if several learning schemes are available, it may be advantageous not to choose the best-performing one for your dataset (using cross-validation) but to use them all and combine the results. Finally, the standard, obvious way of modeling a multiclass learning situation as a two-class one can be improved using a simple but subtle technique.

Many of these results are counterintuitive, at least at first blush. How can it be a good idea to use many different models together? How can you possibly do better than choose the model that performs best? Surely all this runs counter to Occam's razor, which advocates simplicity. How can you possibly obtain first-class performance by combining indifferent models, as one of these techniques appears to do? But consider committees of humans, which often come up with wiser decisions than individual experts. Recall Epicurus's view that, faced with alternative explanations, one should retain them all. Imagine a group of specialists each of whom excels in a limited domain even though none is competent across the board. In struggling to understand how these methods work, researchers have exposed all sorts of connections and links that have led to even greater improvements.

Another extraordinary fact is that classification performance can often be improved by the addition of a substantial amount of data that is *unlabeled,* in other words, the class values are unknown. Again, this seems to fly directly in the face of common sense, rather like a river flowing uphill or a perpetual motion machine. But if it were true—and it is, as we will show you in Section 7.6—it would have great practical importance because there are many situations in which labeled data is scarce but unlabeled data is plentiful. Read on—and prepare to be surprised.

7.1 Attribute selection

Most machine learning algorithms are designed to learn which are the most appropriate attributes to use for making their decisions. For example, decision tree methods choose the most promising attribute to split on at each point and should—in theory—never select irrelevant or unhelpful attributes. Having more features should surely—in theory—result in more discriminating power, never less. "What's the difference between theory and practice?" an old question asks. "There is no difference," the answer goes, "—in theory. But in practice, there is." Here there is, too: in practice, adding irrelevant or distracting attributes to a dataset often "confuses" machine learning systems.

Experiments with a decision tree learner (C4.5) have shown that adding to standard datasets a random binary attribute generated by tossing an unbiased coin affects classification performance, causing it to deteriorate (typically by 5% to 10% in the situations tested). This happens because at some point in the trees that are learned the irrelevant attribute is invariably chosen to branch on, causing random errors when test data is processed. How can this be, when decision tree learners are cleverly designed to choose the best attribute for splitting at each node? The reason is subtle. As you proceed further down the tree, less and less data is available to help make the selection decision. At some point, with little data, the random attribute will look good just by chance. Because the number of nodes at each level increases exponentially with depth, the chance of the rogue attribute looking good somewhere along the frontier multiplies up as the tree deepens. The real problem is that you inevitably reach depths at which only a small amount of data is available for attribute selection. If the dataset were bigger it wouldn't necessarily help—you'd probably just go deeper.

Divide-and-conquer tree learners and separate-and-conquer rule learners both suffer from this effect because they inexorably reduce the amount of data on which they base judgments. Instance-based learners are very susceptible to irrelevant attributes because they always work in local neighborhoods, taking just a few training instances into account for each decision. Indeed, it has been shown that the number of training instances needed to produce a predetermined level of performance for instance-based learning increases exponentially with the number of irrelevant attributes present. Naïve Bayes, by contrast, does not fragment the instance space and robustly ignores irrelevant attributes. It assumes by design that all attributes are independent of one another, an assumption that is just right for random "distracter" attributes. But through this very same assumption, Naïve Bayes pays a heavy price in other ways because its operation is damaged by adding redundant attributes.

The fact that irrelevant distracters degrade the performance of state-of-the-art decision tree and rule learners is, at first, surprising. Even more surprising is that *relevant* attributes can also be harmful. For example, suppose that in a two-class dataset a new attribute were added which had the same value as the class to be predicted most of the time (65%) and the opposite value the rest of the time, randomly distributed among the instances. Experiments with standard datasets have shown that this can cause classification accuracy to deteriorate (by 1% to 5% in the situations tested). The problem is that the new attribute is (naturally) chosen for splitting high up in the tree. This has the effect of fragmenting the set of instances available at the nodes below so that other choices are based on sparser data.

Because of the negative effect of irrelevant attributes on most machine learning schemes, it is common to precede learning with an attribute selection stage that strives to eliminate all but the most relevant attributes. The best way to select relevant attributes is manually, based on a deep understanding of the learning problem and what the attributes actually mean. However, automatic methods can also be useful. Reducing the dimensionality of the data by deleting unsuitable attributes improves the performance of learning algorithms. It also speeds them up, although this may be outweighed by the computation involved in attribute selection. More importantly, dimensionality reduction yields a more compact, more easily interpretable representation of the target concept, focusing the user's attention on the most relevant variables.

Scheme-independent selection

When selecting a good attribute subset, there are two fundamentally different approaches. One is to make an independent assessment based on general characteristics of the data; the other is to evaluate the subset using the machine learning algorithm that will ultimately be employed for learning. The first is called the *filter* method, because the attribute set is filtered to produce the most promising subset before learning commences. The second is the *wrapper* method, because the learning algorithm is wrapped into the selection procedure. Making an independent assessment of an attribute subset would be easy if there were a good way of determining when an attribute was relevant to choosing the class. However, there is no universally accepted measure of "relevance," although several different ones have been proposed.

One simple scheme-independent method of attribute selection is to use just enough attributes to divide up the instance space in a way that separates all the training instances. For example, if just one or two attributes are used, there will generally be several instances that have the same combination of attribute values. At the other extreme, the full set of attributes will likely distinguish the instances uniquely so that no two instances have the same values for all attributes. (This will not necessarily be the case, however; datasets sometimes contain instances with the same attribute values but different classes.) It makes intuitive sense to select the smallest attribute subset that distinguishes all instances uniquely. This can easily be found using exhaustive search, although at considerable computational expense. Unfortunately, this strong bias toward consistency of the attribute set on the training data is statistically unwarranted and can lead to overfitting—the algorithm may go to unnecessary lengths to repair an inconsistency that was in fact merely caused by noise.

Machine learning algorithms can be used for attribute selection. For instance, you might first apply a decision tree algorithm to the full dataset, and then select only those attributes that are actually used in the tree. Although this selection would have no effect at all if the second stage merely built another tree, it will have an effect on a different learning algorithm. For example, the nearest-neighbor algorithm is notoriously susceptible to irrelevant attributes, and its performance can be improved by using a decision tree builder as a filter for attribute selection first. The resulting nearest-neighbor method can also perform better than the decision tree algorithm used for filtering. As another example, the simple 1R scheme described in Chapter 4 has been used to select the attributes for a decision tree learner by evaluating the effect of branching on different attributes (although an error-based method such as 1R may not be the optimal choice for ranking attributes, as we will see later when covering the related problem of supervised discretization). Often the decision tree performs just as well when only the two or three top attributes are used for its construction—and it is much easier to understand. In this approach, the user determines how many attributes to use for building the decision tree.

Another possibility is to use an algorithm that builds a linear model—for example, a linear support vector machine—and ranks the attributes based on the size of the coefficients. A more sophisticated variant applies the learning algorithm repeatedly. It builds a model, ranks the attributes based on the coefficients, removes the highest-ranked one, and repeats the process until all attributes have been removed. This method of *recursive feature elimination* has been found to yield better results on certain datasets (e.g., when identifying important genes for cancer classification) than simply ranking attributes based on a single model. With both methods it is important to ensure that the attributes are measured on the same scale; otherwise, the coefficients are not comparable. Note that these techniques just produce a ranking; another method must be used to determine the appropriate number of attributes to use.

Attributes can be selected using instance-based learning methods, too. You could sample instances randomly from the training set and check neighboring records of the same and different classes—"near hits" and "near misses." If a near hit has a different value for a certain attribute, that attribute appears to be irrelevant and its weight should be decreased. On the other hand, if a near miss has a different value, the attribute appears to be relevant and its weight should be increased. Of course, this is the standard kind of procedure used for attribute weighting for instance-based learning, described in Section 6.4. After repeating this operation many times, selection takes place: only attributes with positive weights are chosen. As in the standard incremental formulation of instance-based learning, different results will be obtained each time the process is repeated, because of the different ordering of examples. This can be avoided by using all training instances and taking into account all near hits and near misses of each.

A more serious disadvantage is that the method will not detect an attribute that is redundant because it is correlated with another attribute. In the extreme case, two identical attributes would be treated in the same way, either both selected or both rejected. A modification has been suggested that appears to go some way towards addressing this issue by taking the current attribute weights into account when computing the nearest hits and misses.

Another way of eliminating redundant attributes as well as irrelevant ones is to select a subset of attributes that individually correlate well with the class but have little intercorrelation. The correlation between two nominal attributes A and B can be measured using the *symmetric uncertainty:*

$$U(A,B) = 2\frac{H(A)+H(B)-H(A,B)}{H(A)+H(B)},$$

where H is the entropy function described in Section 4.3. The entropies are based on the probability associated with each attribute value; $H(A,B)$, the joint entropy of A and B, is calculated from the joint probabilities of all combinations of values of A and B. The symmetric uncertainty always lies between 0 and 1. Correlation-based feature selection determines the goodness of a set of attributes using

$$\sum_j U(A_j,C) \Big/ \sqrt{\sum_i \sum_j U(A_i,A_j)},$$

where C is the class attribute and the indices i and j range over all attributes in the set. If all m attributes in the subset correlate perfectly with the class and with one another, the numerator becomes m and the denominator becomes $\sqrt{m^2}$, which is also m. Hence, the measure is 1, which turns out to be the maximum value it can attain (the minimum is 0). Clearly this is not ideal, because we want to avoid redundant attributes. However, any subset of this set will also have value 1. When using this criterion to search for a good subset of attributes it makes sense to break ties in favor of the smallest subset.

Searching the attribute space

Most methods for attribute selection involve searching the space of attributes for the subset that is most likely to predict the class best. Figure 7.1 illustrates the attribute space for the—by now all-too-familiar—weather dataset. The number of possible attribute subsets increases exponentially with the number of attributes, making exhaustive search impractical on all but the simplest problems.

Typically, the space is searched greedily in one of two directions, top to bottom or bottom to top in the figure. At each stage, a local change is made to the current attribute subset by either adding or deleting a single attribute. The downward direction, where you start with no attributes and add them one at a time, is called *forward selection*. The upward one, where you start with the full set and delete attributes one at a time, is *backward elimination*.

In forward selection, each attribute that is not already in the current subset is tentatively added to it and the resulting set of attributes is evaluated—using, for example, cross-validation as described in the following section. This evaluation produces a numeric measure of the expected performance of the subset. The effect of adding each attribute in turn is quantified by this measure, the best one is chosen, and the procedure continues. However, if no attribute produces an improvement when added to the current subset, the search ends. This is a standard greedy search procedure and guarantees to find a locally—but not necessarily globally—optimal set of attributes. Backward elimination operates in an entirely analogous fashion. In both cases a slight bias is often introduced

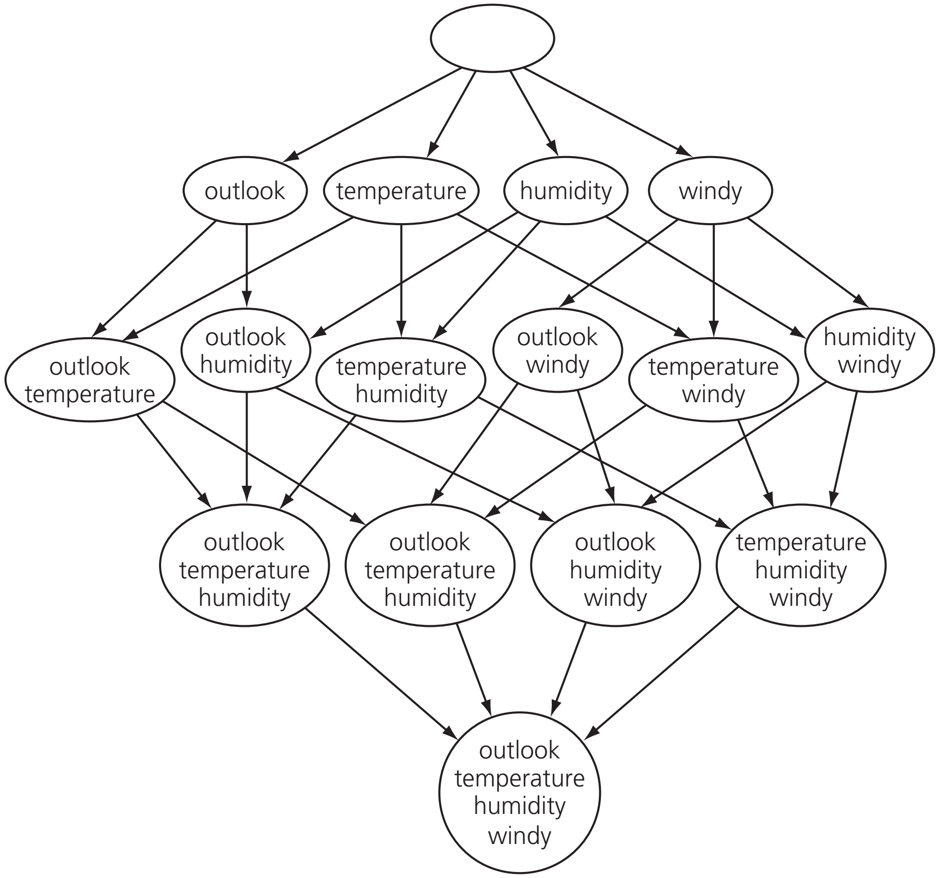


Figure 7.1 Attribute space for the weather dataset.

toward smaller attribute sets. This can be done for forward selection by insisting that if the search is to continue, the evaluation measure must not only increase but also must increase by at least a small predetermined quantity. A similar modification works for backward elimination.

More sophisticated search methods exist. Forward selection and backward elimination can be combined into a bidirectional search; again one can either begin with all the attributes or with none of them. Best-first search is a method that does not just terminate when the performance starts to drop but keeps a list of all attribute subsets evaluated so far, sorted in order of the performance measure, so that it can revisit an earlier configuration instead. Given enough time it will explore the entire space, unless this is prevented by some kind of stopping criterion. Beam search is similar but truncates its list of attribute subsets at each stage so that it only contains a fixed number—the beam width—

of most promising candidates. Genetic algorithm search procedures are loosely based on the principal of natural selection: they "evolve" good feature subsets by using random perturbations of a current list of candidate subsets.

Scheme-specific selection

The performance of an attribute subset with scheme-specific selection is measured in terms of the learning scheme's classification performance using just those attributes. Given a subset of attributes, accuracy is estimated using the normal procedure of cross-validation described in Section 5.3. Of course, other evaluation methods such as performance on a holdout set (Section 5.3) or the bootstrap estimator (Section 5.4) could equally well be used.

The entire attribute selection process is computation intensive. If each evaluation involves a 10-fold cross-validation, the learning procedure must be executed 10 times. With k attributes, the heuristic forward selection or backward elimination multiplies evaluation time by a factor of up to k^2—and for more sophisticated searches, the penalty will be far greater, up to 2^k for an exhaustive algorithm that examines each of the 2^k possible subsets.

Good results have been demonstrated on many datasets. In general terms, backward elimination produces larger attribute sets, and better classification accuracy, than forward selection. The reason is that the performance measure is only an estimate, and a single optimistic estimate will cause both of these search procedures to halt prematurely—backward elimination with too many attributes and forward selection with not enough. But forward selection is useful if the focus is on understanding the decision structures involved, because it often reduces the number of attributes with only a very small effect on classification accuracy. Experience seems to show that more sophisticated search techniques are not generally justified—although they can produce much better results in certain cases.

One way to accelerate the search process is to stop evaluating a subset of attributes as soon as it becomes apparent that it is unlikely to lead to higher accuracy than another candidate subset. This is a job for a paired statistical significance test, performed between the classifier based on this subset and all the other candidate classifiers based on other subsets. The performance difference between two classifiers on a particular test instance can be taken to be -1, 0, or 1 depending on whether the first classifier is worse, the same as, or better than the second on that instance. A paired t-test (described in Section 5.5) can be applied to these figures over the entire test set, effectively treating the results for each instance as an independent estimate of the difference in performance. Then the cross-validation for a classifier can be prematurely terminated as soon as it turns out to be significantly worse than another—which, of course, may never happen. We might want to discard classifiers more aggressively by modifying the *t*-test to compute the probability that one classifier is better than another classifier by at least a small user-specified threshold. If this probability becomes very small, we can discard the former classifier on the basis that it is very unlikely to perform substantially better than the latter.

This methodology is called *race search* and can be implemented with different underlying search strategies. When used with forward selection, we race all possible single-attribute additions simultaneously and drop those that do not perform well enough. In backward elimination, we race all single-attribute deletions. *Schemata search* is a more complicated method specifically designed for racing; it runs an iterative series of races that each determine whether or not a particular attribute should be included. The other attributes for this race are included or excluded randomly at each point in the evaluation. As soon as one race has a clear winner, the next iteration of races begins, using the winner as the starting point. Another search strategy is to rank the attributes first, using, for example, their information gain (assuming they are discrete), and then race the ranking. In this case the race includes no attributes, the top-ranked attribute, the top two attributes, the top three, and so on.

Whatever way you do it, scheme-specific attribute selection by no means yields a uniform improvement in performance. Because of the complexity of the process, which is greatly increased by the feedback effect of including a target machine learning algorithm in the attribution selection loop, it is quite hard to predict the conditions under which it will turn out to be worthwhile. As in many machine learning situations, trial and error using your own particular source of data is the final arbiter.

There is one type of classifier for which scheme-specific attribute selection is an essential part of the learning process: the decision table. As mentioned in Section 3.1, the entire problem of learning decision tables consists of selecting the right attributes to include. Usually this is done by measuring the table's cross-validation performance for different subsets of attributes and choosing the best-performing subset. Fortunately, leave-one-out cross-validation is very cheap for this kind of classifier. Obtaining the cross-validation error from a decision table derived from the training data is just a matter of manipulating the class counts associated with each of the table's entries, because the table's structure doesn't change when instances are added or deleted. The attribute space is generally searched by best-first search because this strategy is less likely to become stuck in a local maximum than others, such as forward selection.

Let's end our discussion with a success story. One learning method for which a simple scheme-specific attribute selection approach has shown good results is Naïve Bayes. Although this method deals well with random attributes, it has the potential to be misled when there are dependencies among attributes, and particularly when redundant ones are added. However, good results have been reported using the forward selection algorithm—which is better able to detect when a redundant attribute is about to be added than the backward elimination approach—in conjunction with a very simple, almost "naïve," metric that determines the quality of an attribute subset to be simply the performance of the learned algorithm on the *training* set. As was emphasized in Chapter 5, training set performance is certainly not a reliable indicator of test-set performance. Nevertheless, experiments show that this simple modification to Naïve Bayes markedly improves its performance on those standard datasets for which it does not do so well as tree- or rule-based classifiers, and does not have any negative effect on results on datasets on which Naïve Bayes already does well. *Selective Naïve Bayes,* as this learning method is called, is a viable machine learning technique that performs reliably and well in practice.

7.2 Discretizing numeric attributes

Some classification and clustering algorithms deal with nominal attributes only and cannot handle ones measured on a numeric scale. To use them on general datasets, numeric attributes must first be "discretized" into a small number of distinct ranges. Even learning algorithms that do handle numeric attributes sometimes process them in ways that are not altogether satisfactory. Statistical clustering methods often assume that numeric attributes have a normal distribution—often not a very plausible assumption in practice—and the standard extension of the Naïve Bayes classifier to handle numeric attributes adopts the same assumption. Although most decision tree and decision rule learners can handle numeric attributes, some implementations work much more slowly when numeric attributes are present because they repeatedly sort the attribute values. For all these reasons the question arises: what is a good way to discretize numeric attributes into ranges before any learning takes place?

We have already encountered some methods for discretizing numeric attributes. The 1R learning scheme described in Chapter 4 uses a simple but effective technique: sort the instances by the attribute's value and assign the value into ranges at the points that the class value changes—except that a certain minimum number of instances in the majority class (six) must lie in each of the ranges, which means that any given range may include a mixture of class values. This is a "global" method of discretization that is applied to all continuous attributes before learning starts.

Decision tree learners, on the other hand, deal with numeric attributes on a local basis, examining attributes at each node of the tree when it is being constructed to see whether they are worth branching on—and only at that point deciding on the best place to split continuous attributes. Although the tree-building method we examined in Chapter 6 only considers binary splits of continuous attributes, one can imagine a full discretization taking place at that point, yielding a multiway split on a numeric attribute. The pros and cons of the local versus the global approach are clear. Local discretization is tailored to the actual context provided by each tree node, and will produce different discretizations of the same attribute at different places in the tree if that seems appropriate. However, its decisions are based on less data as tree depth increases, which compromises their reliability. If trees are developed all the way out to single-instance leaves before being pruned back, as with the normal technique of backward pruning, it is clear that many discretization decisions will be based on data that is grossly inadequate.

When using global discretization before applying a learning method, there are two possible ways of presenting the discretized data to the learner. The most obvious is to treat discretized attributes like nominal ones: each discretization interval is represented by one value of the nominal attribute. However, because a discretized attribute is derived from a numeric one, its values are ordered, and treating it as nominal discards this potentially valuable ordering information. Of course, if a learning scheme can handle ordered attributes directly, the solution is obvious: each discretized attribute is declared to be of type "ordered."

If the learning method cannot handle ordered attributes, there is still a simple way of enabling it to exploit the ordering information: transform each discretized attribute into a set of binary attributes before the learning scheme is applied. Assuming the discretized attribute has k values, it is transformed into $k - 1$ binary attributes, the first $i - 1$ of which are set to *false* whenever the ith value of the discretized attribute is present in the data and to *true* otherwise. The remaining attributes are set to *false.* In other words, the $(i - 1)$th binary attribute represents whether the discretized attribute is less than i. If a decision tree learner splits on this attribute, it implicitly uses the ordering information it encodes. Note that this transformation is independent of the particular discretization method being applied: it is simply a way of coding an ordered attribute using a set of binary attributes.

Unsupervised discretization

There are two basic approaches to the problem of discretization. One is to quantize each attribute in the absence of any knowledge of the classes of the instances in the training set—so-called *unsupervised* discretization. The other is to take the classes into account when discretizing—*supervised* discretization. The former is the only possibility when dealing with clustering problems in which the classes are unknown or nonexistent.

The obvious way of discretizing a numeric attribute is to divide its range into a predetermined number of equal intervals: a fixed, data-independent yardstick. This is frequently done at the time when data is collected. But, like any unsupervised discretization method, it runs the risk of destroying distinctions that would have turned out to be useful in the learning process by using gradations that are too coarse or by unfortunate choices of boundary that needlessly lump together many instances of different classes.

Equal-interval binning often distributes instances very unevenly: some bins contain many instances, and others contain none. This can seriously impair the ability of the attribute to help to build good decision structures. It is often better to allow the intervals to be of different sizes, choosing them so that the same number of training examples fall into each one. This method, *equal-frequency binning,* divides the attribute's range into a predetermined number of bins based on the distribution of examples along that axis—sometimes called *histogram equalization,* because if you take a histogram of the contents of the resulting bins it will be completely flat. If you view the number of bins as a resource, this method makes best use of it.

However, equal-frequency binning is still oblivious to the instances' classes, and this can cause bad boundaries. For example, if all instances in a bin have one class, and all instances in the next higher bin have another except for the first, which has the original class, surely it makes sense to respect the class divisions and include that first instance in the previous bin, sacrificing the equal-frequency property for the sake of homogeneity. Supervised discretization—taking classes into account during the process—certainly has advantages. Nevertheless, it has been found that equal-frequency binning can yield excellent results, at least in conjunction with the Naïve Bayes learning scheme, when the number of bins is chosen in a data-dependent fashion by setting it to the square root of the number of instances. This method is called *proportional k-interval discretization*.

Entropy-based discretization

Because the criterion used for splitting a numeric attribute during the formation of a decision tree works well in practice, it seems a good idea to extend it to more general discretization by recursively splitting intervals until it is time to stop. In Chapter 6 we saw how to sort the instances by the attribute's value and consider, for each possible splitting point, the information gain of the resulting split. To discretize the attribute, once the first split is determined the splitting process can be repeated in the upper and lower parts of the range, and so on, recursively.

To see this working in practice, we revisit the example on page 189 for discretizing the temperature attribute of the weather data, whose values are

64	65	68	69	70	71	72	75	80	81	83	85
yes	no	yes	yes	yes	no	no yes	yes yes	no	yes	yes	no

(Repeated values have been collapsed together.) The information gain for each of the 11 possible positions for the breakpoint is calculated in the usual way. For example, the information value of the test *temperature* < 71.5, which splits the range into four *yes*'s and two *no*'s versus five *yes*'s and three *no*'s, is

$$\text{info}([4,2],[5,3]) = (6/14) \times \text{info}([4,2]) + (8/14) \times \text{info}([5,3]) = 0.939 \text{ bits}$$

This represents the amount of information required to specify the individual values of *yes* and *no* given the split. We seek a discretization that makes the subintervals as pure as possible; hence, we choose to split at the point where the information value is smallest. (This is the same as splitting where the information *gain,* defined as the difference between the information value without the split and that with the split, is largest.) As before, we place numeric thresholds halfway between the values that delimit the boundaries of a concept.

The graph labeled A in Figure 7.2 shows the information values at each possible cut point at this first stage. The cleanest division—smallest information value—is at a temperature of 84 (0.827 bits), which separates off just the very final value, a *no* instance, from the preceding list. The instance classes are written below the horizontal axis to make interpretation easier. Invoking the algorithm again on the lower range of temperatures, from 64 to 83, yields the graph labeled B. This has a minimum at 80.5 (0.800 bits), which splits off the next two values,

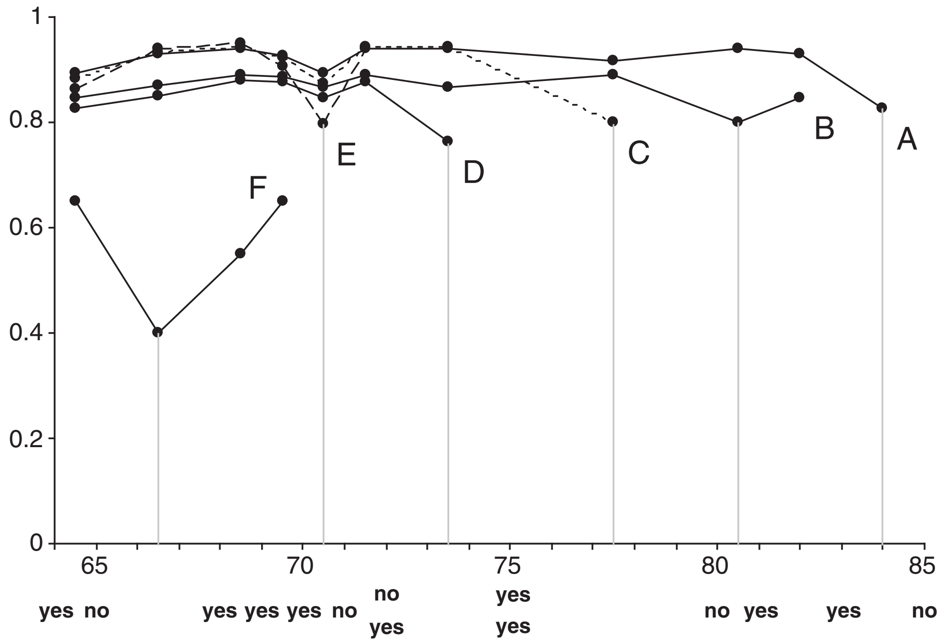


Figure 7.2 Discretizing the *temperature* attribute using the entropy method.

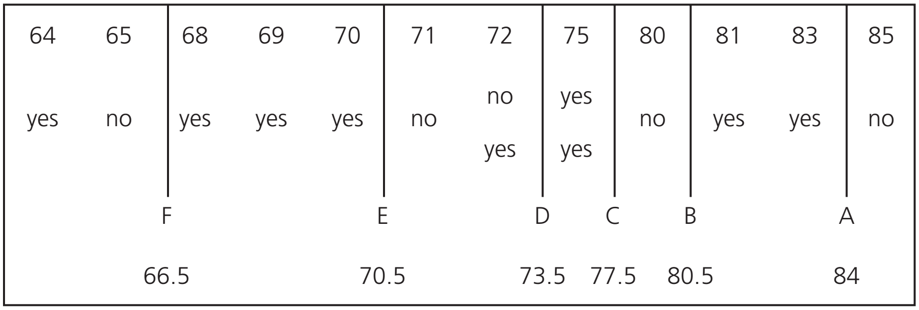


Figure 7.3 The result of discretizing the *temperature* attribute.

both *yes* instances. Again invoking the algorithm on the lower range, now from 64 to 80, produces the graph labeled C (shown dotted to help distinguish it from the others). The minimum is at 77.5 (0.801 bits), splitting off another *no* instance. Graph D has a minimum at 73.5 (0.764 bits), splitting off two *yes* instances. Graph E (again dashed, purely to make it more easily visible), for the temperature range 64 to 72, has a minimum at 70.5 (0.796 bits), which splits off two *no*s and a *yes*. Finally, graph F, for the range 64 to 70, has a minimum at 66.5 (0.4 bits).

The final discretization of the *temperature* attribute is shown in Figure 7.3. The fact that recursion only ever occurs in the first interval of each split is an artifact of this example: in general, both the upper and the lower intervals will have to be split further. Underneath each division is the label of the graph in Figure 7.2 that is responsible for it, and below that is the actual value of the split point.

It can be shown theoretically that a cut point that minimizes the information value will never occur between two instances of the same class. This leads to a useful optimization: it is only necessary to consider potential divisions that separate instances of different classes. Notice that if class labels were assigned to the intervals based on the majority class in the interval, there would be no guarantee that adjacent intervals would receive different labels. You might be tempted to consider merging intervals with the same majority class (e.g., the first two intervals of Figure 7.3), but as we will see later (pages 302–304) this is not a good thing to do in general.

The only problem left to consider is the stopping criterion. In the temperature example most of the intervals that were identified were "pure" in that all their instances had the same class, and there is clearly no point in trying to split such an interval. (Exceptions were the final interval, which we tacitly decided not to split, and the interval from 70.5 to 73.5.) In general, however, things are not so straightforward.

A good way to stop the entropy-based splitting discretization procedure turns out to be the MDL principle that we encountered in Chapter 5. In accordance with that principle, we want to minimize the size of the "theory" plus the size of the information necessary to specify all the data given that theory. In this case, if we do split, the "theory" is the splitting point, and we are comparing the situation in which we split with that in which we do not. In both cases we assume that the instances are known but their class labels are not. If we do not split, the classes can be transmitted by encoding each instance's label. If we do, we first encode the split point (in $\log_2[N-1]$ bits, where N is the number of instances), then the classes of the instances below that point, and then the classes of those above it. You can imagine that if the split is a good one—say, all the classes below it are *yes* and all those above are *no*—then there is much to be gained by splitting. If there is an equal number of *yes* and *no* instances, each instance costs 1 bit without splitting but hardly more than 0 bits with splitting—it is not quite 0 because the class values associated with the split itself must be encoded, but this penalty is amortized across all the instances. In this case, if there are many examples, the penalty of having to encode the split point will be far outweighed by the information saved by splitting.

We emphasized in Section 5.9 that when applying the MDL principle, the devil is in the details. In the relatively straightforward case of discretization, the situation is tractable although not simple. The amounts of information can be obtained exactly under certain reasonable assumptions. We will not go into the details, but the upshot is that the split dictated by a particular cut point is worthwhile if the information gain for that split exceeds a certain value that depends on the number of instances N, the number of classes k, the entropy of the instances E, the entropy of the instances in each subinterval E_1 and E_2, and the number of classes represented in each subinterval k_1 and k_2:

$$gain > \frac{\log_2(N-1)}{N} + \frac{\log_2(3^k-2) - kE + k_1E_1 + k_2E_2}{N}.$$

The first component is the information needed to specify the splitting point; the second is a correction due to the need to transmit which classes correspond to the upper and lower subintervals.

When applied to the temperature example, this criterion prevents any splitting at all. The first split removes just the final example, and as you can imagine very little actual information is gained by this when transmitting the classes—in fact, the MDL criterion will never create an interval containing just one example. Failure to discretize *temperature* effectively disbars it from playing any role in the final decision structure because the same discretized value will be given to all instances. In this situation, this is perfectly appropriate: the *temper-*

ature attribute does not occur in good decision trees or rules for the weather data. In effect, failure to discretize is tantamount to attribute selection.

Other discretization methods

The entropy-based method with the MDL stopping criterion is one of the best general techniques for supervised discretization. However, many other methods have been investigated. For example, instead of proceeding top-down by recursively splitting intervals until some stopping criterion is satisfied, you could work bottom-up, first placing each instance into its own interval and then considering whether to merge adjacent intervals. You could apply a statistical criterion to see which would be the best two intervals to merge, and merge them if the statistic exceeds a certain preset confidence level, repeating the operation until no potential merge passes the test. The χ^2 test is a suitable one and has been used for this purpose. Instead of specifying a preset significance threshold, more complex techniques are available to determine an appropriate level automatically.

A rather different approach is to count the number of errors that a discretization makes when predicting each training instance's class, assuming that each interval receives the majority class. For example, the 1R method described earlier is error based—it focuses on errors rather than the entropy. However, the best possible discretization in terms of error count is obtained by using the largest possible number of intervals, and this degenerate case should be avoided by restricting the number of intervals in advance. For example, you might ask, what is the best way to discretize an attribute into k intervals in a way that minimizes the number of errors?

The brute-force method of finding the best way of partitioning an attribute into k intervals in a way that minimizes the error count is exponential in k and hence infeasible. However, there are much more efficient schemes that are based on the idea of dynamic programming. Dynamic programming applies not just to the error count measure but also to any given additive impurity function, and it can find the partitioning of N instances into k intervals in a way that minimizes the impurity in time proportional to kN^2. This gives a way of finding the best entropy-based discretization, yielding a potential improvement in the quality of the discretization (but in practice a negligible one) over the recursive entropy-based method described previously. The news for error-based discretization is even better, because there is a method that minimizes the error count in time linear in N.

Entropy-based versus error-based discretization

Why not use error-based discretization, since the optimal discretization can be found very quickly? The answer is that there is a serious drawback to error-based

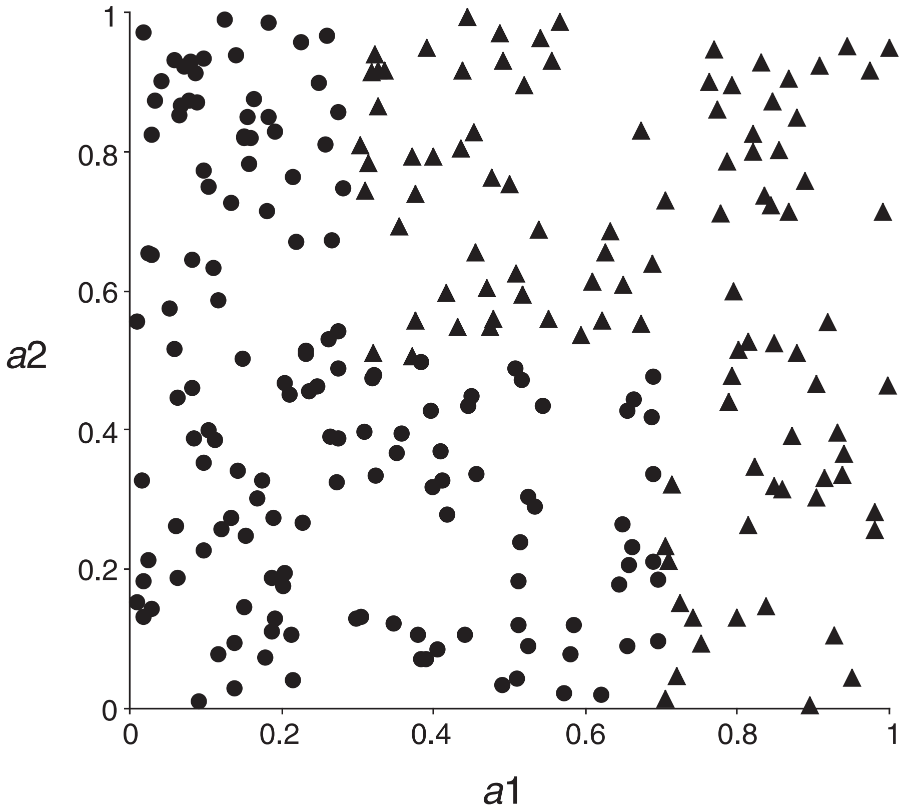


Figure 7.4 Class distribution for a two-class, two-attribute problem.

discretization: it cannot produce adjacent intervals with the same label (such as the first two of Figure 7.3). The reason is that merging two such intervals will not affect the error count but it will free up an interval that can be used elsewhere to reduce the error count.

Why would anyone want to generate adjacent intervals with the same label? The reason is best illustrated with an example. Figure 7.4 shows the instance space for a simple two-class problem with two numeric attributes ranging from 0 to 1. Instances belong to one class (the dots) if their first attribute (*a*1) is less than 0.3 or if it is less than 0.7 *and* their second attribute (*a*2) is less than 0.5. Otherwise, they belong to the other class (triangles). The data in Figure 7.4 has been artificially generated according to this rule.

Now suppose we are trying to discretize both attributes with a view to learning the classes from the discretized attributes. The very best discretization splits *a*1 into three intervals (0 through 0.3, 0.3 through 0.7, and 0.7 through 1.0) and *a*2 into two intervals (0 through 0.5 and 0.5 through 1.0). Given these nominal attributes, it will be easy to learn how to tell the classes apart with a simple decision tree or rule algorithm. Discretizing *a*2 is no problem. For *a*1, however, the first and last intervals will have opposite labels (*dot* and *triangle,* respectively). The second will have whichever label happens to occur most in the region from 0.3 through 0.7 (it is in fact *dot* for the data in Figure 7.4). Either way, this label must inevitably be the same as one of the adjacent labels—of course this is true whatever the class probability happens to be in the middle region. Thus this discretization will not be achieved by any method that minimizes the error counts, because such a method cannot produce adjacent intervals with the same label.

The point is that what changes as the value of *a*1 crosses the boundary at 0.3 is not the majority class but the class *distribution*. The majority class remains *dot.* The distribution, however, changes markedly, from 100% before the boundary to just over 50% after it. And the distribution changes again as the boundary at 0.7 is crossed, from 50% to 0%. Entropy-based discretization methods are sensitive to changes in the distribution even though the majority class does not change. Error-based methods are not.

Converting discrete to numeric attributes

There is a converse problem to discretization. Some learning algorithms—notably the nearest-neighbor instance-based method and numeric prediction techniques involving regression—naturally handle only attributes that are numeric. How can they be extended to nominal attributes?

In instance-based learning, as described in Section 4.7, discrete attributes can be treated as numeric by defining the "distance" between two nominal values that are the same as 0 and between two values that are different as 1—regardless of the actual values involved. Rather than modifying the distance function, this can be achieved using an attribute transformation: replace a *k*-valued nominal attribute with *k* synthetic binary attributes, one for each value indicating whether the attribute has that value or not. If the attributes have equal weight, this achieves the same effect on the distance function. The distance is insensitive to the attribute values because only "same" or "different" information is encoded, not the shades of difference that may be associated with the various possible values of the attribute. More subtle distinctions can be made if the attributes have weights reflecting their relative importance.

If the values of the attribute can be ordered, more possibilities arise. For a numeric prediction problem, the average class value corresponding to each value of a nominal attribute can be calculated from the training instances and used to determine an ordering—this technique was introduced for model trees in Section 6.5. (It is hard to come up with an analogous way of ordering attribute values for a classification problem.) An ordered nominal attribute can be replaced with an integer in the obvious way—but this implies not just an ordering but also a metric on the attribute's values. The implication of a metric can be avoided by creating $k - 1$ synthetic binary attributes for a k-valued nominal attribute, in the manner described on page 297. This encoding still implies an ordering among different values of the attribute—adjacent values differ in just one of the synthetic attributes, whereas distant ones differ in several—but it does not imply an equal distance between the attribute values.

7.3 Some useful transformations

Resourceful data miners have a toolbox full of techniques, such as discretization, for transforming data. As we emphasized in Section 2.4, data mining is hardly ever a matter of simply taking a dataset and applying a learning algorithm to it. Every problem is different. You need to think about the data and what it means, and examine it from diverse points of view—creatively!—to arrive at a suitable perspective. Transforming it in different ways can help you get started.

You don't have to make your own toolbox by implementing the techniques yourself. Comprehensive environments for data mining, such as the one described in Part II of this book, contain a wide range of suitable tools for you to use. You do not necessarily need a detailed understanding of how they are implemented. What you do need is to understand what the tools do and how they can be applied. In Part II we list, and briefly describe, all the transformations in the Weka data mining workbench.

Data often calls for general mathematical transformations of a set of attributes. It might be useful to define new attributes by applying specified mathematical functions to existing ones. Two *date* attributes might be subtracted to give a third attribute representing *age*—an example of a semantic transformation driven by the meaning of the original attributes. Other transformations might be suggested by known properties of the learning algorithm. If a linear relationship involving two attributes, A and B, is suspected, and the algorithm is only capable of axis-parallel splits (as most decision tree and rule learners are), the ratio A/B might be defined as a new attribute. The transformations are not necessarily mathematical ones but may involve world knowledge such as days of the week, civic holidays, or chemical atomic numbers. They could be expressed as operations in a spreadsheet or as functions that are implemented by arbitrary computer programs. Or you can reduce several nominal attributes to one by concatenating their values, producing a single $k_1 \times k_2$-valued attribute from attributes with k_1 and k_2 values, respectively. Discretization converts a numeric attribute to nominal, and we saw earlier how to convert in the other direction too.

As another kind of transformation, you might apply a clustering procedure to the dataset and then define a new attribute whose value for any given instance is the cluster that contains it using an arbitrary labeling for clusters. Alternatively, with probabilistic clustering, you could augment each instance with its membership probabilities for each cluster, including as many new attributes as there are clusters.

Sometimes it is useful to add noise to data, perhaps to test the robustness of a learning algorithm. To take a nominal attribute and change a given percentage of its values. To obfuscate data by renaming the relation, attribute names, and nominal and string attribute values—because it is often necessary to anonymize sensitive datasets. To randomize the order of instances or produce a random sample of the dataset by resampling it. To reduce a dataset by removing a given percentage of instances, or all instances that have certain values for nominal attributes, or numeric values above or below a certain threshold. Or to remove outliers by applying a classification method to the dataset and deleting misclassified instances.

Different types of input call for their own transformations. If you can input sparse data files (see Section 2.4), you may need to be able to convert datasets to a nonsparse form, and vice versa. Textual input and time series input call for their own specialized conversions, described in the subsections that follow. But first we look at two general techniques for transforming data with numeric attributes into a lower-dimensional form that may be more useful for data mining.

Principal components analysis

In a dataset with *k* numeric attributes, you can visualize the data as a cloud of points in *k*-dimensional space—the stars in the sky, a swarm of flies frozen in time, a two-dimensional scatter plot on paper. The attributes represent the coordinates of the space. But the axes you use, the coordinate system itself, is arbitrary. You can place horizontal and vertical axes on the paper and represent the points of the scatter plot using those coordinates, or you could draw an arbitrary straight line to represent the X-axis and one perpendicular to it to represent Y. To record the positions of the flies you could use a conventional coordinate system with a north–south axis, an east–west axis, and an up–down axis. But other coordinate systems would do equally well. Creatures such as flies don't know about north, south, east, and west—although, being subject to gravity, they may perceive up–down as being something special. As for the stars in the sky, who's to say what the "right" coordinate system is? Over the centuries our ancestors moved from a geocentric perspective to a heliocentric one to a purely relativistic one, each shift of perspective being accompanied by turbulent religious–scientific upheavals and painful reexamination of humankind's role in God's universe.

Back to the dataset. Just as in these examples, there is nothing to stop you transforming all the data points into a different coordinate system. But unlike these examples, in data mining there often *is* a preferred coordinate system, defined not by some external convention but by the very data itself. Whatever coordinates you use, the cloud of points has a certain variance in each direction, indicating the degree of spread around the mean value in that direction. It is a curious fact that if you add up the variances along each axis and then transform the points into a different coordinate system and do the same there, you get the same total variance in both cases. This is always true provided that the coordinate systems are *orthogonal,* that is, each axis is at right angles to the others.

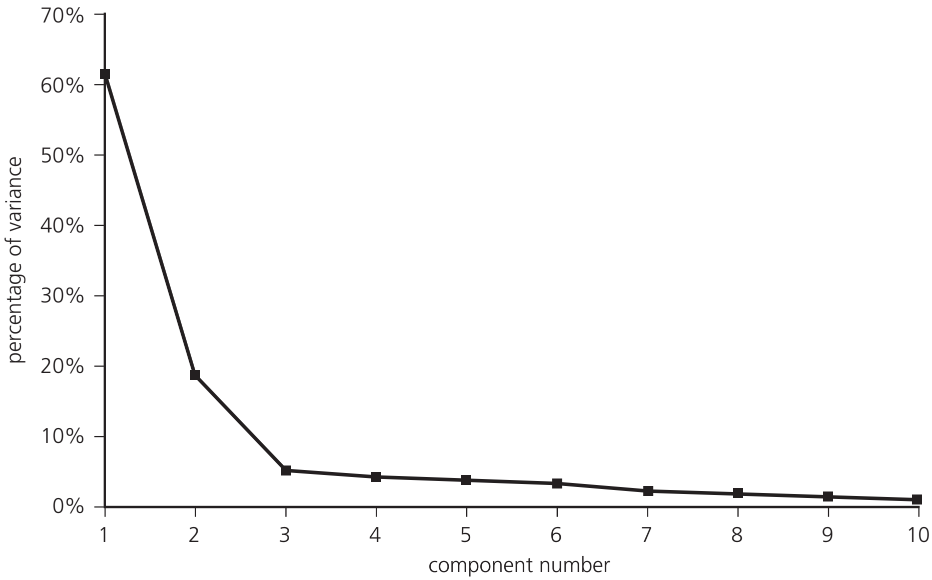
The idea of principal components analysis is to use a special coordinate system that depends on the cloud of points as follows: place the first axis in the direction of greatest variance of the points to maximize the variance along that axis. The second axis is perpendicular to it. In two dimensions there is no choice—its direction is determined by the first axis—but in three dimensions it can lie anywhere in the plane perpendicular to the first axis, and in higher dimensions there is even more choice, although it is always constrained to be perpendicular to the first axis. Subject to this constraint, choose the second axis in the way that maximizes the variance along it. Continue, choosing each axis to maximize its share of the remaining variance.

How do you do this? It's not hard, given an appropriate computer program, and it's not hard to understand, given the appropriate mathematical tools. Technically—for those who understand the italicized terms—you calculate the *covariance matrix* of the original coordinates of the points and *diagonalize* it to find the *eigenvectors*. These are the axes of the transformed space, sorted in order of *eigenvalue*—because each eigenvalue gives the variance along its axis.

Figure 7.5 shows the result of transforming a particular dataset with 10 numeric attributes, corresponding to points in 10-dimensional space. Imagine the original dataset as a cloud of points in 10 dimensions—we can't draw it! Choose the first axis along the direction of greatest variance, the second perpendicular to it along the direction of next greatest variance, and so on. The table gives the variance along each new coordinate axis in the order in which the axes were chosen. Because the sum of the variances is constant regardless of the coordinate system, they are expressed as percentages of that total. We call axes *components* and say that each one "accounts for" its share of the variance. Figure 7.5(b) plots the variance that each component accounts for against the component's number. You can use all the components as new attributes for data mining, or you might want to choose just the first few, the *principal components,*

Axis	Variance	Cumulative
1	61.2%	61.2%
2	18.0%	79.2%
3	4.7%	83.9%
4	4.0%	87.9%
5	3.2%	91.1%
6	2.9%	94.0%
7	2.0%	96.0%
8	1.7%	97.7%
9	1.4%	99.1%
10	0.9%	100%

(a)



(b)

Figure 7.5 Principal components transform of a dataset: (a) variance of each component and (b) variance plot.

and discard the rest. In this case, three principal components account for 84% of the variance in the dataset; seven account for more than 95%.

On numeric datasets it is common to use principal components analysis before data mining as a form of data cleanup and attribute generation. For example, you might want to replace the numeric attributes with the principal component axes or with a subset of them that accounts for a given proportion—say, 95%—of the variance. Note that the scale of the attributes affects the outcome of principal components analysis, and it is common practice to standardize all attributes to zero mean and unit variance first.

Another possibility is to apply principal components analysis recursively in a decision tree learner. At each stage an ordinary decision tree learner chooses to split in a direction that is parallel to one of the axes. However, suppose a principal components transform is performed first, and the learner chooses an axis in the transformed space. This equates to a split along an oblique line in the original space. If the transform is performed afresh before each split, the result will be a multivariate decision tree whose splits are in directions that are not parallel with the axes or with one another.

Random projections

Principal components analysis transforms the data linearly into a lower-dimensional space. But it's expensive. The time taken to find the transformation (which is a matrix comprising the eigenvectors of the covariance matrix) is cubic in the number of dimensions. This makes it infeasible for datasets with a large number of attributes. A far simpler alternative is to use a random projection of the data into a subspace with a predetermined number of dimensions. It's very easy to find a random projection matrix. But will it be any good?

In fact, theory shows that random projections preserve distance relationships quite well on average. This means that they could be used in conjunction with k*D*-trees or ball trees to do approximate nearest-neighbor search in spaces with a huge number of dimensions. First transform the data to reduce the number of attributes; then build a tree for the transformed space. In the case of nearest-neighbor classification you could make the result more stable, and less dependent on the choice of random projection, by building an ensemble classifier that uses multiple random matrices.

Not surprisingly, random projections perform worse than ones carefully chosen by principal components analysis when used to preprocess data for a range of standard classifiers. However, experimental results have shown that the difference is not too great—and that it tends to decrease as the number of dimensions increase. And of course, random projections are far cheaper computationally.

Text to attribute vectors

In Section 2.4 we introduced string attributes that contain pieces of text and remarked that the value of a string attribute is often an entire document. String attributes are basically nominal, with an unspecified number of values. If they are treated simply as nominal attributes, models can be built that depend on whether the values of two string attributes are equal or not. But that does not capture any internal structure of the string or bring out any interesting aspects of the text it represents.

You could imagine decomposing the text in a string attribute into paragraphs, sentences, or phrases. Generally, however, the word is the most useful unit. The text in a string attribute is usually a sequence of words, and is often best represented in terms of the words it contains. For example, you might transform the string attribute into a set of numeric attributes, one for each word, that represent how often the word appears. The set of words—that is, the set of new attributes—is determined from the dataset and is typically quite large. If there are several string attributes whose properties should be treated separately, the new attribute names must be distinguished, perhaps by a user-determined prefix.

Conversion into words—*tokenization*—is not such a simple operation as it sounds. Tokens may be formed from contiguous alphabetic sequences with nonalphabetic characters discarded. If numbers are present, numeric sequences may be retained too. Numbers may involve + or – signs, may contain decimal points, and may have exponential notation—in other words, they must be parsed according to a defined number syntax. An alphanumeric sequence may be regarded as a single token. Perhaps the space character is the token delimiter; perhaps white space (including the tab and new-line characters) is the delimiter, and perhaps punctuation is, too. Periods can be difficult: sometimes they should be considered part of the word (e.g., with initials, titles, abbreviations, and numbers), but sometimes they should not (e.g., if they are sentence delimiters). Hyphens and apostrophes are similarly problematic.

All words may be converted to lowercase before being added to the dictionary. Words on a fixed, predetermined list of function words or *stopwords*—such as *the, and,* and *but*—could be ignored. Note that stopword lists are language dependent. In fact, so are capitalization conventions (German capitalizes all nouns), number syntax (Europeans use the comma for a decimal point), punctuation conventions (Spanish has an initial question mark), and, of course, character sets. Text is complicated!

Low-frequency words such as *hapax legomena*[3] are often discarded, too. Sometimes it is found beneficial to keep the most frequent *k* words after stopwords have been removed—or perhaps the top *k* words for each class.

Along with all these tokenization options, there is also the question of what the value of each word attribute should be. The value may be the word count—the number of times the word appears in the string—or it may simply indicate the word's presence or absence. Word frequencies could be normalized to give each document's attribute vector the same Euclidean length. Alternatively,

[3] A *hapax legomena* is a word that only occurs once in a given corpus of text.

the frequencies f_{ij} for word i in document j can be transformed in various standard ways. One standard logarithmic term frequency measure is $\log(1 + f_{ij})$. A measure that is widely used in information retrieval is TF × IDF, or "term frequency times inverse document frequency." Here, the term frequency is modulated by a factor that depends on how commonly the word is used in other documents. The TF × IDF metric is typically defined as

$$f_{ij} \log \frac{\text{number of documents}}{\text{number of documents that include word } i}.$$

The idea is that a document is basically characterized by the words that appear often in it, which accounts for the first factor, except that words used in every document or almost every document are useless as discriminators, which accounts for the second. TF × IDF is used to refer not just to this particular formula but also to a general class of measures of the same type. For example, the frequency factor f_{ij} may be replaced by a logarithmic term such as $\log(1 + f_{ij})$.

Time series

In time series data, each instance represents a different time step and the attributes give values associated with that time—such as in weather forecasting or stock market prediction. You sometimes need to be able to replace an attribute's value in the current instance with the corresponding value in some other instance in the past or the future. It is even more common to replace an attribute's value with the *difference* between the current value and the value in some previous instance. For example, the difference—often called the *Delta*—between the current value and the preceding one is often more informative than the value itself. The first instance, in which the time-shifted value is unknown, may be removed, or replaced with a missing value. The Delta value is essentially the first derivative scaled by some constant that depends on the size of the time step. Successive Delta transformations take higher derivatives.

In some time series, instances do not represent regular samples, but the time of each instance is given by a *timestamp* attribute. The difference between timestamps is the step size for that instance, and if successive differences are taken for other attributes they should be divided by the step size to normalize the derivative. In other cases each attribute may represent a different time, rather than each instance, so that the time series is from one attribute to the next rather than from one instance to the next. Then, if differences are needed, they must be taken between one attribute's value and the next attribute's value for each instance.

7.4 Automatic data cleansing

A problem that plagues practical data mining is poor quality of the data. Errors in large databases are extremely common. Attribute values, and class values too, are frequently unreliable and corrupted. Although one way of addressing this problem is to painstakingly check through the data, data mining techniques themselves can sometimes help to solve the problem.

Improving decision trees

It is a surprising fact that decision trees induced from training data can often be simplified, without loss of accuracy, by discarding misclassified instances from the training set, relearning, and then repeating until there are no misclassified instances. Experiments on standard datasets have shown that this hardly affects the classification accuracy of C4.5, a standard decision tree induction scheme. In some cases it improves slightly; in others it deteriorates slightly. The difference is rarely statistically significant—and even when it is, the advantage can go either way. What the technique does affect is decision tree size. The resulting trees are invariably much smaller than the original ones, even though they perform about the same.

What is the reason for this? When a decision tree induction method prunes away a subtree, it applies a statistical test that decides whether that subtree is "justified" by the data. The decision to prune accepts a small sacrifice in classification accuracy on the training set in the belief that this will improve test-set performance. Some training instances that were classified correctly by the unpruned tree will now be misclassified by the pruned one. In effect, the decision has been taken to ignore these training instances.

But that decision has only been applied locally, in the pruned subtree. Its effect has not been allowed to percolate further up the tree, perhaps resulting in different choices being made of attributes to branch on. Removing the misclassified instances from the training set and relearning the decision tree is just taking the pruning decisions to their logical conclusion. If the pruning strategy is a good one, this should not harm performance. It may even improve it by allowing better attribute choices to be made.

It would no doubt be even better to consult a human expert. Misclassified training instances could be presented for verification, and those that were found to be wrong could be deleted—or better still, corrected.

Notice that we are assuming that the instances are not misclassified in any systematic way. If instances are systematically corrupted in both training and test sets—for example, one class value might be substituted for another—it is only to be expected that training on the erroneous training set would yield better performance on the (also erroneous) test set.

Interestingly enough, it has been shown that when artificial noise is added to attributes (rather than to classes), test-set performance is improved if the same noise is added in the same way to the training set. In other words, when attribute noise is the problem it is not a good idea to train on a "clean" set if performance is to be assessed on a "dirty" one. A learning method can learn to compensate for attribute noise, in some measure, if given a chance. In essence, it can learn which attributes are unreliable and, if they are all unreliable, how best to use them together to yield a more reliable result. To remove noise from attributes for the training set denies the opportunity to learn how best to combat that noise. But with class noise (rather than attribute noise), it is best to train on noise-free instances if possible.

Robust regression

The problems caused by noisy data have been known in linear regression for years. Statisticians often check data for outliers and remove them manually. In the case of linear regression, outliers can be identified visually—although it is never completely clear whether an outlier is an error or just a surprising, but correct, value. Outliers dramatically affect the usual least-squares regression because the squared distance measure accentuates the influence of points far away from the regression line.

Statistical methods that address the problem of outliers are called *robust*. One way of making regression more robust is to use an absolute-value distance measure instead of the usual squared one. This weakens the effect of outliers. Another possibility is to try to identify outliers automatically and remove them from consideration. For example, one could form a regression line and then remove from consideration those 10% of points that lie furthest from the line. A third possibility is to minimize the *median* (rather than the mean) of the squares of the divergences from the regression line. It turns out that this estimator is very robust and actually copes with outliers in the X-direction as well as outliers in the Y-direction—which is the normal direction one thinks of outliers.

A dataset that is often used to illustrate robust regression is the graph of international telephone calls made from Belgium from 1950 to 1973, shown in Figure 7.6. This data is taken from the Belgian Statistical Survey published by the Ministry of Economy. The plot seems to show an upward trend over the years, but there is an anomalous group of points from 1964 to 1969. It turns out that during this period, results were mistakenly recorded in the total number of *minutes* of the calls. The years 1963 and 1970 are also partially affected. This error causes a large fraction of outliers in the Y-direction.

Not surprisingly, the usual least-squares regression line is seriously affected by this anomalous data. However, the least *median* of squares line remains

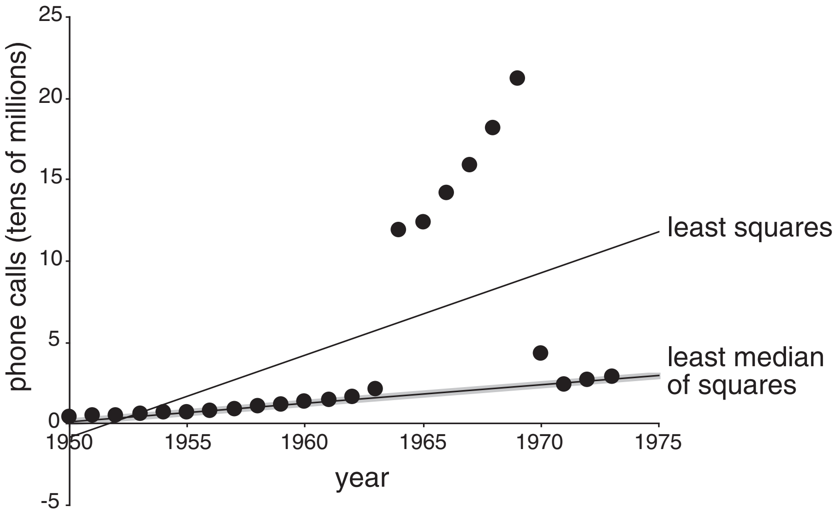


Figure 7.6 Number of international phone calls from Belgium, 1950–1973.

remarkably unperturbed. This line has a simple and natural interpretation. Geometrically, it corresponds to finding the narrowest strip covering half of the observations, where the thickness of the strip is measured in the vertical direction—this strip is marked gray in Figure 7.6; you need to look closely to see it. The least median of squares line lies at the exact center of this band. Note that this notion is often easier to explain and visualize than the normal least-squares definition of regression. Unfortunately, there is a serious disadvantage to median-based regression techniques: they incur a high computational cost, which often makes them infeasible for practical problems.

Detecting anomalies

A serious problem with any form of automatic detection of apparently incorrect data is that the baby may be thrown out with the bathwater. Short of consulting a human expert, there is really no way of telling whether a particular instance really is an error or whether it just does not fit the type of model that is being applied. In statistical regression, visualizations help. It will usually be visually apparent, even to the nonexpert, if the wrong kind of curve is being fitted—a straight line is being fitted to data that lies on a parabola, for example. The outliers in Figure 7.6 certainly stand out to the eye. But most problems cannot be so easily visualized: the notion of "model type" is more subtle than a regression line. And although it is known that good results are obtained on most standard datasets by discarding instances that do not fit a decision tree model, this is not necessarily of great comfort when dealing with a particular new dataset. The suspicion will remain that perhaps the new dataset is simply unsuited to decision tree modeling.

One solution that has been tried is to use several different learning schemes—such as a decision tree, and a nearest-neighbor learner, and a linear discriminant function—to filter the data. A conservative approach is to ask that all three schemes fail to classify an instance correctly before it is deemed erroneous and removed from the data. In some cases, filtering the data in this way and using the filtered data as input to a final learning scheme gives better performance than simply using the three learning schemes and letting them vote on the outcome. Training all three schemes on the *filtered* data and letting them vote can yield even better results. However, there is a danger to voting techniques: some learning algorithms are better suited to certain types of data than others, and the most appropriate method may simply get out-voted! We will examine a more subtle method of combining the output from different classifiers, called *stacking,* in the next section. The lesson, as usual, is to get to know your data and look at it in many different ways.

One possible danger with filtering approaches is that they might conceivably just be sacrificing instances of a particular class (or group of classes) to improve accuracy on the remaining classes. Although there are no general ways to guard against this, it has not been found to be a common problem in practice.

Finally, it is worth noting once again that automatic filtering is a poor substitute for getting the data right in the first place. If this is too time consuming and expensive to be practical, human inspection could be limited to those instances that are identified by the filter as suspect.

7.5 Combining multiple models

When wise people make critical decisions, they usually take into account the opinions of several experts rather than relying on their own judgment or that of a solitary trusted adviser. For example, before choosing an important new policy direction, a benign dictator consults widely: he or she would be ill advised to follow just one expert's opinion blindly. In a democratic setting, discussion of different viewpoints may produce a consensus; if not, a vote may be called for. In either case, different expert opinions are being combined.

In data mining, a model generated by machine learning can be regarded as an expert. *Expert* is probably too strong a word!—depending on the amount and quality of the training data, and whether the learning algorithm is appropriate to the problem at hand, the expert may in truth be regrettably ignorant—but we use the term nevertheless. An obvious approach to making decisions more reliable is to combine the output of different models. Several machine