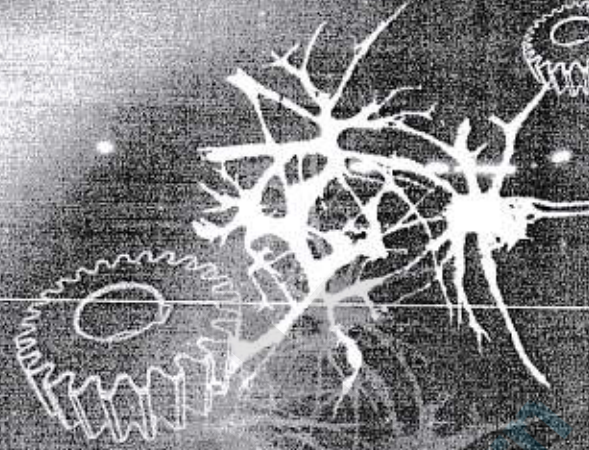


PG 1572

Zbigniew Michalewicz
Martin Schmidt
Matthew Mitchell
Constantin G. Botea



0100 0001
0100 0010
0100 1001

Adaptive Business Intelligence



TCET
005.74
MICSCH
PG001572




Springer

PG 1572

Zbigniew Michalewicz · Martin Schmidt
Matthew Michalewicz · Constantin Chiriac

Adaptive Business Intelligence

005.74
MIC / MIC

 Springer

5 Prediction Methods and Models

"When you have eliminated the impossible, whatever remains, *however improbable*, must be the truth."

The Sign of Four

"As to Holmes, I observed that he sat frequently for half an hour on end, with knitted brows and an abstract air, but he swept the matter away with a wave of his hand when I mentioned it. 'Data! data! data!' he cried impatiently. 'I can't make bricks without clay.'"

The Adventure of the Copper Beeches

Most "prediction problems" can be categorized as classification problems, regression problems, or time series problems. When placing a prediction problem into one of these three categories, two major aspects have to be taken into account: the *expected output* and *time*. Let us explain these two aspects further.

For some problems, there are only two possible expected outputs: "yes" or "no," "true" or "false," "buy" or "sell," etc. These are classic classification problems,¹ because they assign new cases to a class. The best example would be classification of credit card transactions into two classes: "fraudulent" and "legitimate" (this problem is discussed in more detail in Sect. 12.5). A classification problem may have, however, more than two outputs – in fact, the number of possible classes (i.e., expected outputs) might be quite significant (e.g., different types of diseases). In these classification problems, time does not exist; the "future" is understood as an arrival of a new (yet unknown) case, or it is included as a variable of the case.

Similar comments are also applicable to regression problems. The general purpose of (multiple) regression is to discover the relationship between several independent ("predictor") variables and a dependent ("criterion") variable, with the output being a concrete number. For example, we may want to predict salary levels as a function of position, number of years at the position, number of supervised employees, etc. A regression model will also tell us which variables are better predictors than others, and we can easily identify "outliers."² Again, the issue of time is either non-existent or included as a variable of the case.

¹ A prediction model developed for a classification problem is often called a *classifier*.

² An *outlier* is an observation that lies at an abnormal distance from the other values in a random sample. For example, the annual salary level for 1,000 randomly selected people might be in the range of \$17,832 to \$167,942, with the exception of one person, an outlier, who earns \$938,400 per year.

In contrast to classification and regression problems, “time” is the main feature of a time series problem, with each case containing many values measured over some time period in the past. In other words, the time-dependencies among the cases are so strong that the cases must be kept in a sequential time order. In time series problems, the future is referenced explicitly: we would like to predict a variable’s value in the future (tomorrow, next month, etc.). A classic example in economics would be to predict next year’s Gross Domestic Product (GDP). Plenty of historical data are available (released every quarter), and the prediction model may include many additional economic indicators as variables (e. g., employment, financial, survey, production, and sales indicators).

Despite the fact that prediction problems come in all shapes and sizes – varying in the number of variables, types of data patterns, time horizons, and types of expected output – only two types of prediction methods exist for addressing these problems: *quantitative* and *qualitative* methods. The quantitative methods assume that a sufficient amount of data exists about the past, that these data can be quantified in the form of numerical data, and that past patterns will continue into the future. Conversely, qualitative methods are applied in situations where very little quantitative data are available, but where sufficient qualitative knowledge exists.

Although quantitative methods vary from simple (and intuitive) methods based on empirical experience to formal methods based on statistical principles, all these methods require data! Fortunately, the amount of stored data are growing at a rapid rate. This growth takes place on two dimensions: the number of cases stored (e. g., new transactions) and the number of variables in each case (e. g., the detail of each transaction). In general, the more data the better, as data mining can produce better results when performed on large data sets, and the resulting prediction models are more accurate.

In the car distribution example, there are several important elements of prediction. For example, we would like to predict the sale prices for different cars at different auction sites on different days. Because these predictions are based on past cases, we should know all the variables (e. g., “make,” “model,” “body style,” “mileage”) of the cars that were sold over the last, say, three years; and we should also know the sale price, and the exact date and location. Having all this information, we can then apply various prediction methods to develop a good prediction model. Of course, as we discussed in Chap. 3, the prediction model should also take into account the distribution of other cars as well, because of the volume effect.

The process of building a prediction model usually consists of a few steps:

- *Data preparation.* To avoid the situation of “garbage in, garbage out,” the relevant data must be “prepared.” This step includes data transformation, normalization, creation of derived attributes, variable selection, elimination of noisy data, supplying missing values, and data cleaning. This stage is often augmented by preliminary data analysis to identify the most relevant variables and to determine the complexity of the underlying problem. The data preparation step can be the most laborious, and many people believe that it constitutes 80% of any data mining effort.

- **Model building.** This step includes a complete analysis of the data (i. e., the data mining stage), the selection of the best prediction method on the basis of (a) explaining the variability in question, and (b) producing consistent results, and the development of one or more prediction models.
- **Deployment and evaluation.** This step includes implementing the best prediction model, and applying it to new data to generate predictions. However, because new data arrive on a continuous basis, it is essential to measure the prediction model's performance and tune it accordingly.

Let us examine each of these steps.

5.1 Data Preparation

Generally speaking, there are only two "types" of variables: *numerical* and *nominal*. Numerical variables are numbers (e. g., "34,982" for mileage), while nominal variables take their values from a predefined set (e. g., "black," "white," or "red" for color). Because the values of nominal variables are symbols (strings of characters), there is rarely any order between them, and mathematical comparisons and operations do not make much sense (as it is difficult to add "50" to "blue," or to compare which is larger: "blue" or "green"). Hence, it makes sense to talk about *ordered* nominal variables, where comparisons of the type "greater than," "less than," and "equal to" have meaning.

An additional type of variable is binary (also called a Boolean or true/false variable), as it only takes one of two possible values (e. g., "yes" or "no," "true" or "false"). We may also come across other types of variables (e. g., variables that store free text as a value, or that contain a set of values). Most prediction methods and models require that variables be either binary or numerical (or nominal with numerical codes as values), thus allowing some order. So, what should we do with truly nominal variables, such as color? Well, there are two possibilities: Either the color of a car can be coded as a unique number, or it can be converted into several binary (true/false) variables, with each variable representing a particular color. For example, if the color of the car is white, then the variable can take on the value "true" (or "1"); if the color of the car is not white, then the value would be "false" (or "0").

To properly prepare the data, it is important to first identify the variable "type" (i. e., to know whether the values of a variable allow arithmetical operations or logical comparisons, whether there is a natural order imposed among them, and whether it is meaningful to define a distance between the values). For example, "very light," "light," "medium," "heavy," and "very heavy" follow a natural order, but the distance between them is not defined. Mileage values, on the other hand, have a natural measure of distance: a car with 34,789 miles has 3,500 miles less than a car with 38,289 miles. Because the goal of any prediction model is to produce an output (i. e., the prediction), it is important to note that the output is also a variable. In the car distribution example, the predicted output is the sale price, which is a single numerical variable.

In the data preparation phase, some variables may also require “transformation.” For example, it is quite typical for “date of birth” to be recorded as a variable, but many decisions (or queries to the system) may be based on the “age” of an individual. A simple data transformation step would convert the variable “date of birth” into the variable “age” by subtracting a person’s date of birth from the current date. Returning to the car distribution example, the VIN is clearly the key variable, because we can use it to identify any car. However, the VIN itself is not useful for data mining activities (after all, the string of 17 characters looks random and meaningless: e. g., JD8320DJ2094GK2X3), but by using a VIN decoder it can be transformed into meaningful information about the make, model, year, and trim-level of a car.

Although data transformation is an important step in the data preparation process, *variable selection* and *variable composition* are even more important. Variable composition – which is somewhat similar to data transformation – requires problem-specific knowledge to create new variables. Because these new variables (often called *synthetic variables*) present existing data in a “better” form, they may have a greater impact on the results than the specific prediction model used to produce these results. A trivial example is the creation of a new variable to record the average miles driven per year, which corresponds to the ratio:

$$\text{Mileage} / (\text{Current Year} - \text{Year} + 1)$$

The denominator would tell us the number of years the car was in service, and the entire ratio would tell us the average miles driven per year.

Variable selection on the other hand (also known as *feature selection* or *attribute selection*) is the process of selecting the most relevant variables. This process should be performed carefully, because if meaningful variables are not selected then everything else – from data transformation all the way to the final prediction model – will be meaningless. Conversely, selecting irrelevant variables may deteriorate the accuracy of a prediction model (in other words, removing irrelevant variables usually improves the performance of a prediction model). This may seem straightforward: After all, there are a finite number of variable subsets, so we can examine all of them and select the best one! Unfortunately, it is not quite that simple. First, the number of possible subsets may be too large: For a database with “only” 20 variables, there are over 1 million possible subsets. Second, to evaluate each subset, we will need to build a prediction model and evaluate it by measuring the prediction error (we will discuss this in detail in Sect. 5.3, along with some other validation issues). So, what is the solution?

Although the best way to select the most relevant variables is still manual (based on problem-specific knowledge), there are numerous automatic methods that can be divided into several categories. For example, we can consider methods that evaluate the relevance of a single variable versus methods that evaluate a subset of variables. Another category of automatic methods is based on the timing of selection: Some methods select variables at the very beginning using characteristics of the data, while other methods select relevant variables during the model construction process.

Let us consider a few (simple) examples of different automatic methods for variable selection that we could apply to a problem. Say the prediction problem is one of classification (e. g., “fraudulent” and “legitimate”) and we are trying to evaluate the usefulness of particular variables (such as the time of transaction, or the amount) for predicting the outcome. One of the most popular automatic methods we could use is based on “means and variances.” Using a simple statistical test, the means of a variable are compared for the two classes to see whether the difference is likely to be random or not. Small differences in means usually imply irrelevant variables. This method evaluates the variables one by one, and does so before the development of any prediction model. On the other hand, we could use an automatic method where the variable selection process is an inherent part of the prediction model. For example, when a decision tree is built (these are covered in the next section), the relevant variables are selected, one by one, during the tree-building process. Lastly, we could also use automatic methods that evaluate the entire subset of variables. Many optimization techniques discussed in Chap. 6 would be appropriate for this type of approach, as the problem is really an optimization problem (i. e., finding the “optimal” subset of variables).

Because the variable selection step removes redundant and/or non-productive variables, we can consider this step as part of the “data reduction” process, the general goal of which is to delete nonessential data (as the data set may be too big for some prediction models and/or the expected time for building a model might be too long). As data are represented in a table, we can: (1) reduce some variables (columns) in the table, (2) reduce some values present in the table, and/or (3) reduce some cases (rows) from the table. We have already discussed the removal of some variables, which is equivalent to the task of variable selection, so let us move on to reducing values.

It is often necessary to “discretize” a numeric attribute into a smaller number of distinct categories (e. g., the variable “mileage” can be grouped into values of “below 10,000 miles,” “between 10,000 and 19,999 miles,” “between 20,000 and 29,999 miles,” and so on, right up to “over 200,000 miles”). This looks natural, but how can we be sure that such discretization is any good? Moreover, what is a good way to discretize numeric variables into categories? As usual, there are a few possibilities to consider. One approach would be to discretize an attribute by rounding: The actual mileage of the car can be rounded off to the closest 1,000 miles, thus 23,772 miles would become 23,000 miles. Another possibility would be to create some number of discrete categories (say, 20), and distribute all values to these categories in such a way that the average distance of a value from its category mean is the smallest. For example, the first category may contain mileages from 0 to 11,209, the second category may contain mileages from 11,789 to 18,991, and so on. Some mathematical methods (such as *k-means clustering*) can deliver near-optimal solutions for such distributions. However, this approach might be a bit risky for time-changing data. In the case of off-lease cars, new cases are coming in at regular intervals, so the optimal mileage distributions might change quite frequently.

Finally, let us turn our attention to the last possibility of data reduction, which is the reduction of cases from the table. Clearly, the number of cases is often the

largest dimension of the data; it is not unusual to have hundreds of millions of cases containing 20 to 30 variables each. This does not mean, however, that the process of case reduction is easy. Just the opposite is true: very often, case reduction is the hardest type of data reduction to perform. The general approach for handling case reduction is based on random sampling. Rather than using the whole data set to build a prediction model, random samples are used instead. Two popular techniques for random sampling include:

- *Incremental sampling.* Where the model is trained on increasingly larger random subsets of cases, the trends are observed, and the process is stopped when no significant progress is made.
- *Average sampling.* Where several samples of the same size are drawn from the data set, a prediction model is created for each sample, and the outputs of all the models are combined by voting or averaging (more on this in Sect. 10.1).

While discussing data preparation, it is also worthwhile to mention some other aspects of this phase. Some problems require *data normalization* (e. g., scaling some values to a specific range, say, [0, 1]). For example, the age of a car (in number of years) should be interpreted on a different scale than the mileage. In particular, two cars of the same age, but which differ by five miles, can be considered quite similar (assuming that the other variables are the same), whereas two cars of the same mileage, but which differ by five years, are quite different. Data normalization also allows us to express some values as integers, categories, floating point numbers, labels, etc. For instance, we can transform \$400 in damage into "damage level" = 0.04, or we can assign damage to 1 of 10 categories, such as category 1 for damage under \$500, category 2 for damage between \$501 and \$1,000, and so forth. As indicated earlier, we can also transform the variable "color" into 20 binary variables, one for each color: "white," "silver," "red," "green," "blue," ..., "black."

Another important issue connected with data preparation is that of inaccurate or missing values. Inaccurate values usually arise from typographical errors. Some of these errors can be "discovered" by analyzing the outliers for each variable, but some of them may be difficult to find. Furthermore, in almost any data set, some values are not recorded. For example, the color might be unknown for some cars, or the mileage might be missing. Sometimes missing values are treated as just another variable value (e. g., "white," "silver," "red," ..., "black," "unknown"). Another possibility would be to ignore all cases with missing values, but in some data sets we might lose over 90% of the cases by doing this! Yet another way of approaching the problem of missing values is to replace them with the variable's mean value. This might be tempting, but it is very risky, as the data could become biased. Instead, it is safer to observe a relationship between the variable in question and some other variables, and then replace the missing value with an estimated value. For example, we can estimate the mileage on the basis of other variables (such as "year" and "type"): a four-year-old off-lease car should have around 48,000 miles, because car leases typically allow for 12,000 miles per year.

The final aspect of data preparation is connected with time-dependent data. Because all orders, deliveries, and transactions have some sort of a time stamp, most real-world business problems have some time-dependent relationships within their

data sets. Even some relatively “stable” data sets, such as bank customers, change over time. Of course, these types of changes happen at a much slower rate than changes in the stock market, but they do happen. Thus, this additional dimension of time – additional to cases and variables – plays a significant role in most prediction models. This time factor necessitates updates of the prediction model at regular intervals. This can be done online, when new data arrive, or offline, by analyzing the new data and modifying the prediction model. We will return to this issue later, in Sect. 10.3, when we discuss the process of updating a prediction model.

Time dependencies should be recognized and dealt with during the data preparation phase. Usually, time series models assume that the values for some variables are recorded at fixed intervals. For example, we can record the US Gross Domestic Product at the end of each quarter, the Dow Jones Industrial Average at the end of each business day, the temperature at some location every four hours (i. e., six readings a day), and so on. However, if we look at the car distribution example, our time series is far less regular. Although the price prediction is for a particular make/model, there are many subcategories within each make/model category (because of different mileage, color, trim, etc.). Hence, if we find several exact cases from the past, they will not have regular time intervals: For example, a blue Toyota Corolla with 33,000 miles was sold on April 13th, two more were sold in early May, and another was sold in late August – but we have to make prediction for this exact car for mid-October. Because of these interval irregularities, we should relax the precision of some input variables. For instance, color need not be exactly the same (and for some makes/models, color is not a major influencer of price anyway). On top of everything, we are not predicting the value of a variable for the “next” time unit. If we ship a car from California to an auction site in Arizona, we might be interested in a price prediction for next week (as the shipping time would be several days). On the other hand, if we ship the same car to an auction site in New York, then we might be interested in a price prediction for three weeks from today! Needless to say, the volume effect should also be taken into account, as other distribution decisions may influence the actual price of the car!

Another important issue related to time-dependency is the “time horizon” of the historical data. Simply put, we have to make a decision on how far back to look. It seems natural that we should pay more attention to recent data, as “old” data may have lost their significance. For example, using pre-September 11th, 2001 data to predict air traffic for 2002 would not yield good results.

Some people also consider a preliminary (exploratory) analysis to be a part of the data preparation phase, while others consider it a separate stage of the data mining process. In either case, such an analysis is extremely helpful for gaining an understanding of the data. Preliminary data analysis usually includes graphing data for visual inspection (e. g., we can graph the prices for a particular make/model with respect to mileage), and computing some simple statistics such as averages, minimums, maximums, means, standard deviations, and percentiles for each data set (e. g., the prices of a particular make/model at a particular auction site). We can also use “decomposition analysis” to detect trends, seasonality, cycles, and to identify

outliers. Anyway, the main purpose of such an analysis is not to immediately select a prediction model, but rather to get a “feel” for data. This stage is vital, as it can suggest the appropriate prediction method.

5.2 Different Prediction Methods

After the data are prepared, we can begin our search for the right prediction method. The goal is to build a prediction model that will predict the “outcome” of a new case. This outcome might be the price of a used car sent to auction, the classification of a loan application, the assignment of a new customer to the appropriate cluster, and so on. Many prediction methods have been developed over the years that differ from one another in the representation of a solution (e. g., decision tree versus a set of rules), as well as some other differences (e. g., whether they are capable of “explaining” the prediction, the ease with which a solution can be edited). We can group these different prediction methods into a few broad categories:

- Mathematical (e. g., linear regression, statistical methods).
- Distance (e. g., instance-based learning, clustering).
- Logic (e. g., decision tables, decision trees, classification rules).
- Modern heuristic (e. g., neural networks, evolutionary algorithms, fuzzy logic).

The first three categories are covered in this chapter, but the last category, modern heuristic methods, is covered in later chapters. These heuristic methods include fuzzy systems (Chap. 7), neural networks (Chap. 8), genetic programming (Chap. 9), and agent-based systems (Chap. 9). One can argue, of course, that neural networks can be placed in the category of mathematical models, whereas fuzzy systems and genetic programming are in the category of logic models (as they represent classification rules and decision trees, respectively). However, these techniques are of growing importance for building prediction models, and so we have moved them into separate chapters to discuss them in greater depth.

5.2.1 Mathematical Methods

As discussed earlier in this chapter, there are three types of prediction problems: classification, regression, and time series. Classification problems have been the focus of data mining research for the last few decades, and some prediction methods (e. g., distance and logic) were developed explicitly for classification problems. For the time being, however, let us focus on regression and time series problems.

The major difference between regression and time series problems is that the former assumes that the expected output exhibits some explanatory relationship with some other variables. For example, someone’s (predicted) salary might be a function of education, experience, industry, and location. In such cases, an explanatory method would be used to find the relationship between these variables

and make a prediction. The goal of time series models, on the other hand, is not to discover or explain the relationships between variables; their goal is purely one of prediction. Neural networks (Chap. 8) are a good example of this: We may not understand the connection weights, the importance of particular variables or their relationship, and yet the neural network model might be producing quite accurate predictions ...

Probably the most popular explanatory method is *linear regression*. If the predicted outcome is numeric and all the variables in the prediction model are numeric, then linear regression is the classic choice.³ In this method, we build a linear expression that uses the values of different variables to produce a predicted value for a “new” variable (i.e., a variable not used in the model). To illustrate this prediction method in more detail, let us consider linear regression for predicting the auction price of a car. In this case, the “new” variable would be the predicted sale price. Note that many variables are *not* numeric, so we have to address this issue first. It is clear that the non-numeric variables “make,” “model,” and “location” are of key importance, as they determine the basic price range (which is further influenced by the mileage, year, trim, etc.). By building a separate regression model for each make/model at each location, we can eliminate these three non-numeric variables.

Next, we should convert the remaining non-numeric variables into numeric variables. For example, we can take a list of the available colors, sort them from white to black according to some standard order (e.g., how they appear on a spectrum), and assign consecutive natural numbers. Assuming we have 30 different colors, *white* would be 1 and *black* would be 30. Similar assignments can be made for other non-numeric variables. Note that the variables “mileage,” “year,” and “damage level” are already numeric, so there is no need to convert these.

Because a linear regression model must answer (i.e., produce a value for) questions such as: “What’s the price of a Toyota (“make”) Camry (“model”) at auction site Jacksonville, Florida (“location”)?”⁴ we need to develop a function:

$$\text{Sale Price} = a + (b \times \text{Mileage}) + (c \times \text{Year}) + (d \times \text{Color}) + \dots$$

that provides the predicted price for a new case (i.e., a used Toyota Camry) when supplied with the numeric values of the other variables (“mileage,” “year,” “color,” etc.). The main challenge here is to find the values for parameters *a*, *b*, *c*, *d*, etc. that give the prediction model the best possible performance (i.e., that minimize the predictive error). Since we have all the historic data from three million cases, we can extract all cases where “location” = Jacksonville, “make” = Toyota, and “model” = Camry. This subset of cases (say we identified 150 such cases) would constitute the data set available for training the prediction model (some of

³ Note also that in some situations we would like to predict only one of two values (“yes” or “no,” “fraudulent” or “legitimate,” “buy” or “sell,” etc.). This type of regression is called logistic regression, and a similar methodology is applied (e.g., transformation of variables, building a linear model).

⁴ Note that the Jacksonville location would contain many prediction models (for all distinct pairs of make/model).

these cases would also be used for validation and testing; see Sect. 5.3 for more details on this).

To minimize the error on the training set, there are several standard procedures for determining the parameter values. Once these parameters are determined, the prediction model (for all Toyota Camry cars sold at the Jacksonville auction) is ready. For every new case (again, by *new* case we mean a *used* Toyota Camry), we can determine the sale price for the Jacksonville location by inserting the appropriate values for "mileage," "year," "color," etc. into the sale price function.

Note, however, that the training process might not be that simple (this is true for any prediction model, not just linear regression). First of all, some values might be missing (e. g., the mileage was not recorded). In such cases we can:

- Remove the case from consideration and contact the appropriate auction site to recover the mileage value. Once this value is recovered, we can insert the case back into the system for processing. Although this would cause a delay in processing the car, it might prevent us from making a serious prediction error.
- Estimate the mileage on the basis of other variables. For example, if the car was "leased," it might be reasonable to assume that the average mileage allowance is 12,000 miles per year. Thus, a three-year-old car is likely to have 36,000 miles.

Second, because the prediction model has to provide more than just tomorrow's price (as it takes some time to transport the car to Jacksonville, and so we need a predicted price for next week and/or three weeks from today), the training process might be much more complex. The reason for this increased complexity is hidden in the fact that the prediction model's accuracy must be assessed for both shorter and longer time periods. Hence, the process of searching for the best prediction model is more difficult, as it is harder to compare and select the better of two models where one provides better short-term predictions while the other provides better longer-term predictions. This is a typical multi-objective optimization problem (as discussed in Sect. 2.4).

Third, from time to time the linear regression model would process a "rare" car, such as a Dodge Viper or Acura NSX. Note that we assumed a linear regression model for each make/model at each location. This assumption is fine, but the historical data set may only contain 100 Dodge Viper cars with zero occurrences at some locations! How can we build a model for a location where the data set is empty? Well, as usual, there are several ways of dealing with this problem. One way would be to estimate the price on the basis of (1) prices of the same make/model at nearby locations, and (2) prices of similar models at the same location. This approach would require some additional, problem-specific knowledge. Another possibility would be to use an approach based on agent modeling (Chap. 9), which can be used as a data mining technique for "data-less" problems!

The above example serves to underline the simple fact that *the devil is in the detail*. This is true for any prediction method, because developing a prediction model for a real-world problem usually involves the resolution of many issues ranging from incomplete information to insufficient data. Something else to consider is that regression might be far more complicated than our simple example. Note that the prediction model above has one powerful disadvantage: it is *linear*! Real-world

data often display nonlinear dependencies that we would like to capture (recall the nonlinear transportation model in Sect. 2.5). Of course, a linear regression model would find the best possible line, but the line may not fit very well.

One approach to this problem is to replace the line with a curve, which can be done by transforming the variables (by multiplying some of them together, squaring or cubing them, or taking their square root). After completing these transformations, we can then determine the new parameters (i.e., a , b , c , d , etc.) of the prediction model (although this new model is more complex and we are now talking about *nonlinear* regression). It is possible to experiment with a wide variety of transformations, and if they do not provide a meaningful contribution to the prediction model, then their parameters will stay close to zero. The difficulty, however, is that the number of possible transformations might be too prohibitive (i.e., the number of possible parameters to explore might be too high, and any training would be infeasible). Moreover, with complex transformations we should guard against overfitting,⁵ as the use of complex transformations guarantees high precision on the training set that may not carry over to new predictions.

Now let us turn our attention to time series problems. As mentioned earlier, the only purpose of a time series model is to predict future values; the relationships between the variables are of no interest. The problem might be expressed as follows:

Given $v[1]$, $v[2]$, ..., $v[t]$, predict the values of $v[t+1]$, $v[t+2]$, ..., $v[t+k]$

where t is the present time interval, $t-1$ is the previous time interval, $t+1$ is the next time interval, and so on. If we are only predicting the next interval ($t+1$), then a time series model is concerned with a function F such that:

$$v[t+1] = F(v[1], v[2], \dots, v[t])$$

Note that the above function may include some other variables, and not just the values of variable v from earlier time intervals. In such cases, we talk about *composite forecasting models*, which consist of past time series values, past variables, and past errors.

Many statistical time series models have been proposed during the last few decades, including exponential smoothing models, autoregressive/integrated/moving average models, transfer function models, state space models, and others. Each model is based on some assumptions, and involves a few (at least one) parameters that must be tuned on the basis of historical data.

Now let us consider the category of prediction methods that are collectively known as "exponential smoothing." These methods generalize the *moving average method*, where the mean of past k cases is used as a prediction. All exponential smoothing methods assign weights to past cases in such a way that recent cases are given more weight than the older cases (as the more recent cases usually provide better future direction than the less recent ones). Hence, it is reasonable to develop a weighting scheme that assigns smaller weights to older cases. Such a weighting

⁵ *Overfitting* occurs when a model tunes itself during the training stage to such an extent that all predictions on the training data set are perfect.

scheme also requires at least one parameter a . For example, a prediction for the time $t+1$ is calculated as:

$$\text{Prediction}(t+1) = (a \times \text{Actual}(t)) + ((1-a) \times \text{Prediction}(t))$$

which simply means that the prediction for the next (future) case is calculated as a total of two values: the actual last case ($\text{Actual}(t)$) with parameter a and the last prediction ($\text{Prediction}(t)$) with the weight $1-a$. Note that parameter a provides the significance of the last case in making the prediction; in particular, if $a=1$, then the prediction would always report the last actual value as a new prediction. It is easy to generalize this method to include more past cases:

$$\begin{aligned} \text{Prediction}(t+1) = & (a \times \text{Actual}(t)) + (a \times (1-a) \times \text{Actual}(t-1)) \\ & + (a \times (1-a)^2 \times \text{Actual}(t-2)) + (a \times (1-a)^3 \times \text{Actual}(t-3)) + \dots \\ & + (a \times (1-a)^{t-1} \times \text{Actual}(1)) + ((1-a)^t \times \text{Prediction}(1)) \end{aligned}$$

so $\text{Prediction}(t+1)$ represents a weighted moving average of all past observations. Note again, that different values of parameter a would result in a different distribution of weights. Also, it was assumed that the prediction horizon was just one period away ($t+1$). For longer-term predictions, it is often assumed that the function is flat:

$$\text{Prediction}(t+1) = \text{Prediction}(t+2) = \text{Prediction}(t+3) = \dots$$

as exponential smoothing works best for data that have no trend or seasonality. However, since some form of trend or seasonality exists in most data sets, *decomposition* methods can be used to identify the separate components of the underlying trend-cycle and seasonal factors. The trend-cycle (which is sometimes separated into trend and cyclical components) represents long-term changes in the time series values, whereas seasonal factors relate to periodic fluctuations of constant length caused by phenomena such as temperature, rainfall, holidays, etc.

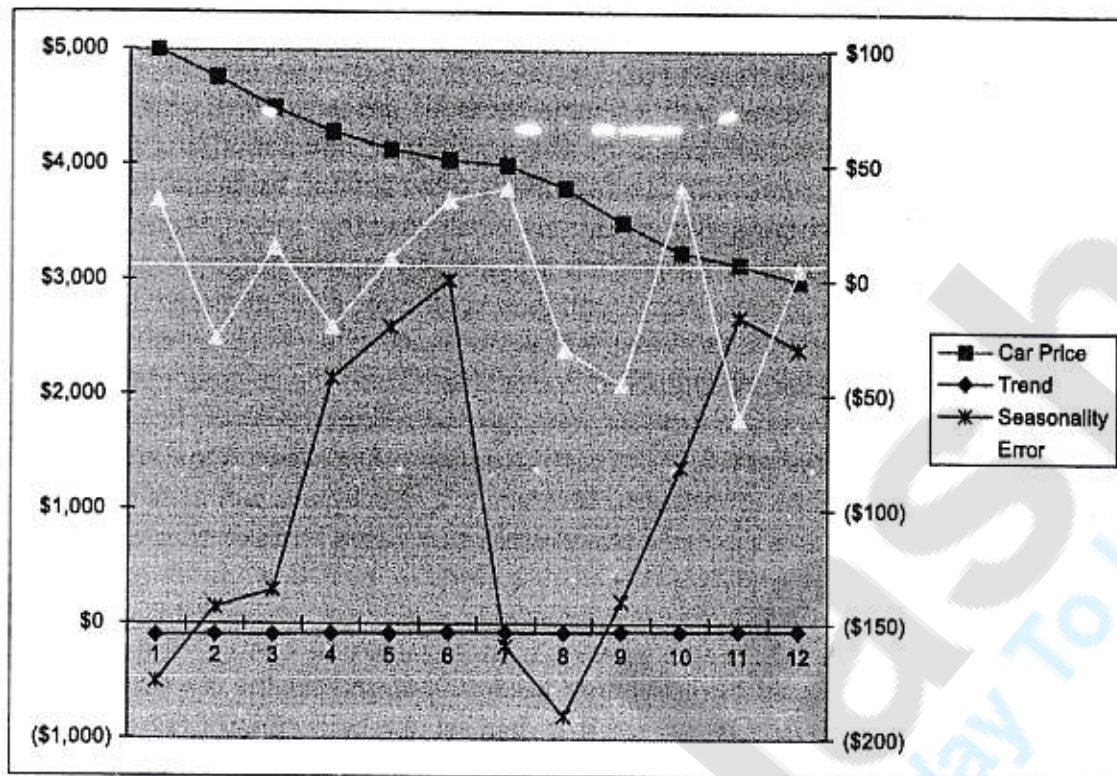
Although there are several approaches to decomposing a time series problem into separate components, the basic concept is based on experience: First the trend-cycle is removed, then the seasonal components are addressed. Any remaining error is attributed to randomness; thus:

$$\text{Data} = \text{trend-cycle} + \text{seasonality factors} + \text{error}$$

Note that the relationship between the data and trend-cycle, seasonality factors, and error need not be linear (additive, as above); in general, decomposition methods search for a function D that would “explain” a data point at any time t :

$$\text{Data}(t) = D(\text{trend-cycle}(t), \text{seasonality factors}(t), \text{error}(t))$$

The figure below illustrates what such a decomposition of data might look like for the car distribution example:



First of all, note that the "Car Price" (Data) corresponds to the left y-axis, while the "Trend", "Seasonality" and "Error" correspond to the right y-axis, and the x-axis represents the month. In general, the "Trend" is a continued decrease of the "Car Price," while "Seasonality" can have a negative effect or no effect at all. The "Error" can be positive or negative. Let us take June (month "6" in the figure above) as an example: The "Car Price" is \$4,045, the "Trend" during June is a decrease of \$152, the "Seasonality" effect is \$0, and the "Error" is \$35. If we add up all these numbers then we get a "Car Price" of \$3,928 for the beginning of July.

Going back to exponential smoothing, the relationship between the past and future is linear, but this might not be appropriate for many real-world applications of time series. Linear models cannot capture some features that commonly occur in actual data, such as asymmetric cycles (which are data patterns in which the period of repeating cycles is not fixed, and the average number of data on the up-cycle is different than the average number of data on the down-cycle) and occasional outliers. Although linear methods often deal with nonlinear time series by logarithmic or power transformations of data, these techniques do not account for asymmetric cycles and outliers.

Some nonlinear methods assume that asymmetric cycles are caused by distinct underlying phases of the time series, and that a transition period (either smooth or abrupt) exists between these phases. The individual phases are usually given a linear functional form, and the transition period (if smooth) is modeled as an exponential or logarithmic function. Some other methods are used to deal with time series that display variable variance of residuals (error values). In these methods,

the variance of error values is modeled as a quadratic function of past variance values and past error values.

Although all these linear and nonlinear methods are capable of characterizing the variables found in actual data, they also assume that the underlying process of data generation is constant. This assumption is often invalid for actual time series data, as changing environmental conditions may cause the underlying data generating process to change. For all prediction methods, human judgment is required to first select an appropriate method, and then set the appropriate parameter values for the model's parameters. In the event that the underlying data generating process changes, the time series data must be reevaluated and the parameter values re-adjusted (in extreme cases, a new model might be required). We will address the issue of adaptability in Sect. 10.3, as well as a few other subsections in Part II of this book.

5.2.2 Distance Methods

Another method for building prediction models is based on the concept of "distance between cases." Any two cases in a data set can be compared for similarity, and this similarity measure (called "distance") is assigned some value: the more similar the cases, the smaller the value. Using a distance measure within a data set would allow us to compare a new case with the most "similar" existing case. The outcome of the most similar case (e.g., the loan was repaid, the transaction was fraudulent) would be the prediction for the new case. Going back to our example of the Toyota Camry at the Jacksonville auction site, we may search our database of three million cases for the most similar Toyota Camry sold in Jacksonville and use its sale price as our prediction. Ideally, the existing case would be recent and have the same mileage, color, trim, etc. as the new case. Hence, instead of building a function where the variable values (magnified by some weights) determine the outcome, we just keep the past cases.

The essential aspect of this approach is creating a similarity measure between cases, because the probability of finding an identical case is very low. Hence, we have to base our decisions on similarities, which is far from trivial: For instance, is a silver Toyota Camry with 33,000 miles "more similar" to a white Toyota Camry with 34,100 miles, or to a silver Toyota Camry with 36,000 miles? Or, is the difference in "similarity" between "silver" and "white" the same as between "red" and "yellow"? To answer such questions, it is necessary to define some *distance* between cases (again, the shorter the distance, the greater the similarity).

One of the most popular distance-based prediction methods is *k* nearest neighbor, where *k* nearest neighbors (i.e., *k* most similar cases) of a new case are determined. Clearly, if $k = 1$ (i.e., we find only one neighbor), the outcome of this single neighbor is the prediction for the new case. If $k > 1$, then a voting mechanism is used (classification problems) or the average value of the *k* answers is calculated (regression problems).

Note, however, that the most important step of the *k* nearest neighbor method is calculating the distance between cases – this is crucial for getting high-quality

results. There are many ways of defining a distance function, but experimentation is often the best way. In any case, careful data preparation is always the first step (it is likely that the data will be normalized to equalize the scale for computing distances and/or some weighting will be applied where different variables get different weights). Note that calculating the distance is trivial when there is only one numeric variable, (e. g., $5.7 - 3.8 = 1.9$). With several numerical variables, a Euclidean distance⁶ can be used, provided that the variables are normalized and of equal importance (otherwise a weighting must be applied).

The largest problem, however, is with nominal variables. Given our earlier question of whether “the difference in similarity between ‘silver’ and ‘white’ is the same as between ‘red’ and ‘yellow’?” we can assume that different colors are just different (resulting in a distance of 1), or we can introduce a more sophisticated matrix that would assign a numeric measure for each color (e. g., so that the difference between “light blue” and “dark blue” is smaller than the difference between “blue” and “red”). These are the two standard approaches for evaluating differences between the values of nominal variables.

Another issue to consider is that of missing values. A standard approach is to assume that the distance between an existing value and a missing value is as large as possible. Hence, for nominal values, the distance is assigned a normalized value of 1 (all distances are between 0 and 1), and for numeric variables the distance is assigned the largest possible normalized value between 0 and 1. For example, if an existing value is 0.27 and the other value is missing, then the distance is 0.73; if the existing value is 0.73 and the other value is missing, then the distance is also 0.73.

Yet another issue is the number of stored cases. A distance-based method might be too time consuming for large data sets, because the whole data set must be searched to evaluate each new case. With larger values of parameter k , the computation time increases significantly. For efficiency reasons, it would be beneficial to reduce the number of stored cases. By selecting a subset of “representative cases,” the process of finding the closest neighbor (or neighbors) might be more efficient. And to make the representative cases as “representative” as possible (i. e., as good as possible), a new set of representative cases can be selected from the current representative cases and all misclassified cases that produced a prediction error larger than some threshold. In other words, the current representative and misclassified cases could constitute an input for some reclassification method (e. g., decision trees), which would be responsible for creating a better set of representative cases.

Also, some clustering methods can be used to group the cases into meaningful categories. A new case would then be assigned to an existing category and the predicted value would be drawn from the cases present in that category (again, by voting for classification, or averaging for regression). Note that it is not necessary to store all the cases per category; again, we can select some representative cases instead. A few clustering techniques might be considered for this task

⁶ *Euclidean distance* is defined as the length of a line segment between two points in an n -dimensional space. In particular, the distance d between two points (x_1, y_1) and (x_2, y_2) in a 2-dimensional space is determined by the following function: $d^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2$.

(e. g., *k*-means algorithm, incremental clustering, or statistical clustering based on a mixture model), which we will discuss later in the text.

5.2.3 Logic Methods

A *decision table* (also known as a *lookup table*) is the simplest logic-based method for prediction, and there are many such tables published for estimating the price of a used car sold at auction (e. g., *Black Book*, *Kelley Blue Book*, *Manheim Market Report*). In these tables, we can locate the appropriate make/model/year/body style, get a basic price, and adjust this price for additional variables such as mileage, color, trim, damage level, etc. However, not all variables are included (e. g., for some makes/models, *color* might not be included).

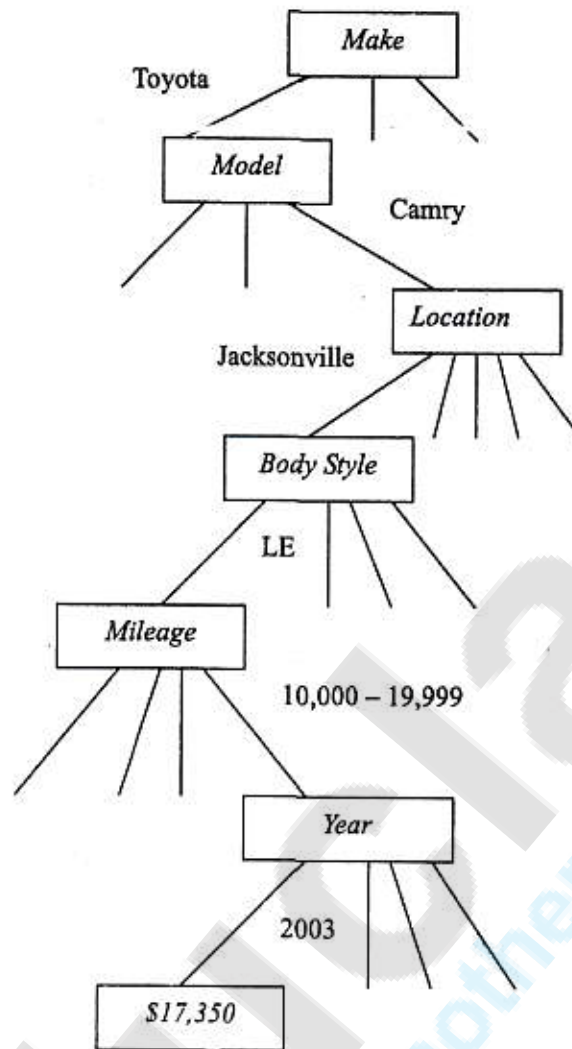
The most widely used logic method, on the other hand, is the *decision tree*. Because the structure of a decision tree is relatively easy to follow and understand (especially for smaller trees), its popularity is widespread. To make a prediction for a new case, the root of a tree is examined, a test is performed,⁷ and, depending on the result of the test, the case moves down the appropriate branch. The process continues until a terminal node (also known as a “leaf”) is reached, and the value of this terminal node is the predicted outcome.

Although decision trees are used for all types of predictions problems, they are especially popular for classification problems. If the test involves a nominal variable, the number of branches corresponds to the number of possible values that variable can take (i. e., there is one branch for each possible value). If the test involves a numeric variable, there are usually two branches, as the test determines whether the value is “greater than” or “less than” (possibly also “equal to” for integer numbers) some predefined fixed value.⁸ In the case of missing values, an additional branch is assigned or some other heuristic is used (e. g., selection of the most popular branch or selection of a few branches).

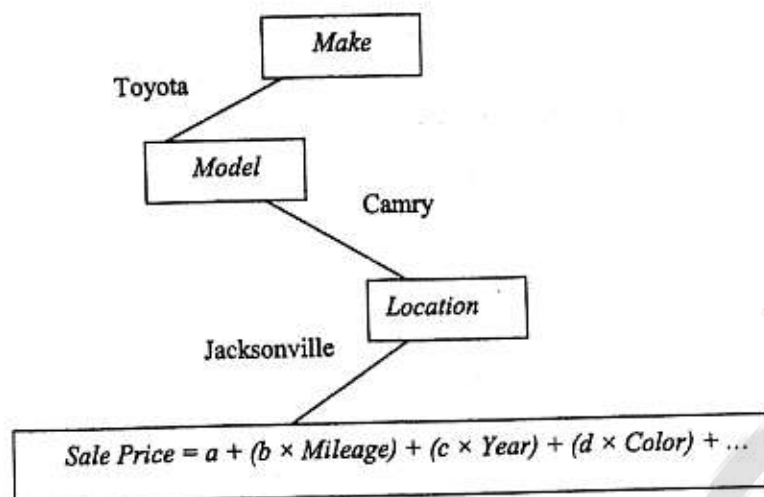
We can easily picture a decision tree for used car prices. At the root of the tree a decision is made on “make”: if there are 30 different makes, then the root node would have 30 branches. On the second level of the tree, a decision is made on “model,” then the third level provides branches for “location.” Further down, we may have nodes that test a new case for “body style” and “mileage” and refer it to the appropriate branch. For example, a test on “mileage” might involve the selection of an appropriate category (e. g., “0 to 9,999 miles,” “10,000 to 19,999 miles,” and so on). The following illustrates the branch of a simplified decision tree:

⁷ By “test” we mean that a node compares a value of a variable with some constant. However, it is possible to include more sophisticated tests, where more variables and/or additional functions are involved.

⁸ It is also possible to have a decision tree with more than two branches for a numeric variable, where a range of values is assigned to each branch.



Naturally, there are better and more sophisticated ways to use decision trees for numeric prediction. It might not be practical to represent every value (or range of values) as a separate branch in a decision tree, as the size of the tree might be too large. Instead of keeping a single, numeric value at each terminal node (as illustrated above), it might be easier to keep a model (e. g., a linear regression model) that predicts a value for all cases that reach this terminal node. Such a tree could be used to answer our question from Sect. 5.2.1: “What’s the price of a Toyota (“make”) Camry (“model”) at auction site Jacksonville (“location”)?” The variables “make,” “model,” and “location” are used for building the tree (and later for branch determination when processing a new case), whereas mileage, year, color, etc. are used as variables in the linear regression model at each terminal node, as illustrated below:



As before, the predicted sale price might be adjusted further to take into account the time factor, as it would take some time to transport the car to Jacksonville. Note that the number of parameters (a, b, c, d , etc.) and their values might be different at each terminal node.

To achieve better prediction accuracy, it might be worthwhile to build a linear regression model for each node of the tree (rather than just for the terminal nodes).⁹ Note, however, that the root node would now have a function that involves all the variables:¹⁰

$$\text{Sale Price} = a + (b \times \text{Make}) + (c \times \text{Model}) + (d \times \text{Location}) + (e \times \text{Mileage}) + (f \times \text{Year}) + (g \times \text{Color}) + \dots$$

For second-level nodes, the linear function will not include the variable "make," because the appropriate branch of the decision tree has already been selected. Thus, the linear function would be:

$$\text{Sale Price} = a + (b \times \text{Model}) + (c \times \text{Location}) + (d \times \text{Mileage}) + (e \times \text{Year}) + (f \times \text{Color}) + \dots$$

For third-level nodes (where the decision for make and model has already been made), the function would be:

$$\text{Sale Price} = a + (b \times \text{Location}) + (c \times \text{Mileage}) + (d \times \text{Year}) + (e \times \text{Color}) + \dots$$

and so on. (Note, however, that the parameters a, b, c , etc. are different in all these functions.) In our earlier diagram, the terminal node was placed on the fourth level with a linear function of:

$$\text{Sale Price} = a + (b \times \text{Mileage}) + (c \times \text{Year}) + (d \times \text{Color}) + \dots$$

⁹ Experimental evidence shows that prediction accuracy can be increased by combining several prediction models together (we will discuss this in Sect. 10.1).

¹⁰ This approach usually involves nominal attributes as well (e.g., make, model, location) as all the variables are represented on different levels of the decision tree. Such nominal variables are transformed into binary variables and treated as numeric.

To compensate for the differences between “adjacent” linear models at the fourth level, some averaging (also called smoothing) can be applied when processing a new case. Instead of using the predicted value from the terminal node, the predicted value can be “filtered” back up the tree and averaged at each node by combining the predicted value from a lower level with the predicted value from the current level. This usually improves the accuracy of predictions.

Another logic method is based on *decision rules*, which are “similar” to decision trees: after all, a decision tree can be interpreted as a collection of rules. For example, the single branch of the decision tree displayed earlier can be converted into the following rule:

if Make = Toyota & Model = Camry & Location = Jacksonville
& Body Style = LE & $10,000 \leq \text{Mileage} \leq 19,999$ & Year = 2003,
then Sale Price = \$17,350

The “if” parts of a rule (e. g., “model” = Camry) are combined logically together by the “and” (&) operator, and all the tests must be true if the rule is “to fire” (i. e., for the conclusion of the rule to be applied: *Price* = \$17,350). Note that there must be several decision rules in the system (the above rule represents a single branch of a tree) and we can interpret this collection of rules as connected through the “or” operator: if one rule applies to a new case, its conclusion is taken as the predicted outcome. If two (or more) rules fire, then we can combine the conclusions of these rules to determine the final predicted outcome. The other (in some sense, opposite) problem can arise if *no* rules fire for a new case! As usual, some standard remedies exist, such as the creation of a default rule that will always fire

if $0 \leq \text{Mileage} \leq 999,999$, **then** Sale Price = \$15,000

which is the overall average price of a used car. Of course, one can question the usefulness of such a rule ...

These two simple cases, when two or more rules fire or no rules fire, illustrate the point that rules can be difficult to deal with. The reason is that each rule represents a separate “piece” of knowledge and *all* the rules together operate as one system (often called a *rule-based system*). Thus, it is essential to understand the consequences of adding or dropping a rule in the system. This is important in many practical situations, where experts add their own rules (from experience) to the data-generated rules. Although dropping and adding rules in a rule-based system is not a trivial task, it is much easier to drop or add a rule than to modify an entire decision tree by cutting or adding some new branches. Hence, each method has its own advantages and disadvantages.

As mentioned earlier, classification problems have been the focus of data mining research for the last few decades, and the creation of decision rules¹¹ has been the most popular approach for addressing these problems. Several aspects of generating rules from data have been investigated, including:

¹¹ A decision rule for a classification problem is often called a *classification rule*.

- *Association rules*, which describe some regularity present in the data and can “predict” any variable (rather than just the class). For example, an associate rule may state that

if Make = Porsche & Model = Carrera,
then Location in {Jacksonville, Tampa, Los Angeles, San Francisco,
 San Diego}

as Porsche Carreras are sold only at auction sites in Florida and California.

- *Rules with exception*, which extend a rule with exceptions, may refer to association rules, e. g.,

if Make = Porsche & Model = Carrera, **then** Location in {Jacksonville,
 Tampa, Los Angeles, San Francisco, San Diego}, **except if** Year \leq 1997,
then Location in {Austin, Houston, Dallas}

which states that older Porsche Carreras (produced in 1997 or earlier) are sent to auction sites in Texas; or to a classification rule, e. g.,

if Make = Toyota & Model = Camry & Location = Jacksonville
 & Body Style = LE & $10,000 \leq$ Mileage \leq 19,999 & Year = 2003,
then Sale Price = \$17,350, **except if** Color = Red, **then** Sale Price = \$18,450

as red was a rare (but popular) color for Toyota Camry cars in 2003 and that increases the price.

Rule-based systems, which consist of a collection of rules and an inference system,¹² are quite popular, because each rule specifies a small piece of knowledge and people are good at handling small pieces of knowledge! Separate rules can be discovered from data mining activities or interviewing experts, and instead of specifying the overall model only the decision rules and inference system are needed. Note that the rule-based system will try to behave like an expert, performing some reasoning on the basis of the knowledge present in the system.

5.2.4 Modern Heuristic Methods

As indicated earlier, a few prediction methods fall into the category of “modern heuristics”; these include fuzzy systems, neural networks, genetic programming, and agent-based systems. These methods originated in different research communities, and their “mechanics” are very different to classic methods such as statistics and machine learning. Because these prediction methods are of growing importance, we will discuss them in detail in Chaps. 7–9.

¹² An *inference system* is responsible for putting the decision rules in the appropriate order and combining the outcomes of the rules that fired. It may also contain strategies and controls that are typically used by experts.

5.2.5 Additional Considerations

Many other considerations must be taken into account when selecting the “best” prediction method for an Adaptive Business Intelligence system. Although the prediction error is quite possibly *the most* important measure, it only provides one dimension of a model’s quality. For real-world business problems, many other factors must be considered, such as:

- *Response time.* This is an essential consideration, as any Adaptive Business Intelligence system would have a defined response time. Fraud detection systems, for example, process millions of transactions per second, so the frequency of predictions (i. e., classifications of “fraudulent” or “legitimate”) is very high. Other prediction methods, on the other hand, might be used on a weekly basis (e. g., inventory management) and so the response time is not that critical.
- *Editing.* Some prediction models are difficult to edit (e. g., neural networks), while others (e. g., rule-based systems) are easy. The ability to edit a model is an important consideration, as it might be necessary to add the knowledge of experts to the final model.
- *Prediction justification.* This is an often-overlooked aspect of evaluating the usefulness of a prediction model. For some applications (e. g., credit scoring) it is very important to justify the prediction; in some cases, this might even be required by law (e. g., justification for rejecting a loan application).
- *Model compactness.* A prediction model should not be exceedingly large and complex, as that would make it difficult for humans to understand; also, it might take a longer amount of time to make predictions. According to the principle of “Ockham’s Razor,” a more compact prediction model is preferable over a sprawling prediction model assuming they both do an equally good job of predicting.
- *Tolerance for noise.* All prediction methods require some approach for handling missing values (e. g., the mileage of an off-lease car has not been recorded), but some methods do a better job of handling missing values than others. Also, some values might be present, but noisy (i. e., imprecise) – like stating that the color of a car is “dark” ...

Because of these many factors, it may be difficult to select “the best” prediction method for the problem at hand. Different prediction methods have different properties, and so some of them may perform better or worse when trained on different data sets. Hence, it might be worthwhile to use a few methods to build a few models, and then use all the models to reach a consensus. We will explore this *hybrid systems* approach to prediction in Sect. 10.1.

5.3 Evaluation of Models

Although it is possible to use a variety of different prediction methods to build a variety of different prediction models, the key issue is which method should be

account. Although many error measurement techniques exist (e.g., mean-squared error, mean absolute error, relative squared error, relative absolute error), it is much harder to measure the consequences of an error. For example, the error in a price prediction of a used car for a particular location might only be \$150 (approximately 1% of the car's value), but this error may influence the distribution decision, which in turn influences the transportation decision and distributions of other cars (because of the volume effect)!

Because we are interested in the *future* performance of a prediction model – i.e., performance on *new* data, not performance on the training data – we cannot take a model's performance (or error rate) on the training data (i.e., *old* data) as a foolproof indicator of its performance on new data. The reason for this is very simple: The most "reliable" prediction model would be a simple lookup table where all the previous cases are stored. Such a model will score exceptionally well on old cases ... Unfortunately, this score will tell us very little about the model's performance on new data! Most prediction models can be overtrained in the sense that they would behave in a similar way to a lookup table. Hence, a model's performance on the training data set will always be better than the model's true performance ...

To predict a model's performance on new data, we need another data set (usually called a *test set*) that did not participate in the building, training, and tuning of the model. This is important: we need *fresh* data to evaluate the performance of a prediction model. The most popular way of doing this (when there is enough data) is to randomly divide the original data set (i.e., available cases) into a training set and testing set. The prediction method then uses the training set to select variables, compose additional variables, calculate ratios, parameters, etc., but it does not have access to the test set. Once the prediction model is created on the basis of the training data set, it can be fairly evaluated for performance on the test data set.

In many cases, the process of building a prediction model consists of two phases: (1) constructing a model, and (2) tuning the parameters of the model. For this reason, it is also convenient to further split the training data set into two subsets: the primary training set and a *validation set* – the former for building the model, the latter for tuning its parameters. So, altogether, it is convenient to have three independent data sets (the third one being the test data set, which is used to evaluate the model's performance). Each of these three data sets should be selected independently, and each of them plays an important, independent role:

- The training data set is used for building a prediction model.
- The validation data set is used for tuning the parameters of the model (i.e., for optimizing the performance of the model).¹⁴
- The test data set is used to evaluate the performance of the model.

If we had plenty of data for training, plenty of data for validation, and plenty of data for evaluation, then the result should be a better model. However, if there is

¹⁴ If several prediction models were constructed from the training data set, then the validation data set is sometimes used for selecting the best model.

account. Although many error measurement techniques exist (e. g., mean-squared error, mean absolute error, relative squared error, relative absolute error), it is much harder to measure the consequences of an error. For example, the error in a price prediction of a used car for a particular location might only be \$150 (approximately 1% of the car's value), but this error may influence the distribution decision, which in turn influences the transportation decision and distributions of other cars (because of the volume effect)!

Because we are interested in the *future* performance of a prediction model – i. e., performance on *new* data, not performance on the training data – we cannot take a model's performance (or error rate) on the training data (i. e., *old* data) as a foolproof indicator of its performance on new data. The reason for this is very simple: The most "reliable" prediction model would be a simple lookup table where all the previous cases are stored. Such a model will score exceptionally well on old cases ... Unfortunately, this score will tell us very little about the model's performance on new data! Most prediction models can be overtrained in the sense that they would behave in a similar way to a lookup table. Hence, a model's performance on the training data set will always be better than the model's true performance ...

To predict a model's performance on new data, we need another data set (usually called a *test set*) that did not participate in the building, training, and tuning of the model. This is important: we need *fresh* data to evaluate the performance of a prediction model. The most popular way of doing this (when there is enough data) is to randomly divide the original data set (i. e., available cases) into a training set and testing set. The prediction method then uses the training set to select variables, compose additional variables, calculate ratios, parameters, etc., but it does not have access to the test set. Once the prediction model is created on the basis of the training data set, it can be fairly evaluated for performance on the test data set.

In many cases, the process of building a prediction model consists of two phases: (1) constructing a model, and (2) tuning the parameters of the model. For this reason, it is also convenient to further split the training data set into two subsets: the primary training set and a *validation set* – the former for building the model, the latter for tuning its parameters. So, altogether, it is convenient to have three independent data sets (the third one being the test data set, which is used to evaluate the model's performance). Each of these three data sets should be selected independently, and each of them plays an important, independent role:

- The training data set is used for building a prediction model.
- The validation data set is used for tuning the parameters of the model (i. e., for optimizing the performance of the model).¹⁴
- The test data set is used to evaluate the performance of the model.

If we had plenty of data for training, plenty of data for validation, and plenty of data for evaluation, then the result should be a better model. However, if there is

¹⁴ If several prediction models were constructed from the training data set, then the validation data set is sometimes used for selecting the best model.

only a limited amount of data, then what can be done to maximize them? Note again that the general idea is to split the data: some data (usually two thirds) are used for training (this includes validation), and some (usually one third) for testing.

The first issue to consider here is whether each subset is a "representative" sample of the entire set. For example, it may happen that the training data set has no "yellow" cars, while the test data set contains many yellow cars. If a category is missing in the training set, then the prediction model might have serious difficulties in predicting the "right" value for this category (as the "learning" process is based on data). Moreover, the evaluation of the prediction model would be biased, as all (or most) cases of the category in question (e. g., "yellow" cars) would appear only in the test data set!

Clearly, it would be beneficial to "guarantee" that the distribution of cases is uniform across all data sets. One way of approaching this problem is through *stratification*: the algorithm that splits the data into training and testing subsets ensures that the sampling is done in such a way that each category is properly represented. The other approach is repeating the training and testing phases with different data sets, and then averaging the performance of the prediction model from all the iterations. A popular statistical technique, called *cross-validation*, is often used in connection with the latter approach. In this technique, we divide the data set into some number (say k) of disjointed subsets (called *folds*). Then $k - 1$ folds are used for training and one for testing, and we can repeat this process k times, each time with a different group of folds selected for training and a different fold for testing. If $k = 3$ (i. e., the data set is partitioned into three subsets), then the technique is called *three-fold cross-validation*. It is quite common to use $k = 10$ (*10-fold cross-validation*),¹⁵ as 10 is a reasonable number of folds to get a good estimate of the prediction error.¹⁶

One extreme (and, in many cases, useful) application of the cross-validation technique is when the number of folds equals the number of cases in the data set (this approach is called the *leave-one-out* approach). In a database with three million cases, there would be three million folds. Hence, we would repeat the following process three million times: A prediction model is built on a training data set of 2,999,999 cases, and the error estimate is made on the remaining single case. Then the average of all errors will give us the error estimate for the prediction model. In this technique, the greatest possible amounts of data are used for training, and, because the approach is deterministic, there is no need to repeat the process. However, the computational overhead might be too large for large data sets.

The final model evaluation technique we will mention is the *bootstrap*, which has a reputation for being one of the best techniques when the data set is very small. In the bootstrap technique, a collection of cases is selected as the training

¹⁵ *10-fold cross-validation* is often used with stratification. Stratified 10-fold cross-validation is generally held as a standard evaluation technique in cases where the amount of data are limited.

¹⁶ This estimation, however, need not be perfect, as different fold selections may give different error estimates. Thus, it is a standard procedure to repeat the cross-validation process 10 times, which results in building and testing a prediction model 100 times altogether.

set *with repetition*. Further, the number of cases in the training set is the same as the total number of cases available. By doing this, some cases will be selected more than once, while some cases will not be selected at all! It is relatively easy for a mathematician to calculate the probability of a case *not* being selected for the training set by dividing the constant e by 1, which equals to $0.36787944117 \approx 0.368$. This means that approximately 36.8% cases *will not* be selected, and 63.2% of cases will be selected (once or more than once).¹⁷ If we apply the bootstrap technique to our data set of three million used-car cases, then approximately 1,896,362 cases would be selected (once or more than once) for the training data set, whereas the remaining 1,103,638 cases would constitute the test data set. As with cross-validation, the bootstrap procedure is usually repeated several times with different samples.

For a moment, let us return to the issue of time dependencies in the data set. As mentioned earlier, most real-world business problems have some time-dependent relationships within their data sets: Transactions, orders, deliveries, sales – all of these have a time stamp. And because these data sets will inevitably change, the problem lies in not knowing how they will change! Also, some data sets change very quickly (e.g., the closing prices of all stocks in the S&P 500 index), while others change very slowly (e.g., the average income in a particular region). As a matter of fact, some changes are so slow that we consider the data set to be stable, even though small changes are constantly taking place. In any case, it is important to select the appropriate sampling technique when dividing the original data into training and testing sets. It is also essential to organize the cases in such a way that all the training cases have an earlier timestamp than the testing cases. This is done so that the predictions go from “past” to “future.” In other words, we should identify a particular point of time, and take all relevant *preceding* cases for the training set and all relevant *subsequent* cases for the testing set. Note also, that the time dependencies among cases might be so strong that we should treat the data set as a time series, where all cases are kept in a sequential time order.

The inevitable changes that occur in a data set – from which we are supposed to create a prediction model – have powerful consequences. If the changes are slight, then the sampling and evaluation techniques discussed in this section would work. However, if the changes are significant (like after a major stock market crash or natural disaster), then it might necessary to build a new model altogether. Also, as we saw in Sect. 5.2, different prediction methods produce different models of varying complexity. For this reason, it might be safer to select a simpler model that has a higher degree of generality (allowing for better adaptation to small changes that occur in the data set). Another approach (which we will discuss in Sect. 10.3) would be to use an adaptability module to adjust the various parameters of the model.

¹⁷ Because 63.2% of the cases (on average) will be selected for the training set, the method is also called the *0.632 bootstrap*.

5.4 Recommended Reading

In this chapter we gave a general overview of many different types of prediction problems (e. g., classification, regression, time series), methods (quantitative or qualitative), and processes (data preparation, data mining, model building, deployment and evaluation). Because the ultimate goal of any prediction model is to predict the “outcome” of a new case, we also discussed a variety of prediction models based on mathematics, distance, and logic. Our discussion on prediction methods will continue in Chaps. 7–9, where we will present several modern prediction methods, including artificial neural networks, fuzzy logic, and agent-based modeling. Lastly, in Chap. 10 we will discuss the concept of using several prediction models together, along with the role of the adaptability module.

There are a variety of texts available that discuss data mining techniques. The book *Predictive Data Mining* by Sholom M. Weiss and Nitin Indurkha (Morgan Kaufmann, San Francisco, 1998) provides an excellent high-level discussion on most of the topics presented in this chapter (e. g., preparation of data, data reduction, types of solutions), with an additional discussion on data mining and statistical methods, and several case studies.

A slightly more technical introductory text to data mining techniques is *Data Mining: Practical Machine Learning Tools and Techniques* by Ian H. Witten and Eibe Frank (Morgan Kaufmann, San Francisco, 2000). The book presents many algorithms for extracting and validating various models (e. g., decision trees, rules, linear models) from data. The book also provides Java data mining tools that the authors made available through their website.

More advanced texts include *Machine Learning and Data Mining: Methods and Applications* edited by Ryszard S. Michalski, Ivan Bratko, and Miroslav Kubat (Wiley, Chichester, 1998). This volume provides a detailed treatment of many specific topics (e. g., multi-strategy approach, inductive logic programming) as well as discussions on data mining applications in pattern recognition, design, engineering, control systems, medicine, and biology.

Further, there are texts available like *Data Mining and Knowledge Discovery with Evolutionary Algorithms* by Alex A. Freitas (Springer, Berlin, 2002), which discusses the integration of some optimization and data mining techniques.

As one of the main tasks of data mining is “prediction,” it is worthwhile to check some classic texts on forecasting. One of the books we recommend is *Forecasting: Methods and Applications* by Spyros Makridakis, Steven C. Wheelwright, and Rob J. Hyndman (Wiley, Chichester, 1998). The book presents a statistical approach to forecasting: from basic forecasting tools, through time series decomposition and particular methods (e. g., exponential smoothing, regression), to judgmental forecasting.

6 Modern Optimization Techniques

"I have frequently gained my first real insight into the character of parents by studying their children."

The Adventure of the Copper Beeches

"The nature of his tactics suggested his identity to me, and this physical peculiarity – he was badly bitten in a saloon-fight in Adelaide in '89 – confirmed my suspicion."

The Disappearance of Lady Frances Carfax

Whether in banking, manufacturing, or retail, there is scarcely an industry where the term "optimization" does not apply. This is due to the fact that every industry strives for excellence (as there are continual pressures to reduce cost and increase efficiency) and so over the years many optimization techniques have emerged to help managers find better solutions to their business problems. The field of operations research, in particular, developed many techniques to address the complexity of scheduling people, machines, and materials. We often refer to these optimization techniques as "classic" techniques, with the best examples being linear programming, branch and bound, dynamic programming, and network flow programming.

During the last decade, however, we have witnessed the emergence of a new class of optimization techniques that people have termed "modern heuristics." These modern techniques include (among others) simulated annealing, tabu search, and evolutionary algorithms, and they are the main focus of this chapter.

6.1 Overview

Irrespective of the optimization technique used, three things always need to be specified: (1) the representation of the solution, (2) the objective, and (3) the evaluation function. Let us consider each of these in turn.

The representation of a solution will determine the search space and its size. This is an important point, because the size of the search space (i.e., the number of possible solutions to the problem) is not determined by the problem, but by its *representation*. Consequently, choosing the right search space is of paramount importance. If we do not select the correct domain to begin with, we might actually preclude ourselves from ever finding the right solution!

Once we have defined the search space, we need to decide what we are looking for. What is the objective of our problem? This is a mathematical statement of the task to be achieved. It is not a function, but an expression. For example, suppose we wanted to discover a good solution to a traveling salesman problem. The objective would be to minimize the total distance of the route while satisfying the problem constraints. After the objective has been clearly defined, the next thing to do is create an *evaluation function* that allows us to compare the quality of different solutions. Some evaluation functions produce a ranking for various solutions (called *ordinal* evaluation functions), while others are *numeric* and provide a ranking and a quality measure score as well.

In the traveling salesman problem, a numeric evaluation function might map each solution to a distance. By comparing the distance of various possible solutions, we can easily tell if one solution is better than another and by *how much*. However, it might be computationally expensive to calculate the exact distance of each particular solution. In such cases, it might only be necessary to know approximately how good or bad a solution is, or if it compares favorably or unfavorably with some other solution. Such an ordinal evaluation function might evaluate two possible solutions and merely give us an indication as to which solution is favored.

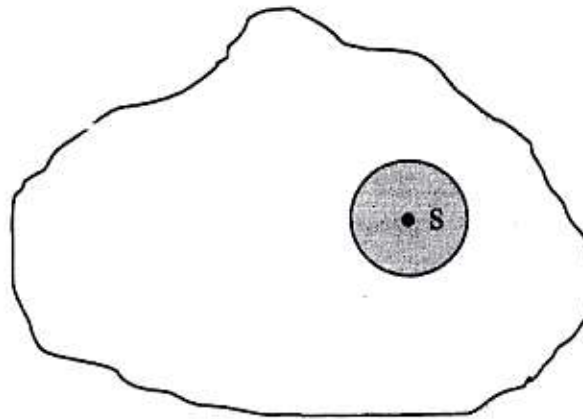
Because the evaluation function is not provided with a problem, how should we go about choosing the correct evaluation function? Oftentimes, the objective can suggest a particular evaluation function. In the traveling salesman problem, for instance, we considered using distance as the evaluation function. This corresponds to the objective of minimizing the total distance of the route. Hence, the objective naturally suggests an evaluation function for finding the best solution. When designing the evaluation function, it is also important to keep in mind that most of the solutions we are interested in will be in a small subset of the search space (because we are only interested in *feasible* solutions – i.e., solutions that satisfy the problem-specific constraints).

Once all of these steps are complete, we can begin searching for a solution. Note, however, that the optimization technique¹⁸ does not know what problem we are trying to solve! All it “knows” is the representation of the solution and the evaluation function. If our evaluation function does not correspond to the objective, then we will be searching for the right answer to the wrong problem!

In any search space, the goal is to find a solution that is *feasible and* better than any other solution present in the entire search space. The solution that satisfies these two conditions is called a *global optimum*. Because finding a global optimum is extremely difficult, a much easier approach is to find the best solution in a subset of the search space.¹⁹ If we can concentrate on a region of the search space that is “near” some particular solution, we can describe this as looking at the *neighborhood* of that solution. Graphically, let us consider some abstract search space with a single solution s :

¹⁸ *Optimization technique* and *search technique* are considered synonymous. The search for the best feasible solution is both an optimization problem and a search problem.

¹⁹ This observation forms the fundamental basis of many optimization techniques.



Our intuition might tell us that solution s is in a neighborhood of the search space where all solutions are very similar to one another. Consequently, we can use a “neighborhood” or “local” optimization technique to find the best solution in this neighborhood. The sequence of solutions that these techniques generate while searching for the best possible solution relies on *local* information at each step of the way.

Local optimization techniques present an interesting trade-off between the size of the neighborhood and the efficiency of the search. If the size of the neighborhood is relatively small, then the algorithm may be able to search the entire neighborhood quickly. Only a few potential solutions may have to be evaluated before a decision is made on which new solution should be considered next. However, such a small neighborhood increases the chance of becoming trapped in a local optimum! This suggests using large neighborhoods, as a larger range of visibility makes it easier for the algorithm to decide where to search next. In particular, if the visibility were unrestricted (i.e., the size of the neighborhood were the same as the size of the whole search space), then eventually we would find the best series of steps to take. However, the number of evaluations might become overwhelming and impossible to compute.

All optimization techniques (whether local optimization techniques, ant systems, or evolutionary algorithms) generate new solutions from existing solutions. The main difference between these different techniques lies in how these new solutions are generated. Because we can only sample a small fraction of the search space (otherwise the computation time would be billions of years!), we should be economical in the process of generating and evaluating new solutions.

To put some of these concepts into context, let us return to the car distribution example and assume that we want to distribute 3,000 cars to 50 auction sites. Clearly, many possibilities exist for representing a "solution." For example, we can assign an index number from 1 to 50 for each auction site, and a solution can be a vector of 3,000 numbers: the first number represents the destination of the first car, the second number represents the destination of the second car, and so forth:

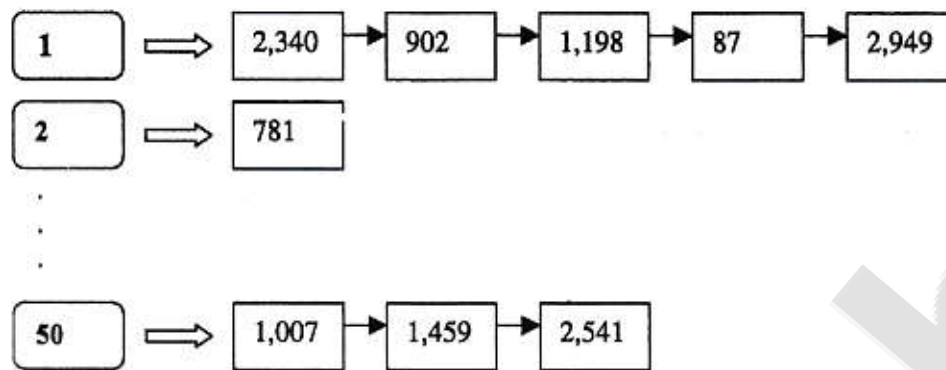
23	41	5	...	19	41
----	----	---	-----	----	----

The above vector represents a solution where the first car is shipped to auction site 23, the second car is shipped to auction site 41, the third car is shipped to auction site 5, and so on, with the last two cars being shipped to auction sites 19 and 41 respectively. Of course, the auction numbers should not be assigned randomly. When we discuss some optimization techniques in the following sections, the advantages of assigning "close" numbers to "close" auctions will become clear.

Note, however, that representing a solution in this way has a couple of disadvantages. First of all, this representation implies an enormous search space that is too time consuming to search. We have 50 possible destinations for each car, so the number of possible distributions for 3,000 cars is $50 \times 50 \times 50 \times \dots \times 50$ (i. e., 50 multiplied by itself 3,000 times!). The size of this search space can be reduced significantly by using a different representation.

The second disadvantage of this representation is that it makes some constraint handling difficult. Recall that the car distribution problem includes many soft and hard constraints, such as inventory level limits, exclusion conditions (e. g., "the total transportation distance for each car must not exceed 700 miles"), and so forth. If the above vector of auction indices were used to represent the solution, then many randomly generated solutions would be infeasible. We could reject these infeasible solutions, lower their quality measure score, or attempt to "repair" them by replacing some values in the vector with new ones. For example, if the auction site for the second car is 41 and it corresponds to Jacksonville, Florida (which is more than 700 miles from the current location of the car), then we could try to replace auction 41 with some other auction site that is closer to the location of the car. Note also that most new solutions would be infeasible with this representation, making the search process less efficient. In this particular case, other representations exist that can make constraint handling easier.

Clearly, many other representations are possible; for instance, we can create a linked-list structure of 50 nodes, where each node represents an auction site and has a list of cars "assigned" to this auction. With this representation, it would be much easier to handle certain constraints (e. g., inventory constraints, as the length of each node implies the number of cars assigned to that particular auction):

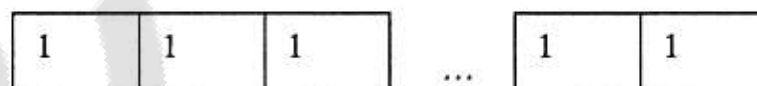


The above vector represents a solution that would send cars 2,340, 902, 1,198, 87, and 2,949 to auction 1, car 781 to auction 2, and so on, with cars 1,007, 1,459, and 2,541 going to auction 50. By using this type of representation, the size of the search space can be significantly reduced by imposing some inventory limits (e. g., each auction site should have at least 20, but no more than 100 cars). Additionally, if some auction sites do not admit cars of a particular type (e. g., high mileage cars), then it would be much easier to check (or enforce) such constraints.

Another possibility is based on indirect representation and some preprocessing. Here we would sort all the available auction sites by *distance* from a particular car, i. e., auction 1 would be the closest (distance-wise), auction 2 would be the second closest, and so forth. Although this representation looks very similar to our first representation, the interpretation is very different:



This vector represents a solution that ships the first car to the closest auction site, the second car to the third-closest auction site, etc. Note that the *same* numbers in the above representation (e. g., number 1) correspond to *different* auction sites! Again, there are several advantages of using this representation. First, the vector:



represents a solution where *each* car is sent to the closest auction site. If we believe that transportation costs play a major role in the decision-making process, then the above solution may represent a reasonable “first draft.” Second, the numbers are meaningful in the sense that they correspond to distances. If for some reason auction 5 is not available (e. g., because of inventory limits), then we can direct the car to auction 6, thereby increasing the transportation distance only slightly. The third advantage is that preprocessing can help us handle many constraints. For example, if a car is red and auction 13 does not admit red cars, then

we can eliminate 13 from the list of available auction sites for this car. Also, if we limit the transportation distance for any car, all we have to do is truncate the auction list to eliminate those sites that exceed the threshold. By doing this, many constraints (e. g., exclusions based on mileage, color, distance) can be handled during the preprocessing stage!²⁰

Note again, that the representation of a solution will define the search space and its size, and that we can define a neighborhood for any solution in any representation. If we assume a solution is represented by a vector of 3,000 numbers, with each number corresponding to an auction site, then for a solution:

23	41	5	...	19	41
----	----	---	-----	----	----

we may define its neighborhood as a collection of all solutions that are identical except for one auction site being different by one (e. g., 23 can be replaced by either 22 or 24). Hence, the following solution:

23	41	6	...	19	41
----	----	---	-----	----	----

is a neighbor of the original solution. Note that the size of the neighborhood is 6,000 solutions, as there are two possible replacements for each auction (23 can be replaced by 22 or 24; 41 can be replaced by 40 or 42; etc.).²¹

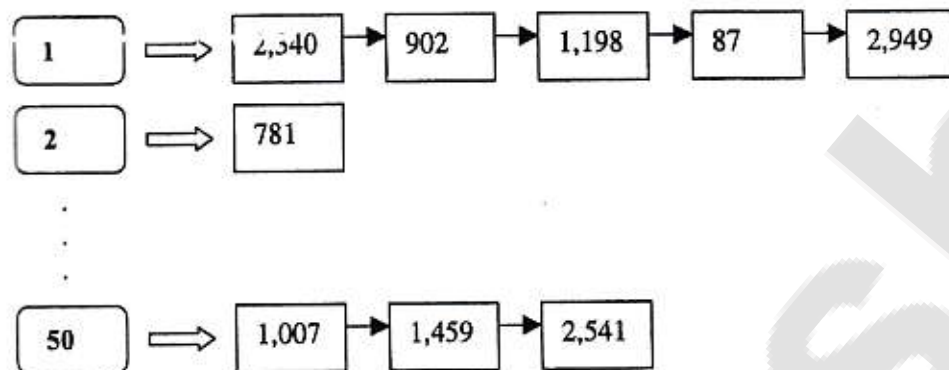
Of course, there are many other possibilities.²² For example, if an auction site were allowed to differ by five (rather than just one), then the neighborhood would be much larger. Each auction site would define 10 possible neighbors (e. g., auction 23 could be replaced by any of the following auctions: 18, 19, 20, 21, 22, 24, 25, 26, 27, 28), so the size of the neighborhood would be 30,000. Alternatively, we can stick to the requirement that an auction site can only differ by one, but relax the restriction on the number of auction sites that can differ! In such a scenario, if any auction site can differ by one (or stay as it was), the size of such a neighborhood would be $3 \times 3 \times 3 \times \dots \times 3$ (3,000 multiplications!) Of course, if we allow bigger changes (e. g., replacing auction 5 with auction 19), then the size of this huge neighborhood would grow even further!

²⁰ Using this representation, we have to build a list of all *feasible* auctions for each car. Although this preprocessing might be computationally expensive, we do it only once, at the beginning of the search. The general rule of thumb is that preprocessing is useful: the more sweat during exercise, the less blood during combat! Also, we will return to the subject of constraint handling in Sect. 6.6.

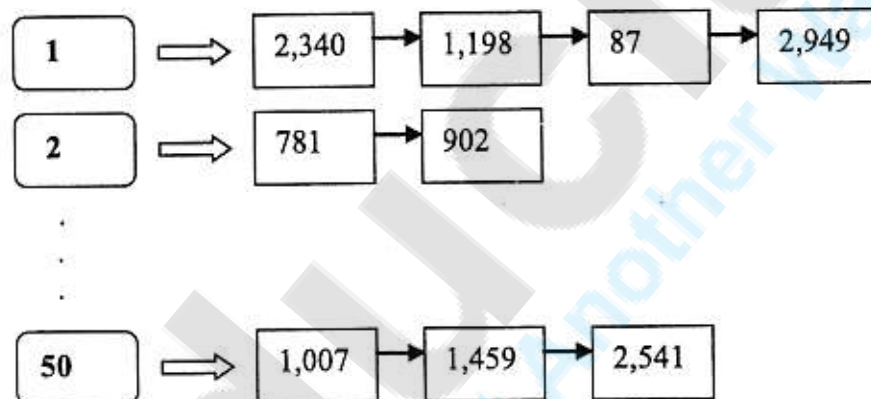
²¹ For some vectors, however, the neighborhood size is slightly less than 6,000 (e. g., auction sites 1 and 50 can only be replaced by 2 and 49 respectively).

²² The typical methods for defining neighborhoods are either based on distance or on some transformation operator.

The linked-list representation offers another possibility. For a solution:



we can define a neighbor as a new solution derived by changing the destination of one car. For example, if we move car 902 from the first auction site to the second, then we would get the following neighboring solution:



In this scenario, the size of the neighborhood is much smaller than in the previous example: there are 49 “other” auctions available for each car, so the number of neighbors is only 147,000 (i. e., $49 \times 3,000$). Again, we can change the size of the neighborhood by allowing some other transformations. For example, if we define a neighbor as a solution obtained by swapping the assignment of two cars (e. g., swapping the assignment of cars 87 and 1,007), then the number of possible neighbors would be less than 4,498,500 (i. e., $3,000 \times 2,999/2$), as swapping the assignment of some cars (e. g., cars 2,340 and 87) does not lead to a new solution.

During different stages of different optimization techniques, it is necessary to compare two different solutions and determine the better one. Hence, we must be able to evaluate any solution and assign a quality measure score to it. If the quality measure score is 123.76 for one solution and 119.92 for another, then we would like to assume that the former solution is better. In the car distribution example, it is necessary to build an evaluation procedure that returns a quality measure score for any solution. However, this task is not trivial; for example, how can we evaluate a solution:

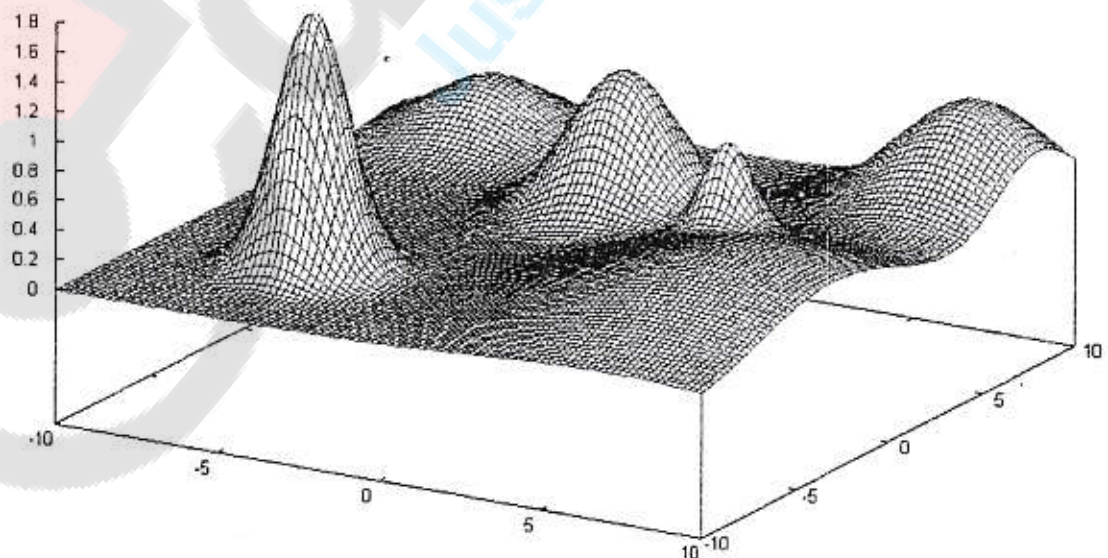
23	41	5	...	19	41
----	----	---	-----	----	----

that assigns the first car to auction 23, the second car to the auction 41, and so on? Clearly, several things must be considered: the predicted sale prices of these cars at the auction sites (taking into account the time delay caused by transportation), transportation costs, "penalties" for violation of various constraints (e. g., a red car is shipped to an auction that does not admit red cars), and so forth. Quite often, there would be many trade-offs to consider (e. g., by sending a red car to a particular auction site we would violate a constraint, but on the other hand we would save a lot on the transportation cost ...), which should be reflected in the evaluation function.

6.2 Local Optimization Techniques

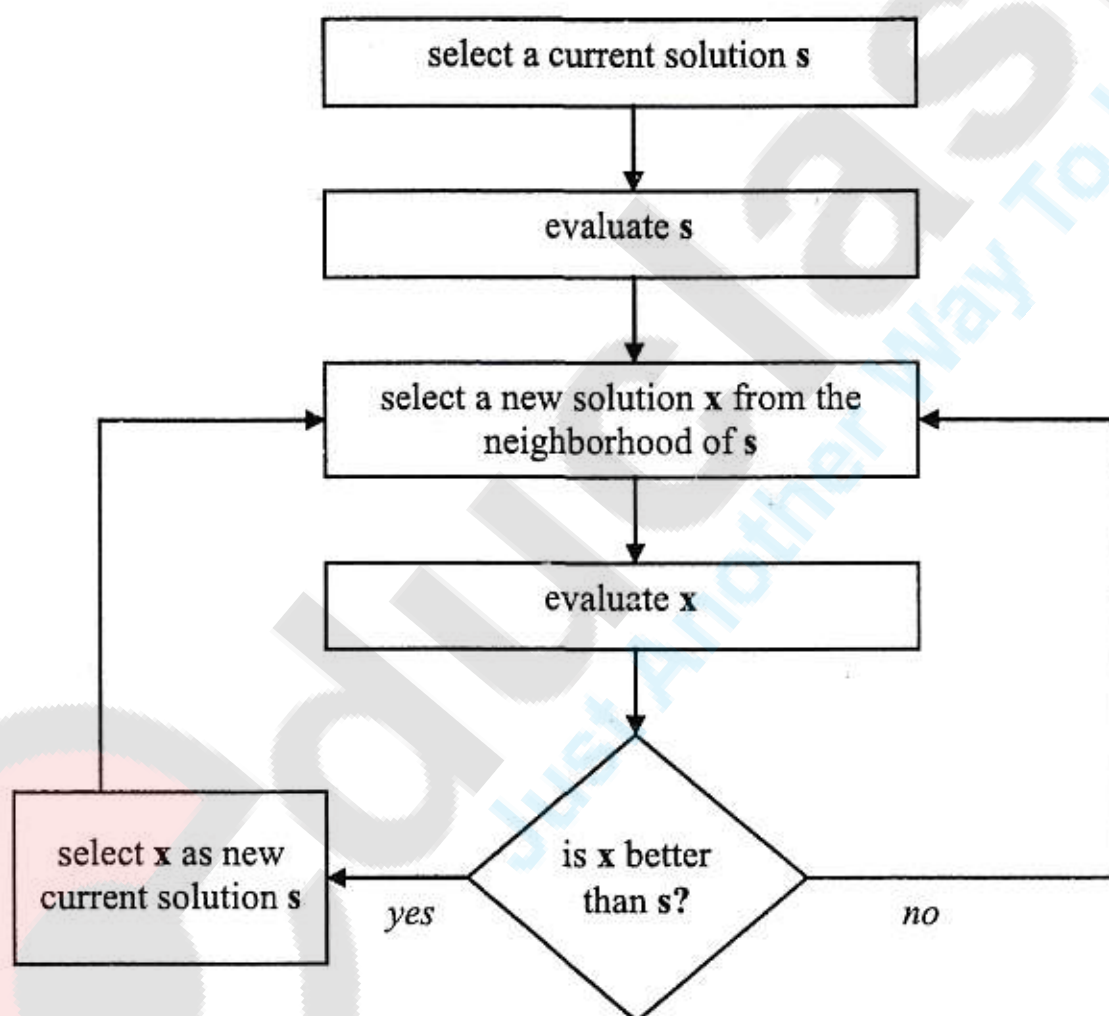
The evaluation function defines a *quality measure score landscape* (also known as a *response surface* or *fitness landscape*) that is much like a topography of hills and valleys. Within this three-dimensional landscape, the problem of finding a solution with the highest quality measure score is similar to searching for a peak in a foggy mountain range. Because our visibility is limited, we can only make local decisions about where to go next. If we always walk uphill, we will eventually reach a peak, but this peak might not be the highest peak in the mountain range; it might just be a "local" optimum. We may have to walk downhill for some period of time to find a path that will eventually lead us to the highest peak (i. e., the "global" optimum).

The quality measure score landscape for a two-variable function is illustrated below. The graph displays the quality measure score for every pair of values for the first and second variable, which allows us to visualize the mountain ranges, highest peaks, local optima, etc.:



Keeping this illustration in mind, let us examine a basic local optimization procedure called *hill climbing*,²³ and its connection with the “neighborhood” concept. Like all local optimization techniques, hill climbing uses iterative improvement. The technique is applied to a single solution (i.e., the current solution) in the search space. During each iteration, a new solution is selected from the neighborhood of the current solution. If that new solution has a better quality measure score, then the new solution becomes the current solution. Otherwise, some other neighbor is selected and tested against the current solution. The technique terminates if no further improvements are possible, or when the allotted time runs out.

A simple flowchart of a hill-climbing sequence is given below:



Note that this flowchart expresses only the general principle of hill climbing without any termination conditions. We have to start with some (possibly randomly generated) solution s , evaluate it, and then generate a new solution x from

²³ The term *hill climbing* implies a maximization problem, but the equivalent *descent* method is easily envisioned for minimization problems. For convenience, the term will be used to describe both methods without any implied loss of generality.

a neighborhood of s . If the new solution x is better than s , then we take an uphill step (i. e., we accept this new solution as the current solution, and try to improve it further by generating yet another new solution from the neighborhood of the current one). On the other hand, if the new solution x is not better than s , we generate another new solution and we repeat this process several times until either (1) the whole neighborhood has been searched, or (2) we have exceeded the threshold of allowed attempts (which is missing from the flowchart). At this stage, we can exit the loop and report the current solution as the best solution, or we can store the current solution in “memory” and restart the whole process, hoping that the next hill-climbing iteration (which starts from a new solution) may produce a better overall solution (a process called *iterated hill-climbing*).

It is clear that such hill-climbing techniques can only provide locally optimum values that depend on the starting solution. Moreover, there is no general procedure for measuring the relative error with respect to the global optimum because it remains unknown. Given the problem of converging on locally optimal solutions, we often have to start the hill-climbing algorithm from a large variety of different solutions. The hope is that at least some of these initial locations have a path that leads to the global optimum. We might choose the initial solutions at random, or we might base them on some grid, regular pattern, or other available information (perhaps using the search results from somebody else’s effort to solve the same problem).

The success or failure of a single iteration (i. e., one complete climb) of the hill-climbing algorithm is determined completely by the initial solution. For problems with many local optima, it is often very difficult to find the global optimum. Consequently, hill-climbing techniques have several weaknesses:

- They usually terminate at solutions that are only locally optimal.
- There is no information as to how much the discovered local optimum deviates from the global optimum, or perhaps even from other local optima.
- The optimum that is obtained depends on the initial configuration.
- In general, it is *not* possible to provide an upper bound for the computation time.

On the other hand, there is a tempting advantage to using hill-climbing techniques: they are very easy to apply! All that is needed is the representation, the evaluation function, and a measure that defines the neighborhood around a given solution.

Effective optimization techniques provide a mechanism for balancing two apparently conflicting objectives at the same time: *exploiting* the best solutions found so far, and *exploring* the search space. Hill-climbing techniques exploit the best available solution for possible improvement, but they neglect exploring a large portion of the search space. In contrast, a random search (where various solutions are sampled from the entire search space with equal probability) explores the search space thoroughly, but foregoes exploiting promising regions of the space. Each search space is different, and even identical spaces can appear very different under different representations and evaluation functions. As a result,

there is no way to choose a single optimization technique that performs well in every case (more on this topic in Sect. 10.2).

Let us illustrate the hill-climbing technique on the car distribution example. Say we would like to implement an iterative hill-climbing algorithm that would generate a car distribution recommendation. Using the first representation (i. e., where a vector of 3,000 values provides indices of auction sites from 1 to 50) and defining a "neighbor" as a solution that differs (at most) by 1 on any position, the hill-climbing algorithm would work as follows.

First, the algorithm would generate a starting solution. This solution might be generated randomly (i. e., for each entry, a random number from 1 to 50 is produced) or we can accept some heuristic-based solution (e. g., an initial solution that assigns each car to the nearest auction site). Either way, let us assume that the initial solution is:

23	41	5	...	19	41
----	----	---	-----	----	----

The algorithm then evaluates this solution and assigns a quality measure score to it. For this example, let us assume that the above solution generates a quality measure score of 171.49. Now we are ready to do some "hill-climbing"! The algorithm generates a neighbor solution by generating some random locations in the vector (any number of locations from 1 to 3,000) and then changing the selected indices in these locations by one (increment or decrement). Assume that the generated solution is (i. e., the selected first, second, ..., and 3,000th location was increased or decreased by one.):

24	41	6	...	19	40
----	----	---	-----	----	----

Next, the evaluation of this solution is needed. If the evaluation produces a quality measure score higher than the original solution (e. g., 176.18), then the algorithm will accept this new solution as the current solution and continue. Note that this new solution (with a higher quality measure score) has its own new neighborhood, and the subsequent new solution is drawn from this new neighborhood. Any acceptance of a new solution means that the algorithm found a better solution and made a step uphill. However, it may happen that the quality measure score of the new solution is lower than the current solutions (e. g., 169.83). In such a case, the algorithm will discard this solution (we are not interested in inferior solutions) and generate another solution from the neighborhood of the current solution. Say the next solution is:

24	42	5	...	18	40
----	----	---	-----	----	----

Again, if there is an improvement in the quality measure score, then the algorithm will accept this solution and continue. If not, the algorithm will generate another solution from the original neighborhood ...

Note also that a hill-climbing algorithm can (a) accept the first solution found that is better than the current one (as presented above), or (b) accept the best solution found in the whole neighborhood. These two possibilities represent two extremes, with plenty of "in between" possibilities (e.g., we can accept the best solution found from 100 generated solutions in the neighborhood).

The question is, how long should the hill-climbing algorithm generate random solutions before giving up? Well, we usually have a counter responsible for counting the algorithm's attempts to improve the current solution. Each time the algorithm finds an improvement the counter is reset to zero. However, if the hill-climbing algorithm experiences a long sequence of unsuccessful attempts, we stop the search upon exceeding a predefined threshold. In this particular example, what should the threshold be? The answer depends on a few factors, with the size of the neighborhood being the most important. It is difficult to claim that we have found the "local optimum" if we did not search the whole neighborhood, but the size of the neighborhood might be too large to evaluate all the neighbors! This problem can be resolved by defining a neighborhood differently. For example, if a neighbor differs from the current solution by only 1 on *one* location, then we will have up to 6,000 neighbors for each current solution and can evaluate all of them before giving up.

In summary, if it is feasible (time wise) to search the whole neighborhood before arriving at the local optimum, then we do not need a counter for controlling the number of unsuccessful attempts because *all* the solutions in the neighborhood will be searched. However, if it is not feasible to search the whole neighborhood, we have to settle for a counter and quit our search after some number of unsuccessful attempts. In our case, let us assume we quit the search after 100,000 unsuccessful attempts.

Returning to our example, say we arrive at the following solution after many iterations and improvements, and all attempts to improve it have failed:

27	31	9	...	45	29
----	----	---	-----	----	----

Note the significant number of improvements the algorithm went through: the original assignment for car 2,999 (second to last position in the vector) changed from auction 19 to auction 45, and all changes were made by adding or subtracting 1. Anyway, in all likelihood we have arrived at the local optimum, and this solution is the outcome of our hill-climbing exercise. The quality measure score is 345.67 and we are confident about the solution's quality. After all, 100,000 neighboring solutions failed to produce any improvement!

However, we are not sure if this is the best solution. If we started our hill-climbing exercise from a different solution (which might be located in a very

"different area" of the search space), we might finish with a local optimum solution that looks like:

43	41	32	...	15	27
----	----	----	-----	----	----

and has a quality measure score of 1,457.81 (which is *much* better than the solution we discovered earlier!).

Recall our earlier discussion on the "hills and valleys" in a quality measure score landscape. Clearly, there are many hills (local optimum solutions) and the hill-climbing algorithm will produce a solution that represents one of these hills. However, the problem is that we do not know whether there are other (possibly much higher) hills somewhere else! And the size of the neighborhood corresponds to our "visibility" during the search: the larger the neighborhood, the better the visibility, and the better chances of discovering the highest peak! However, it might not be feasible to search the whole neighborhood if it is too large ...

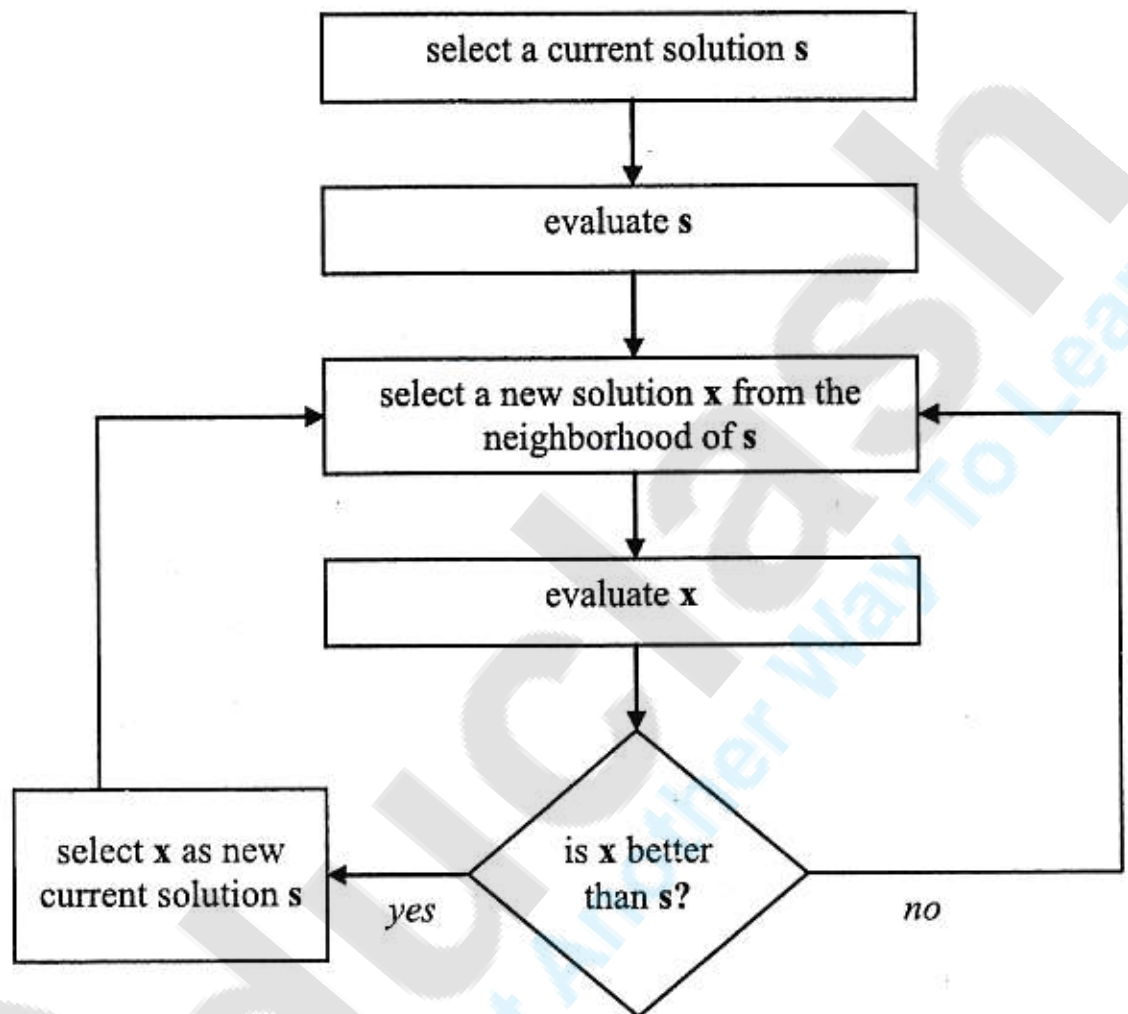
So, what should we do? We can restart our hill-climbing algorithm several times, each time from a different (possibly random) location, and hope that one of these runs will provide us with the global optimum solution (which may or may not happen).

6.3 Stochastic Hill Climber

Getting stuck in local optima is a serious problem. It is one of the main deficiencies of numerical optimization applications, as almost every solution to a real-world problem in factory scheduling, demand planning, land management, and so forth is at best only locally optimal.

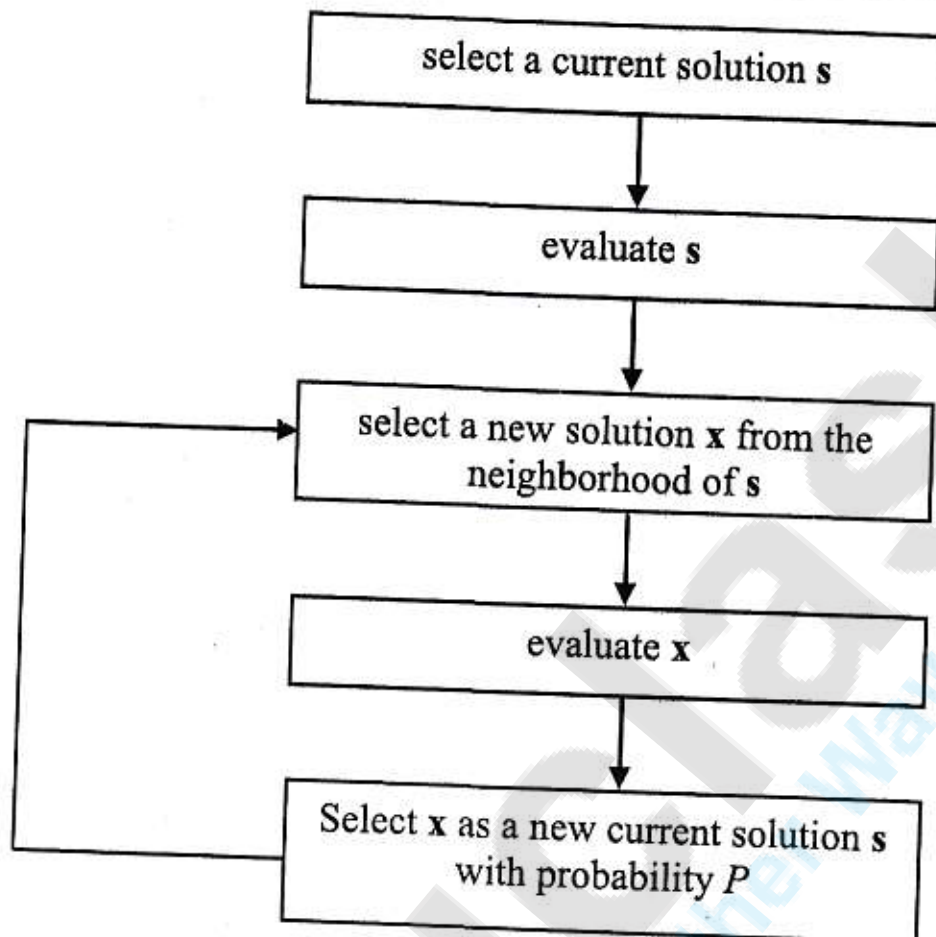
So what can we do about it? How can we design an optimization technique that has a chance to escape local optima, to balance exploration and exploitation, and to make the search independent from the initial configuration? There are a few possibilities, and we will discuss some of them in this chapter, but keep in mind that the proper choice is always dependent on the problem. One option, as we discussed earlier, is to execute a large number of initial configurations for the chosen technique. Moreover, it is often possible to use the results of previous attempts to improve the initial configuration for the next attempt. We have already seen one possibility of this in the previous section, where we discussed a procedure called the "iterated hill climber." After reaching a local optimum, the search is restarted from a different starting solution. Although we can apply this strategy to other algorithms, let us discuss some other possibilities of escaping local optima within a single run of an algorithm. One way of accomplishing this is by modifying the criteria for accepting new solutions that correspond to a *negative* change in the quality measure score. That is, we might want to accept an inferior solution from the local neighborhood in the hope that it will eventually lead us to something better.

To turn an ordinary hill climber into such an algorithm, a few modifications are required. First, let us recall the detailed structure of a hill climber:



Note again that the inner loop *always* returns the local optimum. The only way for this technique to "escape" local optima is by starting a new search (outer loop) from a new (random) location. After some maximum number of attempts, the best overall solution is the final outcome of the algorithm.

By modifying this procedure so that acceptance of a new solution is dependent upon some probability – which is based on the difference between the quality measure score for these two solutions – we obtain a new technique called the *stochastic hill climber*:



The slight (but significant) difference between an ordinary and stochastic hill climber lies in a single box inserted in the flowchart that replaces the condition box. During the execution of the hill climber's internal loop (where the hill-climbing searches for a better solution in the neighborhood of the current one), only a superior solution is accepted as a new current solution. On the other hand, the same internal loop in the stochastic hill climber procedure may accept an inferior solution as a new current solution. This feature does not appear in local optimization techniques. This insertion represents a probabilistic decision on the acceptance of a new solution (as opposed to a deterministic decision in classic hill climbers), and is done to escape local optima ...

Let us discuss this feature carefully. A new solution x is accepted with some probability P , which means that the rule of moving from the current solution to a new neighbor is probabilistic. Consequently, it is possible for the newly accepted solution x to be *inferior* to the current solution s , and it is also possible that a superior solution will *not* be accepted! This probability of acceptance depends on the quality measure score difference between these two solutions, as well as on the value of an additional parameter T (which remains constant during the execution of the algorithm).

Rather than providing a mathematical function for calculating the values of probability P (which is based on a constant value of parameter T), we will instead explain how this function works. In general terms, the probability function is constructed in a such way that:

- If the new solution x has the same quality measure score as the current solution s , then the probability of acceptance is 50% (it does not matter which one is chosen, because each is of equal quality).
- If the new solution x is superior, then the probability of acceptance is greater than 50%. Moreover, the probability of acceptance grows together with the (negative) difference between these two quality measure scores.
- If the new solution x is inferior, then the probability of acceptance is smaller than 50%. Moreover, the probability of acceptance shrinks together with the (positive) difference between these two quality measure scores.

The probability of accepting a new solution x also depends on the value of parameter T , and the general principle is as follows:

- If the new solution x is superior, then the probability of acceptance is closer to 50% for *high* values of parameter T , or closer to 100% for *low* values of parameter T .
- If the new solution x is inferior, then the probability of acceptance is closer to 50% for *high* values of parameter T , or closer to 0% for *low* values of parameter T .

This is interesting, because it means that a superior solution x would have a probability of acceptance of *at least* 50% (regardless of the value of parameter T). Likewise, an inferior solution would have a probability of acceptance of *at most* 50% (varying between 0% for low values of T and 50% for high values of T). The general conclusion is clear: The lower the value of T , the more the algorithm behaves like a classic hill climber that rejects inferior solutions and accepts superior ones. On the other hand, if the value of T is very high, then the algorithm resembles a random search, because the probability of accepting inferior or superior solutions is close to 50%. Thus, we have to find a value for parameter T that is neither too low nor too high for a particular problem.

The stochastic hill climber technique is also a forerunner to another optimization technique called *simulated annealing*, which is covered in the next section.

6.4 Simulated Annealing

The simulated annealing technique (also known as Monte Carlo annealing, statistical cooling, probabilistic hill-climbing, stochastic relaxation, and the probabilistic exchange algorithm) is based on an analogy taken from thermodynamics. To grow a crystal, we begin by turning the raw material into a molten state through heating. Then we reduce the temperature of this crystal melt until the crystal structure is frozen. However, if the cooling process is done too quickly, then the results

are detrimental. In particular, some irregularities are locked into the crystal structure and the trapped energy level is much higher than in a perfectly structured crystal.²⁴ The analogy between the physical system and an optimization problem is evident; the basic "equivalent" concepts are listed below:

- State – feasible solution
- Energy – evaluation function
- Ground state – optimal solution
- Rapid quenching – local search
- Temperature – control parameter T
- Careful annealing – simulated annealing

Simulated annealing is similar to a stochastic hill climber in that it may accept an inferior solution as a new current solution, and the acceptance decision is based on the value of parameter T . However, unlike the stochastic hill climber (which has a fixed value for parameter T), simulated annealing changes the value of parameter T (commonly referred to as *temperature*) during the run. Simulated annealing starts with high values of parameter T – making the process similar to a random search – and then gradually decreases this value during the run. The value of parameter T is quite small toward the end of the run, so the final stages of simulated annealing resemble an ordinary hill climber. Another difference between the stochastic hill climber and simulated annealing is that the latter always accepts superior solutions. Recall from the previous section that the stochastic hill climber used some probability for accepting both inferior *and* superior solutions, which is not the case in simulated annealing.

The following flowchart represents a simulated annealing algorithm:

²⁴ A similar problem occurs in metallurgy when heating and cooling metals.